ASIC Design Laboratory
Lab 2 : Introduction to Verilog Coding Styles, Scalable Logic Design, and Exhaustive Testing
Lab Manual

Spring 2023

The purpose of this lab exercises is to help you become familiar with the different coding styles and their trade-offs, designing scalable and reusable HDL code modules, and design of exhaustive test bench code.

In this lab, you will perform the following tasks:

- Create and Test Verilog code for the Structural model of the Sensor Error Detector System using QuestaSim® (sensor_s.sv)
- Create and Test Verilog code for the Dataflow model of the Sensor Error Detector System using QuestaSim® (sensor_d.sv)
- Create and Test Verilog code for the Behavioral model of the Sensor Error Detector System using QuestaSim® (sensor_b.sv)
- Create and Test Verilog code for a 1-bit Full Adder using QuestaSim® (adder_1bit.sv)
- Create and Test Verilog code for a 4-bit Ripple Carry Adder using QuestaSim® (adder_4bit.sv)
- Create and Test Verilog code for a parameterized N-bit Ripple Carry using QuestaSim® (adder_nbit.sv)

1 Lab Setup

In a UNIX terminal window, issue the following commands, to setup your Lab 2 workspace:

mkdir –p ~/ece337/Lab2
cd ~/ece337/Lab2
dirset
setup2

The setup2 command is an alias to a script file that will check your Lab 2 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 Sensor Error Detector Implementations

2.1 Structural Style Sensor Error Detector Design

2.1.1 Structural Style Sensor Error Detector Specifications

Required Module Name: sensor_s

Required Filename: sensor_s.sv

Required Ports:

Port name	Direction	Description
sensors[3:0]	input	Inputs from sensors
error	output	Detector Result

Create the source file using the following 'ch' script command from your Lab2 folder:

ch sensor_s.sv

2.1.2 Structural Coding of Sensor Error Detector

A structural model for a Verilog model is much like what would be consider a pure netlist description of the cell. A netlist is essentially a text representation that describes a circuit in terms of the explicit interconnections between sub-blocks. This explicit description describes what signals are input to a sub-block and what signals are outputs from this sub-block and how they interact with the other sub-blocks in the design. This is the style that is probably the easiest to understand and create a design in. This is the case because it is relatively easy to map a K-Map derived expression for a logic function into this style of Verilog. Therefore, this is the Verilog style that will seem the most logical to most students; however, this style is not the most powerful of the design styles for Verilog. The structural model is one that lends itself to directly illustrating a hierarchical design in Verilog. A hierarchical design is one in which you use several smaller designs to create a larger design. An example of a hierarchical design would be a 16-bit adder that is built from a combination of 1-bit adders. In this design, you would have 16 instantiations of the 1-bit adder design that are interconnected in order to perform the function of a 16-bit adder.

The hierarchical design methodology is what is employed in industry. This should make sense to you simply from a design point of view. The majority of the designs being done in industry are at such a degree of complexity that no one person on the design team knows every detail of the overall system. Instead, the overall system is divided into units which are divided into blocks, which are divided into sub-blocks. These sub-blocks are relatively easy to manage aspects of the overall project that one engineer or a small group can be responsible for. Even these sub-blocks have a hierarchical aspect to them. For instance, one portion of the design may be highly optimized because it is the most critical portion of the sub-block. This optimized portion of the sub-block is then encapsulated in a symbol and placed into schematic at a higher level in the sub-blocks internal hierarchy.

This hierarchical design methodology has an additional benefit in terms of test-ability of a design also. Logically, small blocks are able to be more thoroughly tested than larger blocks. This statement is derived from the fact that smaller blocks generally have fewer inputs, so it is possible to run a small block through its entire range of input values to make sure it is functioning correctly. However, as one moves up the levels in the design hierarchy the ability to thoroughly test a design becomes more difficult. In the case of microprocessors, thoroughly and completely simulating a design is simply not a feasible option. In order to test a microprocessor all its possible input vectors would require an enormous amount of compute cycles, not to mention the incredible amount of engineering hours it would require to generate the test vectors, determine their expected response(s), and ensure their correctness. From this, one should see that it is imperative that the lower levels in the hierarchy be tested thoroughly to ensure that they function properly. Ensuring that the lowest levels in the hierarchy function correctly allows the top-level design testing process to be an attainable goal, as opposed to an insurmountable goal.

