ECE337 Lab 2: Introduction to Various Styles of Verilog Source Code and Design Test Benches

In this lab, you will:

- Create and Test Verilog code for the STRUCTURAL model of the Sensor Error Detector System using ModelSim (sensor_s.sv)
- Create and Test Verilog code for the DATAFLOW model of the Sensor Error Detector System using ModelSim (sensor_d.sv)
- Create and Test Verilog code for the BEHVIORAL model of the Sensor Error Detector System using ModelSim (sensor_b.sv)
- Modify the Makefile to optimize the 3 versions of the design that you are examining
- Create and Test Verilog code for a 1-bit Full Adder using Modelsim (full_adder.sv)
- Create and Test Verilog code for a 4-bit Serial-to-Parallel Shift Register using Modelsim (stp_sr.sv)
- Create and Test Verilog code for a 4-bit Parallel-to-Serial Shift Register using Modelsim (pts_sr.sv)
- Create and Test Verilog code for a Synchronizer Modelsim (sync.sv)

NOTE: A good habit to develop is one where you name the Verilog source code file the same name as the module name, plus the ".sv" extension. For example, if the module name is "sensor_d", then the file name should be "sensor_d.sv".

1. Lab Exercises

1.1. 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 setup your Lab 2 directory structure. **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 Lab 1. This way, you will always have the original copy in storage. Remember to delete and then check out the Lab2 folder after you import it, so you have a working copy.

1.2. Sensor Error Detector Design

1.2.1. Structural Style Sensor Error Detector Design

1.2.1.1. Structural Style Sensor Error Detector Specifications

The required module name is: sensor_s

The required filename is: sensor_s.sv

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

```
input wire [3:0] sensors
output wire error
```

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

```
ch sensor_s.sv
```

1.2.1.2. 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 hierarchal design in Verilog. A hierarchal design is one in which you use several smaller designs to create a larger design. An example of a hierarchal 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 hierarchal 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 hierarchal 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 hierarchal design methodology has an additional benefit in terms of testability 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 engineer hours it would require to generate the test vectors and ensure that the person designing the test vectors also determines the correct response that should be generated from each test vector. 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. Verilog already has basic gates and logic modules built into the language, using the keywords 'and', 'not', 'or', 'nor', 'nand', 'xor', and 'xnor'. These 'primitives' are predefined flexible module like functions that implement their corresponding Boolean logic operation and scale with the number of input bits you want to feed them. However, since they are not full modules they cannot be port-mapped using the fully explicit style where you say what port you connecting to specifically and have to be uses similar to the normally not recommended implicit port-mapping style. 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.

```
and A1 (int_and1, a, b);
```

Utilizing this information, the specifications in section 1.2.1.1, 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.

1.2.1.3. 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 designs port signals and the signal named 'test_case' to the waves window and then tell ModelSim to run for 1000 ns. **At this point have your TA verify your Waveforms window.**

Now check the design's output for correctness for each test case ('test_case' 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, 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.

1.2.1.4. 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 test benches that are run during corresponding submission commands. To submit your structural sensor detector design for grading issue the following command at the terminal (can be from anywhere).

```
Submit Lab2s
```

1.2.2. Dataflow Style Sensor Error Detector Design

1.2.2.1. Dataflow Style Sensor Error Detector Specifications

The required module name is: sensor_d

The required filename is: sensor_d.sv

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

```
input wire [3:0] sensors
output wire error
```

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

```
ch sensor_d.sv
```

1.2.2.2. Dataflow Coding of Sensor Error Detector

Utilizing the dataflow syntax examples from the lab 1 manual, lab notes, and other Verilog references, create a dataflow style design according to the requirements in section 1.2.2.1. Remember that 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 module for the dataflow style sensor detector as well (tb_sensor_d.sv). Make sure that the design is fully working 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 dataflow source and mapped versions respectively are

make tbsim_sensor_d_source

and

make tbsim_sensor_d_mapped

1.2.2.3. Automated Grading of the Dataflow Sensor Error Detector

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

Submit Lab2d

1.2.3. Behavioral Style Sensor Error Detector Design

1.2.3.1. Behavioral Style Sensor Error Detector Specifications

The required module name is: sensor_b

The required filename is: sensor_b.sv

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

```
input wire [3:0] sensors
output reg error
```

