ASIC Design Lab 1: Introduction to Verilog Simulation via QuestaSim, Gate Net-list Synthesis, and Digital Logic Design Refresher

The purpose of this first lab exercise is to help you become familiar with the Verilog synthesizer, Design Compiler (by Synopsys), and the Verilog simulator, QuestaSim (by Mentor Graphics), that you will be employing throughout the class.

In this lab, you will perform the following tasks:

- Create a text file containing the Verilog source code for a 16-bit Comparator that determines whether A is greater than, less than, or equal to B.
- Compile the source code and correct any syntax errors.
- Simulate the Verilog model via QuestaSim.
- Synthesize the Verilog model utilizing Synopsys' Design Compiler.
- Generate a Design Compiler Script in order to perform several synthesis runs on the Comparator. This will optimize the critical path and the area usage of the Comparator design.
- Solve basic digital logic design problems.
- Create Schematic and Register Transfer Level (RTL) block diagrams for basic ASIC building blocks.

1. Required Background Knowledge and Introduction to Lab Format

Throughout this course, it is assumed that you have rudimentary programming skills that are taught in a Freshman or Sophomore level structured/logical coding class. Also, a detailed understanding of digital logic design, both sequential and combinational, is required in order to complete the laboratory exercises and a necessity when work begins on your respective semester project as you will be creating a design of your own. **It should be noted that while we will be teaching you some Verilog syntax because it is needed in order to describe the more complex designs implemented in this course, the syntax itself is not the primary learning objective of this course and so there will be times where you are expected to familiarize yourself with the simpler syntax from provided references.** The digital logic design requirement includes, but is not limited to, Karnaugh Maps (K-Maps), Truth Tables, and State Machine design, both Moore and Mealy machines.

This course will also serve as an introduction to writing scripts that are utilized by CAD tools in order to automate the process of using the tool. This automation will be in the form of having the scripts issue all the commands to the CAD tool as opposed to you entering the commands via the menu options or repeated command line calls. Through this exposure, it is hoped that you will find scripts to be a very efficient and time-saving mechanism in which to interact with the variety of CAD tools that they must utilize in this course.

During the course of this semester you will be conducting 7 laboratory experiments on the workstations in the VLSI Design lab. The laboratory experiments will be available for you to download in PDF format from **Blackboard**. You should read through the lab before coming to class. This will help you have an idea of the goal of the lab and the steps that will be necessary for its completion. A few formatting specifications about the lab manuals is on the following page. This list illustrates a type of text on the left with the corresponding meaning on the right.

`Command Style`	This text style indicates a command that should be typed into a UNIX command

	prompt or placed into a text file for Scripts.
`Code Text`	This text style indicates a Verilog or Verilog code section for a source file, or code comments.
Question	This indicates a question being posed to you that must have its answer recorded on the evaluation sheet in order for the TA to grade the response
File → Open	This form is used to indicate how a student should navigate through a menu selection in order to get to the desired option.
🛑	This Stop Sign is used to indicate a point where you should get your TA to check off your work up to this point. This is done to ensure that you are doing the lab properly, and to minimize the amount of corrections that may need to be done.
LMB	This stands for Left Mouse Button and is used to indicate when you should use the button to select or highlight an option in a menu or on a design.
RMB	This stands for Right Mouse Button and is used to indicate when you should use the button to bring up a context sensitive menu in the current design tool.

2. Getting Started and Setting Up Your Account

To begin this laboratory, you must login to a workstation. In order to do this, enter your mg account and password in the login window which should appear if you move the mouse. If you are unable to log into the machines in the EE 215 VLSI design lab please let your TA know immediately.

Once you are logged into the machine, the first step that needs to be performed is to setup your account so that you may access the variety of tools that will be required for this class throughout the remainder of the semester. Bring up a terminal window by holding down the right mouse button, **RMB**, and select **"Open Terminal"**. This should bring up a new window containing a command prompt.