In lab 1's post lab, you derived a logic expression from a K-Map for the Sensor Error Detector circuit. This logical expression that you derived was in the Sum of Products form. You are now going to implement this logic equation using the structural Verilog coding style. Table 1 contains the list of 2-input logic cells (and their ports) that you might need from the standard cell library.

Table 1: Table of available basic logic cells in provided standard cell library

Logic Cell Function	Logic Cell Module Declaration
Inverter	INVX1(input A, output Y)
2-Input AND Gate	AND2X1(input A, input B, output Y)
2-Input NAND Gate	NAND2X1(input A, input B, output Y)
2-Input NOR Gate	NOR2X1(input A, input B, output Y)
2-Input OR Gate	OR2X1(input A, input B, output Y)
2-Input XOR Gate	XOR2X1(input A, input B, output Y)
2-Input XNOR Gate	XNOR2X1(input A, input B, output Y)

For example, the following line of code creates an instance of a 2-input AND gate with a label of 'A1', with signals 'a' and 'b' connected to its inputs, and signal 'int_and1' connected to its output.

```
AND2X1 A1 (.Y(int_and1), .A(a), .B(b));
```

Utilizing this information, the specifications in section Section 2.1.1, your sum-of-products equation from lab 1's postlab, and any Verilog code syntax references, create the structural style sensor detector code in your lab 2 source folder.

2.1.3 Testing of the structural style Sensor Error Detector

Part of what the setup2 script did was give you a copy of a Verilog code file (tb_sensor_s.sv), which is what we call a test bench file. This code file is a module that creates an instance of the design file it is used to test and controls the values of this instance's inputs in order to force the design being tested through a variety of test cases that were implemented with the test bench module. This is the way all designs for the rest of the course will be tested, as it is much more powerful and more efficient than using force statements like you did in lab 1. To simplify the usage of test benches for testing designs, the makefile provided by dirset also has simulation targets for simulating test benches of single file designs. To simulate the provided test bench for the structural sensor detector module, execute the following command from your lab 2 folder.

make tbsim_sensor_s_source

This make target compiles both the test bench file tb_sensor_s.sv and the design file sensor_s.sv if needed and then starts a simulation of the tb_sensor_s test bench module. Once the simulation has loaded, add the design's port signals and the signal named 'test_number' to the waves window and then tell QuestaSim® to run for 200 ns.

At this point have your TA verify your Waveforms window.

Now check the design's output for correctness for each test case ('test_number' should always increment when a new test case starts). If an incorrect output is found, make corrections to your design's code, recompile the design (this can be done easily from within QuestaSim® by right clicking on the design instance in the "source_work" library and selecting recompile), and restart the simulation ("restart –f"), and rerun the simulation. After a fully correct source simulation, synthesize the design and simulate the mapped design with the same provided test bench to check for any errors during design synthesis. The command for simulating the mapped design with its test bench is:

make tbsim_sensor_s_mapped

Once you have a fully working design proceed to the next section.

2.1.4 Automated Grading of the Structural Sensor Error Detector

In this class all design code will be graded via a set of grading scripts and custom grading test benches that are run during via submission commands. To submit your structural sensor detector design for grading, issue the following command at the terminal (can be from anywhere).

submit Lab2s

2.2 Dataflow Style Sensor Error Detector Design

2.2.1 Dataflow Style Sensor Error Detector Specifications

Required Module Name: sensor_d

Required Filename: sensor_d.sv

Required Ports:

Port name	Direction	Description
sensors[3:0]	input	Inputs from sensors
error	output	Detector Result

