ASIC Design Laboratory
Lab 3 : Introduction to Synthesis Tuning, Structured Debugging, and Non-Exhaustive Testing
Lab Manual

Fall 2020

The purpose of this lab exercises is to help you become familiar with the various synthesis process steps and their optimization/tuning adjustments, structured debugging practices, the design of non-exhaustive test benches, and the evaluation of their effectiveness via code coverage analsysis.

It is important to note, that given the fundamentally parallel and interactive nature of hardware designs, debugging designs described with HDL code requires a method that strictly identifies and leverages guaranteed cause-effect relationships with in the design's description. Other lazy or speculative debugging methods will generally result in vast amounts of wasted time, effort, and frustration and can easily increase debugging times by a factor of 10x.

In this lab, you will perform the following tasks:

- Modify the dirset provided Makefile to tune the synthesis for a 16-bit adder
- Learn how to leverage good waveform organization and test-vector based test benches to efficiently verify non-trivial hardware designs described with HDLs
- 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 Adder test bench that uses a test-vector approach

1 Lab Setup

In a UNIX terminal window, issue the following commands, to setup your Lab 3 workspace:

mkdir –p ~/ece337/Lab3
cd ~/ece337/Lab3
dirset
setup3

The setup3 command is an alias to a script file 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.

IMPORTANT: 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 Synthesis Tuning and Optimization

The designs you synthesized in lab 2 are incredibly simple hardware systems and so will be naturally very fast and easy for the tools to optimize. This will likely result in your three design schematics looking either identical or very similar. When working with more complex designs, such as the provided 16-bit adder design (adder_16.sv), it may become necessary to utilize different synthesis commands in order to guide, and sometimes force, the tools to optimize the design further than initial synthesis attempts in order to meet either area or timing constraints for the design usage. Therefore, you are going to alter your "SYN_CMDS" variable in the makefile so that it will cause Design Compiler® to perform two synthesis passes. In modifying your makefile to accommodate a second compilation pass, you will add a timing constraint to the compile options and instruct Design Compiler® to allow the mapped design to be restructured. In order to apply the timing constraint, you will have to use the command 'set_max_delay', which has the following syntax:

```
set_max_delay <Delay> -from "<Input>" -to "<Output>"
```

where:

- <Delay> is the numerical value of the delay you wish to obtain
- <Input> is the name of the input signal from which the path starts
- <Output> is the name of the output signal on which the path terminates

The values for <Delay>, <Input>, and <Output> can be found by examining the report file generated for the adder_16 design (reports/adder_16.rep), specifically you should examine the critical path report that was generated. It should be noted that <Input> is equivalent to Startpoint and <Output> is equivalent to Endpoint. The Delay value should be set to something at least 10% smaller than the data arrival time (full circuit delay) of the non-optimized pass. In addition to adding the above constraint to your script, you will need to add the following command in order to instruct Design Compiler® to restructure your design:

```
set_structure true -design <Design_Name> -boolean true -boolean_effort medium
```

Where <Design_Name> is the name of the design(s) which you wish to restructure.

The above command, set_structure, instructs Design Compiler® on how to approach the structure of a Verilog Design. It allows you to customize what type of structuring is used in the design. By default, Design Compiler® uses timing-driven structure. That is, it structures the designs so as to find optimal timing. However, the above command is changing the structuring method in order to use Boolean optimization.

A note about accessing the documentation for Design Compiler® should be stated right now. With these tools you have 2 ways of accessing documentation on the tools and the options. Inside DC Shell you can get help on any command by changing you shell to DC Shell by using the command dc_shell-t and issuing the following command at the dc_shell-t prompt:

man <command>

Where <Command> is the DC Shell function that you wish to receive help on. You can also issue the 'help' command in DC Shell to obtain a list of all available commands and options available in DC Shell. You can use 'man' in association with 'help' by typing 'help', finding a command you want to know more about, then issuing the 'man' command on that function or option in order to obtain a detailed description of the command.

You can also bring up the online documentation for Design Compiler® tools at a UNIX prompt by typing:

sold &

This will bring up an Adobe Acrobat Reader session that has all the documentation for the Design Compiler® tools. To exit the DC Shell and return to your native shell, just use the command quit.

At this time you may find it useful to bring up the man pages on the command 'compile' (in DC Shell) because for the second compilation pass in the makefile you will be required to change the mapping effort to HIGH and set the option to allow BOUNDARY OPTIMIZATION. Now you are ready to begin editing your makefile. As stated above you will need to add a timing constraint to the commands variable, add the command to allow for Boolean optimization and alter your compile statement for the second compile pass so that it uses a HIGH mapping effort and allows BOUNDARY OPTIMIZATIONS. When you finish modifying your makefile, its SYN_CMDS variable definition should look like Source Code 1.