The first thing you should do is change your password. In your terminal window, issue the following command:

```
passwd
```

2.1. Account Setup

After completing the password change, issue the following command (make sure you are in your home directory):

```
~ece337/setup337
```

This batch/script file will copy the necessary setup files required to make Synopsys work properly into your root directory. **If you have trouble with this step please ask for assistance from your TA.** Failure to setup Synopsys properly will prevent you from being able to do the last part of this lab. Throughout the semester, you will encounter these commands to setup or copy information for tools. These are provided so you can get started working on your lab assignments rather than spending time copying source code into a file and possibly introducing errors.

Now whenever you open a new terminal window all the proper environment variables will be set up correctly for you to use the necessary tools. It is HIGHLY recommended that you keep a very neat and orderly directory structure for this course. An organized directory structure will prove very useful during your project. You should, unless instructed to do otherwise, create a directory for each lab exercise that is assigned throughout the semester. This will serve as a good way for you to remember where the code or design for that lab is located, so that you may refer to them once you begin working on your project. **NOTE: If instructed to create any directory throughout this class, make sure that you use the case that is specified.** This is required as some scripts to help you setup your accounts for the labs will be dependent on certain path names. Therefore, always be sure to name directories in the same case that they are given.

Exit the terminal window to set up the changes.

2.2. Git Setup

The setup script should have created an 'ece337' folder in your home directory. It has also initialized this folder as a git repository so that all course work within it can be version controlled.

Note: If you already familiar with Git and have a remote repository/host you would like to use for the main/origin repository, please clone your origin repository as the ~/ece337 folder in your account instead or add your host repository as a remote for the one initialized in the ~/ece337 folder. Keep in mind that these repositories must be private repositories due to the nature of the tool scripts and source code you will be writing for the lab exercises.

Now that your local repository is initialized and you may proceed with starting the first lab.

2.3. Lab 1 Specific Setup

Issue the following command in your new terminal to create the directory for this lab:

```
mkdir –p ~/ece337/Lab1
```

If you are not already aware, the "mkdir" command creates new directories. Also, the "-p" argument tells it to make any higher level directories it needs to in order to create the one at the end of the path, instead of exiting with an error if one of the folders in the path doesn't already exist.

Now, please change to your "Lab1" directory that you just created using the following command:

```
cd ~/ece337/Lab1
```

If the result of this command is not similar to "$HOME/ece337/Lab1", where $HOME is simply the path to your root directory, please change directories so that you are in the ~/ece337/Lab1 directory.

The next command that you will be issuing to the UNIX prompt is another script file. This script however is one that you will be using throughout the remainder of the semester and it would be a good idea to write a reminder to yourself to always run this script whenever you setup a new directory in which you plan to create a design. This command creates several directories inside the present working directory. The directories will be generated after issuing the command. In your terminal window, issue the following command:

```
dirset
```

Now list the contents of your directory by typing:

```
ls
```

Notice that the following directories now appear in your current directory:

Analyzed	This directory will be used as the WORK directory when you begin working with the Synopsys Design Tools of Design Analyzer and Design Compiler. This directory will actually be used to store the intermediate form of your design as you go from your Verilog source code to the output of the synthesis operation performed by the Synopsys tools.
Mapped	This directory is where the results of your synthesis on your Verilog source code will be placed. Essentially, this directory will contain Verilog files that describe your synthesized or mapped circuits. A synthesized or mapped circuit is simply the results of linking or mapping your Verilog source code to a given set of logic gates in a design library. This mapped form of the Verilog code is actually the form that you could use to help generate the Layout for a chip so that it could be manufactured.
Reports	This directory will contain the various reports that you instruct the tool to generate. These reports are text files which can contain several different types of information. For instance, a report file can be generated to detail the amount of area the circuit is utilizing in its current form. It could also contain a listing of the worst timing paths in the circuit. These paths are the ones in the circuit that have the largest time interval between the input and output signals for combination circuits, or the largest time delay between sequential elements, flip-flops or latches, in a sequential, clocked design. A report file could also contain a detailed breakdown of a timing path, so that one could see the delay of each gate in the timing path. Finally, another example of what could be contained in a report file is the power that the design dissipates.
Schematic	This directory is where you will store the schematics of the mapped designs generated by Design Compiler
Scripts	This directory is where you will store the scripts used to manipulate the tools into performing the desired optimizations on the designs you are working.
Source	This directory is where you should place all of your Verilog source code for the current design you are working on. This will give you one place to look for your source code in the given directory. It will also help the tools in searching for source code. If the source code is not located all in one directory, the search path for some scripts will not be able to find the code.
Docs	This directory is where you should place all documents created as part of the lab or design, such as answers to questions, data sheets, and project reports