Create the source file using the following 'ch' script command from your Lab2 folder:

ch sensor_d.sv

2.2.2 Dataflow Coding of Sensor Error Detector

Utilizing the dataflow syntax examples from the lab notes and other Verilog references, create a dataflow style design according to the requirements in Section 2.2.1. A purely dataflow style design cannot have any procedural blocks and all value assignments must be done with the 'assign' syntax. Additionally the setup2 script has provided you with a test bench for the dataflow style sensor detector (tb_sensor_d.sv). Make sure that the design is fully working before proceeding to the next section and submitting it for grading. Based on makefile's pattern rules for simulation, the make targets for simulating the dataflow source and mapped versions respectively are:

make tbsim_sensor_d_source

and

make tbsim_sensor_d_mapped

2.2.3 Automated Grading of the Dataflow Sensor Error Detector

To submit your dataflow sensor detector design for grading, issue the following command at the terminal (can be from anywhere).

submit Lab2d

2.3 Behavioral Style Sensor Error Detector Design

2.3.1 Behavioral Style Sensor Error Detector Specifications

Required Module Name: sensor_b

Required Filename: sensor_b.sv

Required Ports:

Port name	Direction	Description
sensors[3:0]	input	Inputs from sensors
error	output	Detector Result

Create the source file using the following 'ch' script command from your Lab2 folder:

ch sensor_b.sv

2.3.2 Behavioral Coding of Sensor Error Detector

Utilizing the behavioral syntax examples from the lab notes and other Verilog references, create a behavioral style design according to the requirements in Section 2.3.1. Remember that a purely behavioral style design cannot have any functional/logic code outside of the procedural blocks. The combinational logic should be handled inside an 'always_comb' block (or an 'always' block with each of its input signals in the sensitivity list). Also, for this class initial blocks are forbidden inside design modules, and are only allowed to be used in test benches. Additionally the setup2 script has provided you with a test bench module for the behavioral style sensor detector as well (tb_sensor_b.sv). Make sure that the design is fully working in its mapped/synthesized form before proceeding to the next section and submitting it for grading. Also, as a reminder of the use of the makefile's pattern rules for simulation, the make targets for simulating the behavioral source and mapped versions respectively are:

make tbsim_sensor_b_source

and

make tbsim_sensor_b_mapped

2.3.3 Automated Grading of the Behavioral Sensor Error Detector

To submit your structural sensor detector design for grading issue the following command at the terminal (can be from anywhere).

submit Lab2b

3 Viewing Design Schematics for Synthesized Designs

In this section you will be viewing schematic representations of the gate net lists synthesized from your 3 sensor detector implementations.

3.1 Viewing the Structural Style Schematic

In your terminal, in your Lab 2 directory, bring up the Design Compiler GUI (yes, our synthesis tool has a GUI, called Design Vision, but we won't be using it much) by typing:

dv

In the window that comes up, select:

File → Read

Open the file "mapped/sensor_s.v" and select OK.

At the very right of the toolbar below the menu is a box where can choose the current design. Make sure that the top-level module name is selected (sensor_s). Now go to the menu and select

Schematic → New Design Schematic View

If you zoom in (View → Zoom) you can see the component types and names, as well as signal names. If your design has sub-components (though this design probably won't), you can see their schematics by selecting them with the LMB and selecting:

Schematic → Move Down.

You can return to the top level by selecting:

Schematic → Move Up.

Once you have generated a schematic view using Design Vision, have a TA check off your work up to this point.

3.2 Viewing the Dataflow Style Schematic

Analyze the schematic of your mapped dataflow implementation (mapped/sensor_d.v) with Design Vision, as before.

Once you have generated the schematic in Design Vision, have a TA check off your work up to this point.

3.3 Viewing the Behavioral Style Schematic

Use Design Vision to examine the schematic for you mapped behavioral implementation (mapped/sensor_b.v), remembering that you can view internal blocks in the design hierarchy by moving up or down in the schematic.

Once you have a clean schematic (no extraneous wires), have a TA check off your work up to this point.

Also, make sure to commit the versions of your sensor code in Git.

git add Source/sensor*.sv
git commit -m "<Add your own commit message text here.>"

3.4 Follow Up Question

Answer the following question on your evaluation sheet.

Which Style of Verilog Code: DATAFLOW, STRUCTURAL, or BEHAVIORAL is the easiest to modify if the number of bits in the input data bus were altered? Why?