At this point it should be stated that 'MOD_NAME' is a make variable that holds the name of the design it is currently synthesizing. You can obtain the value of 'MOD_NAME' by enclosing it in '$()'. Thus for the adder_16.sv design file, $(MOD_NAME) results in 'adder_16' being substituted in for the '$(MOD_NAME)' statement.

Once you have modified your makefile and re-synthesized the adder_16 design so that your second pass of the design produces a timing result that is improved relative to the first pass, have a TA check off your work up to this point.

Next, please answer the following questions on your Evaluation sheet:

Question: For the adder_16 with the modified makefile, what is the Critical Path Delay and Area of the circuit resulting from the first (unmodified) synthesis pass?

Question: For the adder_16 with the modified makefile, what is the Critical Path Delay and Area of the circuit resulting from the second (modified) synthesis pass?

IMPORTANT: Do not forget to check your work back into the GIT Repository.

```
Define SYN_CMDS
'# Step 1:  Read in the source file
analyze -format sverilog -lib WORK {$(DEP_SUB_FILES) $(MAIN_FILE)}
elaborate $(MOD_NAME)-lib WORK
uniqify

# Step 2: Set design constraints
# Uncomment below to set timing, area, power, etc. constraints
# set_max_delay <delay> -from "<input>" -to "<output>"
# set_max_area <area>
# set_max_total_power <power> mW
$(if $(and $(CLOCK_NAME), $(CLOCK_PERIOD)), create_clock "$(CLOCK_NAME)" -name
    "$(CLOCK_NAME)" -period $(CLOCK_PERIOD))

# Step 3: Compile the design
compile -map_effort medium

# Step 4: Output reports
report_timing -path full -delay max -max_paths 1 -nworst 1 > reports/$(MOD_NAME).rep
report_area >> reports/$(MOD_NAME).rep
report_power -hier >> reports/$(MOD_NAME).rep

# Step 5: Output final Verilog and Verilog files
write_file -format verilog -hierarchy -output "mapped/$(MOD_NAME).v"

# Second Compilation Run.  Repeat Steps 2-5
# Step 2:  Put the max delay constrains in the second pass only.
set_max_delay <delay> -from "<input>" -to "<output>"

# Step 3: Compile the design
set_structure true -design $(MOD_NAME) -boolean true -boolean_effort medium
compile <You Supply Options for Compile>

# Step 4: Output reports
report_timing -path full -delay max -max_paths 1 -nworst 1 >
    reports/$(MOD_NAME)_1.rep
report_area >> reports/$(MOD_NAME)_1.rep
report_power -hier >> reports/$(MOD_NAME)_1.rep

# Step 5: Output final Verilog and Verilog files
write_file -format verilog -hierarchy -output "mapped/$(MOD_NAME)_1.v"

echo "\nScript Done\n"
echo "\nChecking Design\n"
check_design
exit'
endef
```

Source Code 1: Example of Makefile SYN_CMDS definition modified to have a second (tuned) synthesis pass.

3 Efficient Non-Exhaustive Design Verification

This remainder of this lab is going to be focused on efficient approaches for verifying the functionality of designs. The previous designs you have been working with are pretty simple and more or less naturally easy to debug and verify due to both their simplicity and small input sizes. In order to explore approaches that enable efficient verification of both more complex designs and those that have significantly larger input sets, we will be scaling up your n-bit adder to be a 16-bit adder. Make a copy of your 8-bit adder wrapper file from Lab 2 and update it to be a 16-bit adder according to the following requirements.

Required Module Name: adder_16bit

Required Filename: adder_16bit.sv

Required Ports:

Port name	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

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, as there are 2^{33} different value permutations to run through. Additionally, exhaustive testing is unfeasible for most non-trivial designs and even some trivial designs that work with reasonably sized inputs, such as 32-bit logical operators (and, or, xor, etc.) with 2^{64} different input permutations. So what should be the approach to verifying the functionality of these circuits?

The key is to determine unique cases which cover specific subsets of the functionality with minimal overlap. This way maximum functionality can be tested using as few test cases as needed. Continue on to the next section once you have a complete 16-bit adder 'wrapper' file.