This directory structure is being used because it will help you in keeping all the files for your lab assignments and your project in a neatly ordered structure that will make it easy for you to find files. As the Synopsys tools run, they generate a large number of intermediate files which have the same name as your module. These files are also retained after they are created in order to speed up the processing time if you would need to re-analyze that particular design again. Therefore, it is advantageous for you if you keep a directory where all these files are stored, and this directory is kept separate from where your Verilog source code files are stored. In your case, that analyzed directory is where all the intermediate files are stored and the source directory is where all your Verilog files are stored.

This directory structure WILL BE ENFORCED throughout the semester as the tools have been setup in such a fashion, so that their correct operation will be dependent upon the existence of the directories stated above. **Therefore, whenever you create a new directory for this class or your project, make sure to run dirset in it to ensure your directories are setup properly**.

Now that you have setup the folder for Lab 1, please run the following lab setup script.

`setup1`

This setup script will check that you have indeed properly setup the directory for the Lab and will retrieve copies of the needed files for the lab. It should also move you to be in the “Lab1” folder if you weren’t already.

2.4. Initial Lab 1 Git Commit

Now that your Lab 1 directory has been initialized you need to tell the Git repository to start tracking its files. To do issue the following series of commands:

```
git add ~/ece337/Lab1

git ci -m “Initial setup for Lab 1”
```

The first command adds the folder and all currently contained files to be tracked by Git. The second command commits their current versions with the tag/log message “Initial setup for Lab 1”, to clarify the reason for the commit/check in to the Git repository. As you create new files during the lab you will need use the ‘git add’ and ‘git ci’ commands to add them for track.

Only add your source code files, script files, makefile, and modelsim.ini file (the makefile and modelsim.ini are brought in through the add and commit of your initial lab folder setup). Adding generated or temporary files generally will cause problems with checking in updates, as well as wasting potentially large amounts of space in your accounts.

For a more exhaustive list of git commands you can use ‘git help’, online resources, and/or the Git quick reference posted on blackboard.

Once you have set up your repository and added the Lab1 directory properly, have a TA check off your work.

3. Compiling and Verifying the Verilog Code (you need to complete this section in order to be considered completing this lab)

Start a new terminal window and change directories until you are in the ece337/Lab1 directory. Simply type

```
vsim -i
```

This will launch the program called QuestaSim. This is the program that you will be using to simulate and verify that the various designs and you will be creating throughout this semester are correct. Once the window has opened, you will see a prompt inside the window, you must now type:

```
vlib work
```

This will create a folder in your Lab1 folder, which is necessary for compiling designs in QuestaSim. You only need to do this once per directory.

You must now compile your Verilog code for the comparator. In order to do this, type the following command at the QuestaSim command prompt:

```
vlog source/comparator.sv
```

The compiler will now run. When the compiler finishes you will see several errors, indicated by the lines in the QuestaSim window that are colored red. The code that was provided for the comparator had several intentional errors placed into it. This was done to introduce you to debugging the syntactical errors that you will undoubtedly encounter during this course (see the Appendix at the end of the lab for some hints on syntax). At this point, there are two ways you can view the Verilog code in order to correct the errors. One way is a standard text editor, such as nano, emacs, or vi. However, QuestaSim has a built-in text editor that is highly useful in debugging due to the fact that it color codes the reserved words in the Verilog language (note that some versions of emacs and vi support Verilog highlighting also). This color coding helps one locate some common typographical errors that are encountered by students when creating Verilog source files.