4 Scalable Logic Design and Testing (Post-lab)

4.1 1-bit Full Adder Design

4.1.1 Specifications

Design (code and verify) a 1-bit Full Adder module with the following specifications:

Required Module Name: adder_1bit

Required Filename: adder_1bit.sv

Required Ports:

Port name	Direction	Description
a	input	One of the two primary inputs
b	input	Second of the two primary inputs
carry_in	input	The overflow value carried in from a prior addition column
sum	output	The computed sum value
carry_out	output	The overflow value sent to the next addition column

4.1.2 Implementation

In case you don't remember, the equations for calculating the sum and carryout values are below:

```
s = c_in xor (a xor b)
c_out = ((not c_in) and b and a) or (c_in and (b or a))
```

4.1.3 Grading

To submit your working 1-bit Adder for grading use the 'submit Lab2adder1' command.

4.2 4-bit Full Adder Design

The next step is to create the 4-Bit Ripple Carry Adder. Figure 1 illustrates how a 3-Bit ripple carry adder can be constructed from three 1-Bit full adders.

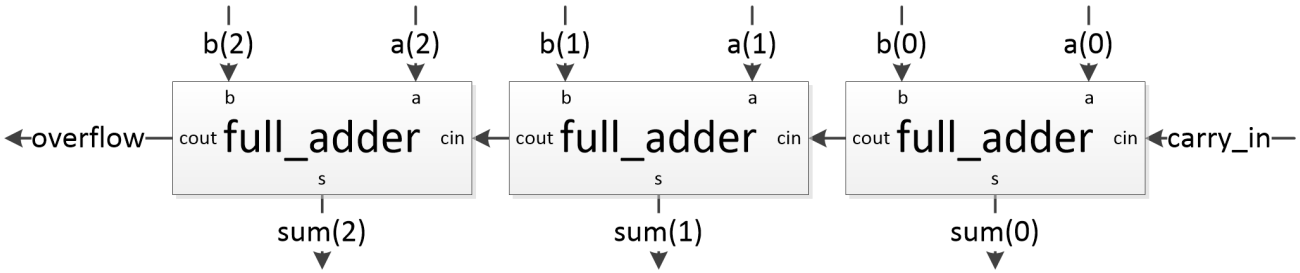

Figure 1: 3-bit Ripple Carry Adder Diagram

4.2.1 Specifications

Required Module Name: adder_4bit

Required Filename: adder_4bit.sv

Required Ports:

Port name	Direction	Description
a[3:0]	input	One of the two primary inputs
b[3:0]	input	Second of the two primary inputs
carry_in	input	The overflow value carried in from a prior addition column
sum[3:0]	output	The computed sum value
overflow	output	The overflow value sent to the next addition column

4.2.2 Implementation

We would like you to use the structural style of coding to create the 4-Bit adder from your 1-bit Full Adder design you made. At simulation time, QuestaSim® 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 specifying the additional libraries for mapped version gates and provided course IP modules. Although it would still be a good idea to investigate how the makefile does that for you. 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, 4 bits in size, to illustrate our point). After you declare the intermediate signal(s) you will need, you can start connecting the components. Source Code 1 is an example of how you would port map the first three 1-bit adder instances:

```
1 adder_1bit ADD0 (.a(a[0], .b(b[0], .carry_in(carry_in), .sum(sum[0]),
      .carry_out(carrys[0]));
2 adder_1bit ADD1 (.a(a[1], .b(b[1], .carry_in(carrys[0]), .sum(sum[1]),
      .carry_out(carrys[1]));
3 adder_1bit ADD2 (.a(a[2], .b(b[2], .carry_in(carrys[1]), .sum(sum[2]),
      .carry_out(carrys[2]));
```

Source Code 1: Example code for 3 stages of Ripple Carry Adder via manual port-mapping