3.1 Test-Vector based Test Bench Design

In order to efficiently describe and execute the various focused test cases that share a pattern of work but have differing inputs or settings, it's best to use a 'test-vector' based approach. Test-vectors are simply groups of all key information that changes with each test case, and are usually described either via logically arranged arrays/lists or custom data types. Then an array of these test-vectors is created to facilitate efficient definition, population, and execution of each of the desired test vectors. A starter test bench based on this approach has been provided to you to help you get started. Source Code 2 illustrates the way test vectors are defined, using a custom datatype, within the provided starter 16-bit adder test bench. And Source Code 3 illustrates how clean and simple it is to define and populate the test vectors using the custom datatype approach.

```
// Declare custom test vector type
typedef struct{
  string                   test_name;
  logic [MAX_INPUT_BIT:0] test_a;
  logic [MAX_INPUT_BIT:0] test_b;
  logic                    test_cin;
} testVector;

// Declare the unpacked/dynamically sized test-vector array
testVector tb_test_cases [];
```

Source Code 2: Test-Vector and Test-Vector Array declaration from starter 16-bit Test Bench

```
// Initial block to cleanly define the contents of the test-vector array
initial begin
  // Create the test-vector array with enough slots for test cases
  // STUDENT TODO: Update the array declaration to have enough slots
  tb_test_cases = new[1];

  // First Test Case/Test-Vector
  tb_test_cases[0].test_name = "Zeros Check";
  tb_test_cases[0].test_a    = '0;
  tb_test_cases[0].test_b    = '0;
  tb_test_cases[0].test_cin  = 1'b0;

  // STUDENT TODO: Add your additional test cases here after increasing the array
    size
end
```

Source Code 3: Test-Vector Array Population example from starter 16-bit Test Bench

3.2 16-bit Adder Minimum Functional Testing Specifications

To help you get started with targeted test cases, below are a number of required test case scenarios.

- Your first test should be A = 0x0000, B = 0x0000 (already provided in starter TB)
- One of your test cases must have A be a large number and B be a small number
- One of your test cases must have B be a large number and A be a small number
- One of your test cases must have A be a large number and B be a large number
- One of your test cases must have A be a small number and B be a small number

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.

TOP_LEVEL_FILE Must contain the filename of your 16-bit adder design

COMPONENT_FILES Must contain the filename of your 1-bit and N-bit adder designs

Once you have simulated your design under these unique functional test cases, show your work to a TA to get checked off and move on to the next section.

3.3 Effective Simulation Waveform Formatting

In general well designed IDE tools enable you to have access to a lot of information about what is happening during the execution of your design, either via simulation-based data gathering or real-time debugging hooks. However with any non-trivial design there will always be far more information available to you than via these tools than you actually need to care about for the problem at hand and managing this plethora of information is critical to having a effective and efficient debugging session. This is especially true when working with HDL based designs, where the IDE's built in simulator will provide you with the full step-by-step history of values for every signal within the design. **If you do not properly manage and focus this information to what you are certain is relevant and organize it in a logical manner, then you will waste tremendous amounts of time both visually jumping around and confusing yourself about what is actually broken or working.**

An illustration of the importance of waveform organization and formatting can be seend by comparing the unformatted result from running 'add wave *' (Figure 1) and a well formatted selection of signals (Figure 2) for a simulation of the 16-bit adder test bench.

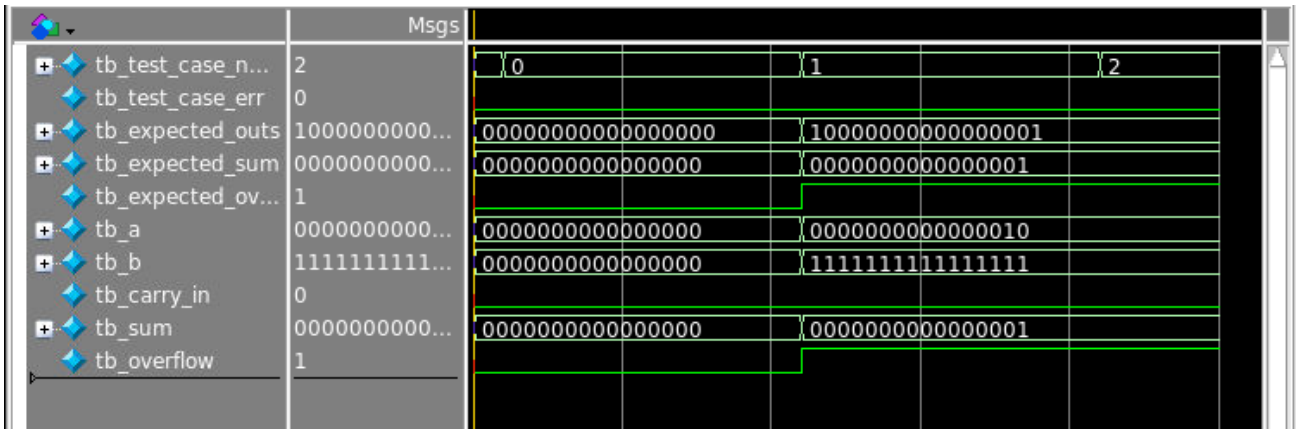

Figure 1: Non-formatted results from 'add wave *' for the 16-bit Adder Test Bench Source Simulation