The editor window will appear in the upper right portion of your screen once you have opened a file. In order to bring up your source code for the comparator, proceed to the file menu in the window that just appeared and select:

File → Open...

Navigate into your "source" directory in the 'Open File' dialog box and choose your source file, comparator.sv, and select OPEN. Your code will now appear in the text window. At this point scroll through the source code in order to familiarize yourself with the colors that this editor assigns to different options of the code. You should also see the line numbers along the left of the page. Using that as a reference, return to your QuestaSim Window and see what errors you have, the line number of the error is indicated in parenthesis in the QuestaSim window. Begin to fix the errors that are present in the code based upon the line number information that is present in the QuestaSim window. A quick Verilog reference is attached at the end of this handout.

After each error has been fixed, save the file and recompile it. Simply retype the following command or push the up-arrow on the keyboard in order to cycle through your previous commands and select the command:

```
vlog source/comparator.sv
```

If you have problems, please ask a TA for help.

QUESTION: In the QuestaSim editor, what color are RESERVED Verilog words? What color are COMMENTS? (Please put your answer on the Evaluation Sheet)

When your code is error free and compiles correctly, run the following command from a terminal to submit your code file to be graded by the automated grading system used for this course:

```
submit Lab1c
```

Note: Since this submission tool is for grading your work and testing it for you, all submissions will require evidence of you at least attempting to check your work before actually grading it. In this case it will scan your Lab 1 folder the existence of the temporary files generated by QuestaSim during compiling. If it does not find them, it will simply notify you and terminate your submission request, so make sure to not delete them before trying to submit or you will need to recompile the code before the submission will be graded.

4. Simulating the Verilog Code (you need to complete this section in order to be considered completing this lab)

4.1. Simulation Setup

Return to your QuestaSim window and select

Simulate → Start Simulation...

Make sure that you are in the "Design" Tab; you should also ensure that you are inside the 'work' library inside the Lab1 folder. If you are not, then navigate there. You should see your comparator listed under Design Unit. At the '+' next to comparator, press the mouse once. You will now get a '-'. Select the comparator module (marked by an "M" symbol on the side) and select OK.

When you have finished loading, your prompt will return. You should now see a new window/box appears in your QuestaSim workspace called "Objects". You will see all the signals that are contained in the comparator. Select all of them (you can use "Ctrl" key or "Shift" key to select multiple signals) and in the QuestaSim window menu bar select:

Add → Wave → Selected Signals

Note: A second way to display all the top-level signals in a design is to type the following at the QuestaSim command-line:

```
add wave *
```

You should see all of the highlighted signals appear in a new window, labeled 'wave-default'. Note: QuestaSim might show all the workspace boxes in the workspace. When you feel that your workspace is getting too cluttered, you can always detach individual boxes to be a separate window by clicking the arrow button on the right hand corner of each workspace box.

At this point have your TA verify your Signal and Wave windows.

4.2. Testing Part A

We are now ready to perform a simulation on the Verilog source code function to ensure that it works properly. Return to the QuestaSim window. In order to test the logical function of the comparator, you will apply forces to the two input signals, A and B. This simulation will by no means be exhaustive, it is simply meant to introduce you to the syntax that QuestaSim utilizes. However, for your subsequent designs and your project you will need to provide adequate test vectors in order to ensure and demonstrate complete and proper functionality of the overall design.