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. Source Code 2 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 [4:0] carrys;
genvar i;

assign carrys[0] = carry_in;
generate
for(i = 0; i <= 3; i = i + 1)
  begin
    adder_1bit IX (.a(a[i]), .b(b[i]), .carry_in(carrys[i]), .sum(sum[i]),
    .carry_out(carrys[i+1]));
  end
endgenerate
assign overflow = carrys[4];
```

Source Code 2: Example code for 4 stages of Ripple Carry Adder via generate-based port-mapping

After you have finished creating your generate based 4-bit adder version, extend the exhaustive test bench provided for the 1-bit adder to exhaustively test your 4-bit adder for verifying your design. The simulation commands you have been using previously will only work for designs with only one source file, and one test bench involved.

In order to simulate multi-file designs, referred to as hierarchical designs, you will need to use the 'full' commands (sim_full_source and sim_full_mapped) instead and populate the following variables in the makefile:

TOP_LEVEL_FILE: Must contain the filename of your 4-bit adder design (not including the source folder)

COMPONENT_FILES: Must contain the filename of your 1-bit and design (not including the source folder)

4.2.3 Grading

To submit your working 4-bit adder and exhaustive test bench for grading use the 'submit Lab2adder4'command.

4.3 Creating a Scalable Ripple Carry Adder Design with Parameters

4.3.1 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 <name>
#(
  <parameter declaration>,
  ...
```

```
  <parameter declaration>
)
(
  <port declaration>,
  ...
  <port declaration>
);
```

When creating module definitions for a parameterized design it often is necessary to have one or more port(s) scale based on a parameter. Doing this is as simple as directly using the parameter to determine the port dimensions. Below is a simple example:

```
module example_scalable_design
#(
  parameter NUM_BITS = 4
)
(
  input wire [(NUM_BITS - 1):0] operand_a,
  ...
);
```

4.3.2 Implementation

Using the above discussion of parameters and your generate syntax based 4-bit Ripple Carry Adder, create a parameterized Ripple Carry Adder with a parameter called 'BIT_WIDTH' that determines the number of bit pairs added and the size of the 'a' and 'b' ports. The default value for this parameter must be 4.

Required Module Name: adder_nbit

Required Filename: adder_nbit.sv

Required Ports:

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

NOTE: The actual port declaration should use the BIT_WIDTH parameter value to determine the value of the '#' in the port definitions.

4.3.3 Synthesis and Verification

Only source versions of designs can be scaled or modified by parameters. Therefore in order to test the mapped functionality of the scaled version (where something other than the default value is used) you will need to use a wrapper file that includes your scalable design and then overrides the parameter size locally. Then this wrapper file must be synthesized together with the flexible design's source code in order to create the full scaled size mapped design file that can be compiled and tested. A template 8-bit adder file (adder_8bit.sv) has been provided to you via the setup2 script to aid you in this task. The module declaration has been defined for you in, but you must insert the proper port map code for using your scalable ripple carry adder.

When verifying your parameterized version for the unscaled (default value sized) form, you should be able to use a only slightly modified copy of your 4-bit test bench. Since it's default value should make it function identically to a 4-bit adder (both in source and mapped forms), you will only need to update the design port map (to use the new modules name) after copying your 4-bit adder's test bench. For testing the scaled form, you will need to update a copy of your 4-bit adder test bench to work with the 8-bit wrapper file directly.

Be sure to update the make variables used for hierarchical designs to match your current system:

- When testing your N-bit adder directly using it's default values:
  TOP_LEVEL_FILE: Must contain the filename of your n-bit adder design
  COMPONENT_FILES: Must contain the filename of your 1-bit adder design
- When testing your the scaled version of your N-bit adder using the 8-bit wrapper file:
  TOP_LEVEL_FILE: Must contain the filename of your 8-bit wrapper file
  COMPONENT_FILES: Must contain the filename of your 1-bit and n-bit adder designs

NOTE: Remember source filenames/filepaths specified in the makefile variable definitions should never include the source folder, as the makefile already assumes all source files are in the 'source' folder.

4.3.4 Grading

To submit your working scalable Ripple Carry Adder, 8-bit adder wrapper file, and exhaustive 8-bit adder test bench for grading use the 'submit Lab2addern'command.

5 Closing Remarks

- Turn in your check-off sheet at the beginning of Lab Lab 3 .
- Late submission commands: Lab2a1l, Lab2a4l, Lab2anl, Lab2bl, Lab2dl, Lab2sl
- Remediation output commands: Lab2a1ro, Lab2a4ro, Lab2anro, Lab2bro, Lab2dro, Lab2sro