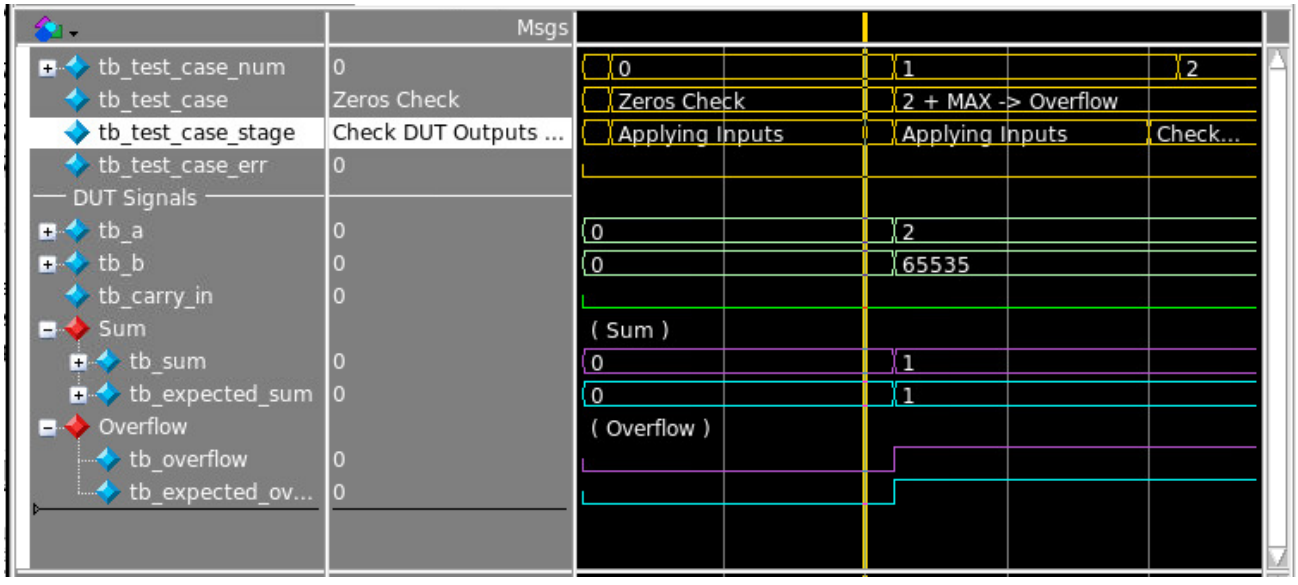

Figure 2: Formatted and Organized Waveforms for the 16-bit Adder Test Bench Source Simulation

In general the organizational flow that is recommended for a cleanly readable waveforms view is as follows:

- Keep high-level test bench signals such as the test case number, descriptions, and correctness error signals at the top (Select and add them from the 'Objects' view first)
- Add 'dividers' (via the right-click menu within the waveforms view) between major groups of signals
- After the high-level test case signals, add test bench signals that are connected to the DUT's ports
- Organize DUT connection signals with inputs first then resultant outputs, to simplify the effort for establishing cause and effect relationships when checking specific test cases
- Convert any signals that represent numerical values to be in a meaningful radix (generally unsigned is best)
- Add each DUT output port's expected result signal and then 'group' them with their respective DUT port connection signal (Remember be consistent with the relative order among all of the groups)
  1. Select the signals you which to group
  2. Right-click and select 'Group...'
- When you are not specifically interested in a signal 'group', collapse it away via the '-' button
- Use a consistent color-coding scheme to make it easier to find specific signals/signal types of interest (i.e., The golden color for the top 'navigation signals')

Once you have formatted your waveforms view in such a way, you can save the the results to a script that can be run by QuestaSim® to recreate that waveforms view for each fresh simulation run. With the waveforms view active (the top bar of the panel should be a bluish color) Select 'File'→'Save Format...'. Then specify the filename to save it as (the default is 'wave.do' in folder you ran QuestaSim® from) and click 'Ok'.

