ASIC Design 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 based on your N-bit Ripple Carry Adder design from Lab 2
- Use code coverage analysis to evaluate and improve a starter 16-bit Ripple Carry Adder test bench
- Design and Verify flexible and scalable versions of the Serial-to-Parallel and Parallel-to-Serial Shift Register designs you drew RTL diagrams for in the Lab 1 postlab
- 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 add this new workspace into your 337 Repository, like you did in Lab1. This way, you will always have the original copy in storage.

2. 16-Bit Adder Circuit

2.1. Using your N-Bit Adder to make a 16-Bit Adder

Make a copy of your 8-bit adder wrapper file and update it to be for a 16-bit adder instead of an 8-bit one. Symbolic links to your 1-bit, n-bit, and 8-bit adder files from your Lab 2 folder have been created into your Lab 3 folder for you by the 'setup3' script. A symbolic link is a 'soft' copy of a file where that 'copy' is actually just a reference/link to the original one so that the contents of the original are automatically updated/shared between both the original and the link/reference.

The required module name is: adder_16bit

The required filename is: adder_16bit.sv

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

Signal	Direction	Description
a[15:0]	input	One of two primary inputs.
b[15:0]	input	Second of two primary inputs.
carry_in	input	The overflow value carried in from a prior addition column
sum[15:0]	output	The computed sum value.
overflow	output	The overflow value from the calculation

Once you have finished creating your 16-bit adder design, create a simple test bench that just tests a couple pairs of numbers for inputs a, b, and carry_in. **Do not try to extend your prior exhaustive adder test benches for the 16-bit adder as with 16-bit inputs for a and b there is over 8 billion unique permutations of input values which is infeasible (and effectively impossible with the prior test benches) to run a simulation for.**

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>)
      $info("<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".*
- *<Error message> is a user defined error Message. An Example is: "DUT's output does not match the expected output".*

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 $info() code and the semicolon after it. The assert command is treated as an "if" statement with two changes: The "true" branch can be omitted and $error will be called if the "false" branch is not specified.

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

Note: The "#(2)" in front of the assert forces the assert to be delayed to be 2 timescale units after the input change, skipping periods during which the sum may contain glitches/transient behavior. This requires the same timescale settings as in the test bench to be added to the design file.

Now update your 1-bit adder to use internal assertions to check its inputs and update your generate based structural N-bit adder design to use assertions to check that each 1-bit adder is functioning correctly. Also update both your N-bit and 16-bit adders to use internal assertions to check for proper input values. 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 exhaustive test benches from Lab 2.
- To help with visually identifying problem test cases the test bench 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.

Remember that 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 and N-bit adder designs (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), move on to the next section.

3. 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.

3.1. 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.

3.2. 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.

3.3. 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.

3.4. Coverage Reporting

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

1. Clear your work area in Questasim
   make clean
   vsim -i
2. Compile > compile options(tab)
   Make sure the options are selected as shown in Figure 2 to the right.
3. Close Questasim and rerun the make simulation target.
   (i.e. make sim_full_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 Questasim will display the report.

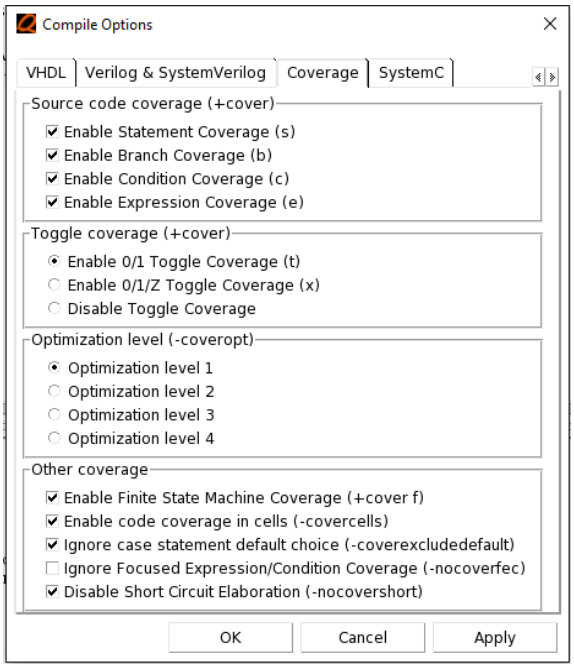

Figure 2: Coverage Compile Options

6. View the statements/Expressions that are not covered by double-clicking the design name (DUT) in "sim" tab.

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

3.5. 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 0, and the text file report of coverage shows 100% coverage of the design files during both a source simulation (adder_1bit.sv & adder_nbit.sv & adder_16bit.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 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.

4. Creating Flexible and Scalable Sequential Designs

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

4.1.1. N-bit Serial-to-Parallel Shift Register Design

Using parameters and your RTL diagrams from Lab 1's postlab design a flexible Serial-to-Parallel Shift Register design that has 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 defines the shifting direction of the register such that a Boolean value of 'true' assumes input serial data is sent most significant bit first and 'false' assumes input serial data is least significant bit first.