The interface for QuestaSim that we will be using is text-based. This means that you will be typing in the forces you would like to apply to the inputs of your design. In order to enter the forces, go to the command prompt in the QuestaSim window and type the following commands:

```
force a 16#0000 0 ns, 16#ABCD 25 ns, 16#8808 50 ns –repeat 75 ns
force b 16#0000 0 ns, 16#9876 25 ns, 16#AAAA 50 ns –r 75 ns
```

The first of the two lines is instructing QuestaSim to force input signal a(15:0) to "0000" at time 0 ns, "ABCD" at time 25 ns, and "8808" at time 50 ns. The '16#' in front of the value means that the value being input is in hexadecimal or base 16. Being able to input values in different radices is very convenient especially with large vectors. For instance, one does not want to input a 32-bit bus, by supplying a 32-bit binary vector, this would get very tedious and time consuming when testing a large design. Therefore, the ability to input the values in hexadecimal will be critical time-saving feature of QuestaSim. The final portion of the command '-repeat 75 ns' instructs QuestaSim to "repeat" the force command with time unit 75 ns now being equivalent to time 0. That is a time 75 ns a(15:0) will be "0000", at time 100 ns a(15:0) will be "ABCD", at time 125ns a(15:0) will equal "8808". The sequence will repeat infinitely. The second line above instructs QuestaSim on how to force input signal b(15:0). Notice that the final portion of this command is '-r 75 ns'. This is simply a short-hand way to represent '-repeat' in QuestaSim.

Since the above commands provide all the input values in hexadecimal, one may be asking themselves at this point: What other bases/radices can I use input data? QuestaSim provides the ability to input data in the most common and convenient forms that you will require during the semester (hex, binary and decimal). The following illustrates 4 equivalent ways to assign a 4-bit quantity, d(3:0) to 0 at time 0 and then the decimal value 10 at time 20.

```
force d 0 0, 16#A 20
force d 0 0, 10#10 20
force d 0 0, 2#1010 20
force d 0 0, 1010 20
```

Notice the last two lines are binary

Now that the forces are setup, it is time to run the simulation. We are going to run the simulation in such a manner so that you can witness the effect of the repeat command that was supplied to the force command.

In the QuestaSim command terminal, issue the following command:

```
run 150 ns
```

By default the time step is picoseconds

This command tells QuestaSim to run a simulation for 150 nanoseconds. You can supply anytime you desire here. So if you wished to run a simulation for 400 nanoseconds, you would simply type: 'run 400 ns'.

Once you have hit ENTER after typing the run command, look back at the 'wave-default' window and examine its contents. You should see several green lines indicating your signals should now have waveforms associated with them. Verify that your output is as expected. You may find it useful to change the radix that the results are being displayed in. In order to do this, select the input signals A and B

(Methods for selecting signals were discussed earlier). After selecting these two signals select the following option from the menu bar:

Wave → Format → Radix → Hexadecimal

You may need to zoom in and out to see your waveforms more clearly. In order to do this, press the RMB inside the trace/waveform portion of the 'wave-default' window. **At this point the waveforms are not correct... You will need to focus on the code functionality rather than just correct syntax now.**

4.3. Testing Part B

Once you have fixed the functionality of your comparator using the prior test vectors, you are to verify it under a different input set just to check that things were not overlooked before. Before doing this however, you must reset your waveforms and simulation. In order to do this, simply type the following line at the command prompt in the QuestaSim terminal window:

```
restart
```

After typing this command and hitting RETURN, a dialog box will appear on the screen. Simply select Restart to accept the default setting in this box. This command will clear all your forces from your input signal and clear the traces from your previous simulation in the 'wave-default' window. It is important to note that whenever you RESTART a simulation all the FORCES for the simulations are lost. This means that whenever you restart and wish to re-run the previous simulation, you will have to re-type all your 'force' statements once again. In a later lab, we will explore a way to alleviate the tedium of having to re-type the force command lines.