To run the script you can either type 'source <the script filename>' or select 'Load'→'Macro File...' from the 'File' drop-down menu from the QuestaSim® main menu bar or 'Load' from the 'File' drop-down menu if you have the waves view undocked from the QuestaSim® main window.

NOTE: These '.do' scripts are just a plain text script that stores the series of QuestaSim® commands need to do all of the signal adding and formatting, and thus can be directly edited and tweaked as well.

Once you have a '.do' script capable of setup a formatted waveforms view for the 16-bit adder test bench simulation, show it to a TA to get it checked off.

3.4 Testing/Verification with Internal Assertion Statements

Another useful feature for testing/debugging 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 (1) to check for valid inputs and (2) 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 does not 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 general syntax for the behavioral style assertion is as follows:

```
assert(<DUT's output> = <expected output>)
[Optional code for when assertion succeeds];
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.
- For the assertion's optional code section:
  - When using it for functionality checks, it is recommended to have '$info{"<some message>"}' to inform when good functionality happened.
  - When using it for extra debugging 'sanity' checks, it is recommended to not have any messaging code to avoid overwhelming your simulation's transcript.
- <Error message> is a user defined error Message. An Example is: "DUT's output does not match the expected output".

NOTE: 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 is in Source Code 4.

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

Source Code 4: Example of input checking for one of the inputs to a 1-bit adder

An example of functional component checking for the sum output of the first 1-bit adder is in Source Code 5

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

Source Code 5: Example of checking the 'sum' result for the first 1-bit adder

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.

4 Test Bench Code Coverage Analysis (Post-lab)

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 QuestaSim®

   make clean
   vsim -i

2. Compile → compile options(tab)
3. Make sure the options are selected as shown in Figure 3.
4. Close QuestaSim® and rerun the make simulation target. (i.e. make sim_full_source)
5. Run the simulation for as long as needed
6. Tools → Coverage Report → Text
   (a) Select "All instances" from the top drop down box
   (b) → Ok

Now QuestaSim® will display the report.

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

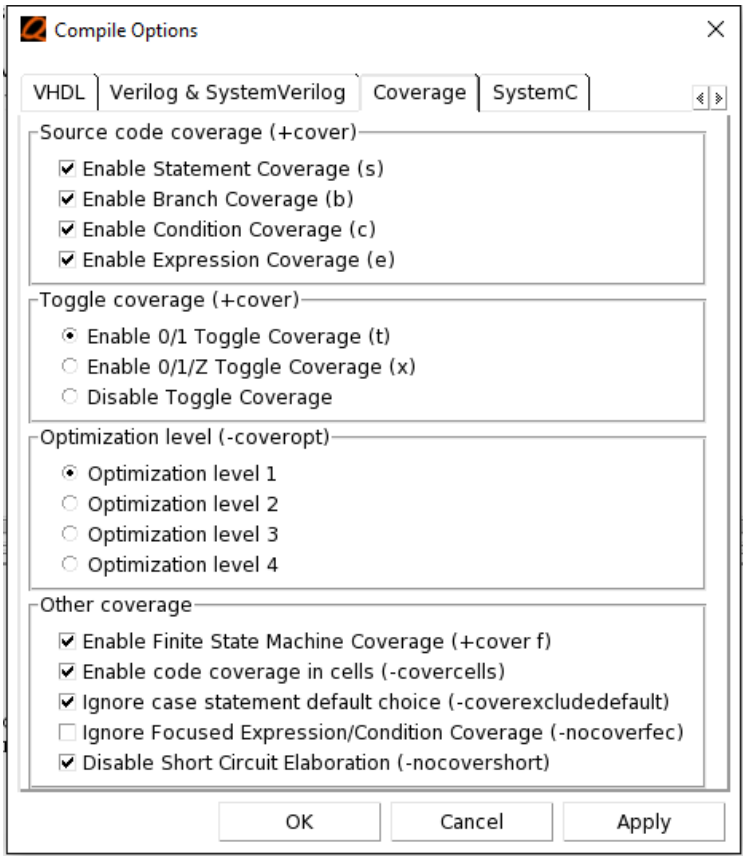

Figure 3: Coverage Compile Options

IMPORTANT: The report you just generated is a text based summary report and is only good for a quick view of how much is covered/missing. You can generate a much more thorough HTML (with default settings) report that will make it easier to figure out what is missing.

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 3.2, 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: Assertion coverage should be ignored, as they are not actual design code and are there to make debugging easier.

5 Closing Remarks

- Turn in your check-off sheet at the beginning of Lab Lab 4 .
- Since both inputs A and B are 16-Bits long (for the 16-bit adder), 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.