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

ch sensor_b.sv

1.2.3.2. Behavioral Coding of Sensor Error Detector

Utilizing the dataflow syntax examples from the lab 1 manual, lab notes, and other Verilog references, create a behavioral style design according to the requirements in section1.2.3.1. Remember that a purely behavioral style design cannot have any functional/logic code outside of the procedural blocks, and the combinational logic should be handled inside 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 dataflow 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

1.2.3.3. 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

1.3. Design Schematics for Synthesized Design Code

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

1.3.1. 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

```
grid 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 entity 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.

1.3.2. 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 an error-free run of Synopsys and generated the schematic in Design Vision, have a TA check off your work up to this point.

1.3.3. 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. Be sure to zoom in far enough to read bus and component names. If you see a bus named something like "*Logic0*^29,X,Y,Z" it means that the high 29 bits of the bus are held at logic zero (i.e., connected to ground), and the low 3 bits of the bus are connected to signals X, Y, and Z, respectively. In order to define the size (in bits) of an integer variable, to prevent this from occurring, declare it as follows:

`reg [<max_bit>:0] <integer variable name>;`

where:

- *<max_bit> is the number of bits needed - 1.*
- *<integer variable name> is the name of the variable*

For example, suppose you have a Verilog design that uses the following variable declaration:

integer int_var;

However, you now want to limit the range of values that int_var can assume to be in the range of 0 to 63. In order to do this you would change the above single statement into the following statement:

`reg [5:0] int_var;`

It may seem strange converting the declaration from type "integer" to type "reg". However, the "integer" datatype is not intended to be used for anything that will manifest in hardware but rather for testing and simulation purposes. Because of this it is required to have a minimum of 32 bits, but can vary somewhat in the maximum depending on the simulator's implementation. In addition all variables (nets denoted by type "reg") can be worked with directly as if they are integers, real numbers (double precision floats), binary values, hexadecimal values, etc.

If you schematic had floating/grounded wires like mentioned above, go in and correct you signal declaration to correct that, resynthesize it, and pull up its new 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 update the versions of your sensor code in svn using the checkin ('ci') command.

1.3.4. Design Synthesis Optimization

If the resulting behavioral design is still not as compact/optimal as the design that resulted from the dataflow or structural styles of Verilog coding this should be a cause of concern for you, since all the styles perform the same logical function. You may be wondering why they do not "map" to the same design? This is a good time to bring up the fact that your resulting design from Synopsys depends upon how the tool chooses to approach the problem and how the designer has written the Verilog code. With behavioral style Verilog, Synopsys does not always yield the most optimal design after the first compilation pass through the tool. Thus, for some designs multiple compilations are required in order to generate the most optimum design. In most cases, the designer, you, alters optimization constraints between successive compiles of the design in an attempt to generate the most optimum design.

Therefore, you are going to alter your "SYN_CMDS" variable in the makefile so that it will cause Synopsys to perform 2 compilation passes**. In modifying your makefile to accommodate a second compilation pass, you will add a timing constraint to the compile options and**

instruct Synopsys 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 sensor_s design (reports/ sensor_s.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 so that your BEHAVIORAL style design has approximately the same critical path delay as your STRUCTURAL style design.

In addition to adding the above constraint to your script, you will need to add lines similar to the following two lines in order to instruct Synopsys 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 first command, set_structure, instructs Synopsys 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, DC Shell 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 Synopsys 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 Synopsys tools at a UNIX prompt by typing:

sold &

This will bring up an Adobe Acrobat Reader session that has all the documentation for the Synopsys 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, it's SYN_CMDS variable definition should look like the following:

```
Define SYN_CMDS
'# Step 1:  Read in the source file                                    \n\
analyze -format sverilog -lib WORK {$(DEP_SUB_FILES) $(MAIN_FILE)}    \n\
elaborate sensor_b -lib WORK -update                                  \n\
                                                                      \n\
# Step 2: Set design constraints                                      \n\
# Uncomment below to set timing, area, power, etc. constraints       \n\
set_max_delay <delay> -from "<input>" -to "<output>"                  \n\
# set_max_area <area>                                                 \n\
# set_max_total_power <power> mW                                      \n\
                                                                      \n\
# Step 3: Compile the design                                          \n\
compile -map_effort medium                                            \n\
                                                                      \n\
# Step 4: Output reports                                              \n\
report_timing -path full -delay max -max_paths 1 -nworst 1 >
reports/$(MOD_NAME).rep                                               \n\
report_area >> reports/$(MOD_NAME).rep                                \n\
report_power -hier >> reports/$(MOD_NAME).rep                         \n\
                                                                      \n\
# Step 5: Output final Verilog and Verilog files                      \n\
write -format verilog -hierarchy -output "mapped/$(MOD_NAME).v"       \n\
                                                                      \n\
# Second Compilation Run.  Repeat Steps 2-5                           \n\
# Step 2:  No new constraints to set for this run.                    \n\
# Step 3: Compile the design                                          \n\
set_structure true -design $(MOD_NAME) -boolean true -boolean_effort
medium                                                                \n\
compile <You Supply Options for Compile>                              \n\
                                                                      \n\
# Step 4: Output reports                                              \n\
report_timing -path full -delay max -max_paths 1 -nworst 1 >
reports/$(MOD_NAME)_1.rep                                             \n\
report_area >> reports/$(MOD_NAME)_1.rep                              \n\
```

```
report_power -hier >> reports/$(MOD_NAME)_1.rep                    \n\
                                                                   \n\
# Step 5: Output final Verilog and Verilog files                   \n\
write -format verilog -hierarchy -output "mapped/$(MOD_NAME)_1.v"  \n\
                                                                   \n\
echo "\nScript Done\n"                                             \n\
echo "\nChecking Design\n"                                         \n\
check_design                                                       \n\
exit'
endef
```

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 this design, $(MOD_NAME) results in 'sensor_b' being substituted in for the '$(MOD_NAME)' statement. Note that the following three definitions are different:

Once you have modified your makefile so that your BEHAVIORAL Style Verilog code produces a timing result approximately equal to the timing for the sensor_s design and has a structure like the DATAFLOW and STRUCTURAL styles, **have a TA check off your work up to this point**.

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

For the BEHAVIORAL style Verilog with the modified makefile, what is the Critical Path Delay and Area of the circuit resulting from the first compilation pass?

For the BEHAVIORAL style Verilog with the modified makefile, what is the Critical Path Delay and Area of the circuit resulting from the second compilation pass?

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?

Do not forget to check your work back into the SVN Repository.

2. Postlab Exercises (Building Blocks Design)

2.1. Deliverables/Grading

Submit your completed building block designs (all together) via the following command.

```
submit Lab2P
```

Note: The code files must be in your source folder. They must also compile without errors and be functionally correct.

2.2. 1-bit Full Adder Design

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

The required module name is: full_adder

The required filename is: full_adder.sv

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

```
input wire a
input wire b
input wire c_in
output wire s
output wire c_out
```

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

```
s    = cin xor (a xor b)
cout = ((not cin) and b and a) or (cin and (b or a))
```

2.3. 4-bit Serial-to-Parallel Shift Register Design

Design (code and verify) the 4-bit serial-to-parallel shift register module you diagramed in lab 1's postlab with the following specifications:

The required module name is: stp_sr

The required filename is: 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.
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[3:0]	Output	This is the 4-bit data that is currently held by the shift register.

2.4. 4-bit Parallel-to-Serial Shift Register Design

Design (code and verify) the 4-bit parallel-to-serial shift register module you diagramed in lab 1's postlab with the following specifications:

The required module name is: pts_sr

The required filename is: 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.
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.
parallel_in[3:0]	Input	This is the 4-bit data that will be loaded into the shift register when the load_enable signal is active.
serial_out	Output	This is the serial input signal and thus will contain the data that is being serially transmitted to the UART. If a packet is not being transmitted, this line will be a logic '1'.

2.5. Synchronizer Design

Design (code and verify) the synchronizer module you diagramed in lab 1's postlab with the following specifications:

The required module name is: sync

The required filename is: sync.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.
async_in	Input	This is the asynchronous input port (the original signal which is not synchronized to the supplied clock signal).
sync_out	Input	This is the synchronous output port (the form of the input that is now synchronized with the supplied clock signal).