You will now input the commands for this new simulation based upon the following lists of parameters. Please note all values are listed in DECIMAL and you should ENTER THEM IN DECIMAL:

Input A:

- Time = 0 ns Value = 9865
- Time = 20 ns Value = 8
- Time = 40 ns Value = 0
- Repeats every 60 nanoseconds

Input B:

- Time = 0 ns Value = 9864
- Time = 20 ns Value = 32767
- Time = 40 ns Value = 0
- Repeats every 60 nanoseconds

Now run this simulation for 180 nanoseconds.

In the resulting waveform window, you will want to change the radix on input signals A and B to DECIMAL in order to ensure that the values you entered are indeed what were used in the simulation. Examine your waveforms, ensure that they are correct and then **have your TA verify your work**.

You have now completed the potion of this lab dealing with QuestaSim. In order to exit QuestaSim, select the following Menu option:

File → Quit

Or type the following command at the QuestaSim terminal prompt:

quit

In either option, click on 'Yes' to confirm that you wish to quit QuestaSim.

NOTE: For future reference, if you wish to view the help documentation for QuestaSim it is available by issuing the "ms_docs" command at a UNIX terminal. This will bring up an Acrobat Reader session that contains the reference manual for QuestaSim. This would prove useful if you wish to explore the various commands available in QuestaSim.

5. Synthesizing the Comparator (you need to complete this section in order to be considered completing this lab)

The next portion of this lab deals with synthesizing your comparator. The process of synthesizing a design involve taking the Verilog source code that you have written and mapping/converting it to a form that can be realized in standard logic cells that are available in a design library. Examples of standard logic cells are Boolean Logic gates such as NAND, NOR, INVERTER, XOR, XNOR, MULTIPLEXORS (MUXes), sequential circuits such as LATCHES and FLIP-FLOPS, and complex gates such as AND-OR-INVERT, AND-NOR, and OR-NAND. Thus this program allows you to turn their Verilog code into schematics that are based upon cells in a library. This would then allow the designer to layout the design so that it could be fabricated/manufactured, tested, and then sold.

Unlike software courses you have taken, once you have gotten your Verilog code to compile and thoroughly tested you are not done. The next step is to ensure that your source code is synthesizable. Your code may perform its function correctly in the simulator but if you are unable to synthesize the design, you will not be able to manufacture a part to sell. Therefore, you must synthesize your final Verilog code in order to make sure that it can produce a gate level representation of your source code. You must then also test this gate level representation to ensure that it also performs the logic function you expect it to.

The tool that you will be using in this class in order to perform the synthesizing operation is from the tool vendor Synopsys and you will primarily be using design compiler. To do this we have a provided a pattern rule target in your provided makefile for synthesizing designs. The use of a pattern rule allows the same make syntax to be used to both directly synthesize a design and also automatically synthesize a design needed if make detects that its source file(s) have been modified since it was last synthesized. Automatic use of pattern rules requires a defined relationship between the design file and its source file(s) or the use of variables that get set with the use of the target. For this class all source files should use the same name as the intended mapped version which will allow the pattern rule to be used automatically to synthesize designs that do not contain sub modules, and a variable is used to specify the files needed for sub modules of more complex designs. Open up your makefile and use the quick Make discussion in the GNU Make quick reference on the course blackboard site to answer the following questions.

How many options/variables are available to be used with the synthesis pattern rule "mapped/%.v:"?

(Please put your answer on the Evaluation Sheet)

What do you think the command should be to synthesize your comparator file?

(Please put your answer on the Evaluation Sheet)

When you have answered these questions correctly, **have a TA check off your work**.

Now execute the make synthesis target to synthesize your mapped file (note you need to be in your ~/ece337/Lab1 directory). In you terminal window view the log file that was created, by issuing the following command:

```
less comparator.log
```

Note: Use the 'b' key to navigate backwards and the spacebar to navigate forward a terminal length at a time.

You might be wondering why we go through the trouble of synthesizing your Verilog source code. As mentioned above this process actually creates a digital circuit netlist, whereas your source code is just a simulation of the circuit. You will undoubtedly find out sometime this semester that just because your source code works, does not mean that your synthesized version will.