The default values for the parameters for this design should make it the implementation of your 4-bit MSB Serial-to-Parallel Shift Register RTL diagrams from Lab 1's post lab.

The required module name is: flex_stp_sr
The required filename is: flex_stp_sr.sv

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 (the common serial data bus idle value of '1').
shift_enable	Input	This is the active-high enable signal that allows the shift register to shift in the value of the serial_in signal.
serial_in	Input	This is the serial input signal and thus will contain the value to be shifted into the register. The default/idle value for this signal is a logic '1', since this is the common idle value used in serial communications.
parallel_out[#:0]	Output	This is the data that is currently held by the shift register. The actual port declaration should use the NUM_BITS parameter value to determine the value of the '#'.

4.1.2. 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. 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 parameterized serial-to-parallel shift register design (not including the source folder)

4.1.3. 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
```

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

4.2.1. N-bit Parallel-to-Serial Shift Register Design

Using parameters and your RTL diagrams from Lab 1's postlab design a flexible Serial-to-Parallel Shift Register design that has 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 defines the shifting direction of the register such that a Boolean value of 'true' results in shifting data out most significant bit first and 'false' results in shifting data out least significant bit first.

The default values for the parameters for this design should make it the implementation of your 4-bit MSB Parallel-to-Serial Shift Register RTL diagrams from Lab 1's post lab.

The required module name is: flex_pts_sr
The required filename is: flex_pts_sr.sv

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 (the common serial data bus idle value of '1').
shift_enable	Input	This is the active-high enable signal that allows the shift register to shift out the next value for the serial_out port.
load_enable	Input	This is the active-high enable signal that allows the shift register to load the value at the parallel_in port. If both the shift_enable and the load_enable are active the load_enable takes priority.
parallel_in[#:0]	Input	This is the data that will be loaded into the shift register when the load_enable signal is active. The actual port declaration should use the NUM_BITS parameter value to determine the value of the '#'.
serial_out	Output	This is the serial output signal and thus will contain the data that is being serially transmitted from the shift register.

4.2.2. 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 parameterized parallel-to-serial shift register design (not including the source folder)

4.2.3. 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
```

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. The output should form a sequence like: {0, 1, …, RO-1, RO, 1, 2,

…}. It activates the value of a rollover flag output port whenever the desired rollover value is reached and clears the value upon the next desired event occurrence. Additionally since the counters are very commonly used in control logic for designs it is imperative to keep its output logic (especially the rollover flag) as simple and fast as possible. Otherwise this delay could end up slowing down a design unnecessarily. *In order to have very fast rollover flag output logic you should design it so that the actual port is simply connected to output of a flip-flop and that all flag decisions are done as next state logic for that flip-flop.*

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.
clear	Input	This is the active-high signal that forces the counter synchronously clear its current count value back to 0.
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 '#'.
count_out[#:0]	Output	This is the current N-bit count value stored in the counter. 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 when the rollover value is reached and cleared with the next increment. *Note: This flag will be used as a control signal later and thus must come directly from a flip-flop.*

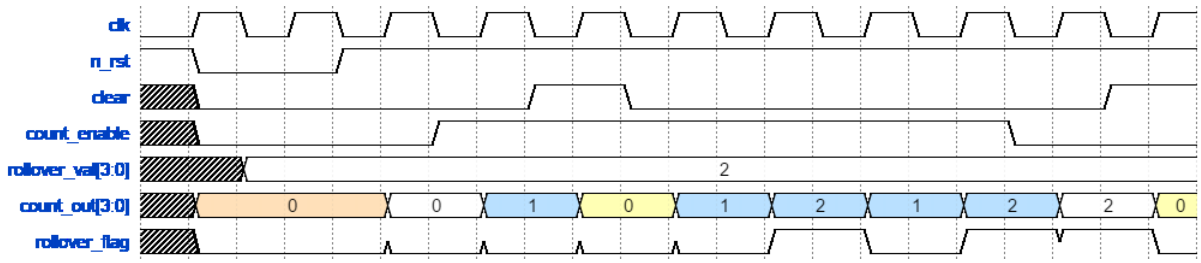

Figure 1: Example 4-bit Counter Timing Waveform Diagram

Note: The orange filled count value results from the asynchronous reset, the yellow values result from the clear pin, the blue values result from counting due to the active count enable, and the white values result from maintaining the current count value.

5.2. Flexible Counter Grading

You will need to create a RTL diagram for the counter design and have it signed off for correctness by a TA. It is expected that you will have this step done before beginning to implement your design and as such, course staff may refuse to help with design debugging if you do not have a valid RTL diagram for this design.

You will need to verify your design using a test bench that fully covers the default parameter value form of your design as well as another test bench for testing a scaled version of your design (using a wrapper file like was used for the 8-bit and 16-bit adder testing). Once you have completed these steps, **submit your design's code and default parameter value form test bench** for grading, use the following command:

```
submit Lab3fc
```

Make sure to test your design using different values for NUM_CNT_BITS.
Also make sure that the tb_flex_counter.sv test bench submitted only tests your flex_counter design with default values otherwise the grading of your test bench may not work properly. Your scaled design should be tested with a separate test bench design/file.