You should be able to find your newly created mapped file in your 'mapped' directory. After each synthesis of a design you should check the synthesis log file to see if any errors were encountered. In addition to manually reading through the log file you can use the following script to quickly check for the presence of error and warning messages:

```
syschk <design name>
```

Note: This script simply uses grep to extract any lines that have the structure for an error or warning message. You may also use the optional flag '-w' to ignore warning messages and only output errors.

If you have any errors, call over you TA.

When you have an ERROR-FREE synthesis of your comparator, run the following command from a terminal to submit your code file to be graded by the automated grading system used for this course:

```
submit Lab1s
```

Note: Since this submission tool is for grading your work and testing it for you, all submissions will require evidence of you at least attempting to check your work before actually grading it. In this case it will scan your Lab 1 folder the existence of the temporary files generated by design compiler and the makefile during synthesis. If it does not find them, it will simply notify you and terminate your submission request, so make sure to not delete them before trying to submit or you will need to resynthesize the code before the submission will be graded.

6. Simulating the Synthesized Verilog Code (you need to complete this section in order to be considered completing this lab)

The next section of this lab is to simulate your synthesized circuit. There are two main methods to do this; manually open QuestaSim, compile the mapped file, and then simulate the mapped file like you did with the source versions or you may use the make file simulation targets to automate much of this process for you. The makefile will list of the prepared usage help message with information about the main targets that are provided if you type the following command within your 'Lab 1' directory:

```
make
```

It is recommended that you start learning how to use and modify makefiles as early in the course as possible as they will enable you to simplify and speed up a lot of the design flow for later labs.

You need to do the following things to receive the last check-off:

- Compile your synthesized circuit (the resulting synthesized Verilog code is in your mapped directory)
- Load the synthesized design into QuestaSim
- Add the appropriate waveforms
- Use force statements to do TESTING PART A (seen above in handout)

All these steps have been described previously in this lab handout when you were simulating the source version of the Verilog code.

Once you have simulated your synthesized circuit and it works properly, run the following command from a terminal to submit your code file to be graded by the automated grading system used for this course:

```
submit Lab1f
```

Note: Since this submission tool is for grading your work and testing it for you, all submissions will require evidence of you at least attempting to check your work before actually grading it. In this case it will scan your Lab 1 folder the existence of the temporary files generated by QuestaSim during simulation of your synthesized design. If it does not find them, it will simply notify you and terminate your submission request, so make sure to not delete them before trying to submit or you will need to resimulate the design before the submission will be graded.

Make sure to update Git with any changes you've made to your source and script files and add any new files that you have created. To see if there are any files that need to be added or updated use the following command, which will check the status of all files in the "source" and "scripts" folders relative to the git repository:

```
git status ./source ./scripts
```

7. Digital Design Refresher (Postlab)

This next section of the lab is to refresh your memory on digital logic design. All deliverables in this section must be done as digital documents and submitted as PDF or common image files via the **"submit Lab1p"** command.

For this class all diagrams (state transition and RTL) must be done as digital drawings and stored in your 'docs' folder as images or PDF file(s). There are a variety of diagraming programs available to you. Any windows lab, as well as through software remote, should have Microsoft's Visio installed. The Linux machines should have DIA installed. Alternatively, DIA is a free to use diagraming tool that you can download and install for any windows or Linux machine that you own. Additionally, all table like documents (present-state next-state tables, k-maps, etc.) must be done in a digital format as well and submitted as PDF file(s). Failure to follow these directions will result in a 25% grade penalty on the offending deliverable(s). This rule is in place both to guarantee a minimum level of readability of submitted deliverables and to require you to have a digital form/second copy of your work to use with related/dependent work in later labs.

7.1. Sensor Detector

Given the following specifications of a Sensor Error Detector:

The Sensor Detector that you are designing monitors four (4) sensors and reports the existence of an error condition when at least two (2) independent low priority sensor events occur simultaneously or there is an event from the high priority sensor. However, the sensors on the third (3rd) and fourth (4th) inputs are a paired set and the triggering of either one or both only counts as 1 distinct sensor event. The high priority sensor is attached to the first (1st) input and a low priority sensor is attached to the second (2nd) input. A sensor detector like this might be used in a manufacturing process to detect when a significant enough error occurs that requires the production line to halt. Sensor events correspond to a logic '1' and a logic '0' corresponds to no event from that sensor. Consider two examples (MSB to LSB): "1100" indicates that both redundant sensors have had events while the other two haven't and thus the output should be a logic '0'. However, "0001" indicates that the high priority sensor had an event and thus the output should be a logic '1'.

- **Generate the Truth Table for the Sensor Error Detector Circuit.**
- **Using K-Maps, determine the optimal 2-level logic circuit, in SUM OF PRODUCTs form.**
- **The above items should be signed off by a TA for points, as you will be using it in Lab 2.**

7.2. "1101" detector.

The purpose of a "1101" detector is to assert an output whenever the sequence "1101" is detected in a serial input data stream. The module declaration should have the following ports:

```
input wire n_rst,
input wire clk,
input wire i,
output wire o
```

Below are the requirements for this section of the post lab exercises

- The input is serial.
- When the sequence "1101" is detected, '1' should be asserted.
- Must detect overlapping occurrences of the desired 1101 pattern (I.e., an input sequence of "001101101100" should detect the 1101 pattern twice).
- Neatly and legibly illustrate the STATE TRANSITION DIAGRAM for a Serial "1101" sequence detector as a Moore model state machine.
- Neatly and legibly illustrate the STATE TRANSITION DIAGRAM for a Serial "1101" sequence detector as a Mealy model state machine. (hint: your Mealy model will have one less state than your Moore model. Why?)
- Every state machine will always consist of three components: A state register, next state logic and output logic. Create a block diagram depicting how the input port and the output port are connected to those components and how all these components are connected to each other in your Moore model. Draw another block diagram for your Mealy model. How are they different? The diagrams you just created are called "RTL" (Register Transfer Level) diagrams of Moore and Mealy machines. We will discuss more about this later in the semester.

7.3. Hardware Building Blocks

In digital design there is a pool of very commonly used hardware modules affectionately referred to as building blocks. For this section of the digital refresher we are going to focus on arguably three of the most common ones, the most significant bit first (MSB) serial-to-parallel (StP) shift-registers, MSB parallel-to-serial (PtS) shift registers, and synchronizers.

7.3.1. Synchronizer

Whenever a design has input signals which are asynchronous to the system clock, these input signals need to be synchronized. Assuming the inputs are read on the rising edge of the system clock, synchronization will prevent reading a transition in the input signal. In order to synchronize a signal, the input must go through TWO flip-flops clocked on the system clock.

In general the synchronizer circuit has two purposes:

1. Synchronize the input signal to the system clock domain
2. Reduce the metastability of the signal.

Metastability (the unknown value of a signal) is an issue when synchronizing an asynchronous signal into a flip-flop device. If the incoming asynchronous signal happens to change its state (logic value) at the same time the flip-flop device is capturing the data or during the setup or hold regions, an indeterminate value on the output of the flip-flop may be seen. Now we ask, "How does this affect the functionality of the circuit?" It can adversely affect the circuit if the captured data is fanned out to multiple places within the circuit. The indeterminate value of the captured signal can be interpreted differently (either a logic '1' or '0') by different logic gates within the receiving circuit, thus leading to a possible malfunction of the circuit. To try to reduce the possibly of this occurring, a second flip-flop is attached in series to the data capturing flip-flop. It will be the output of this second flip-flop that is used inside the receiving circuit (see Figure 8). Please note that adding the second flip-flop does not guarantee the output of the second flip flop to be stable but the chances of the output being metastable are greatly reduced (note: In some modern designs, three flip-flops in series are used to synchronize asynchronous signals to a clock domain).

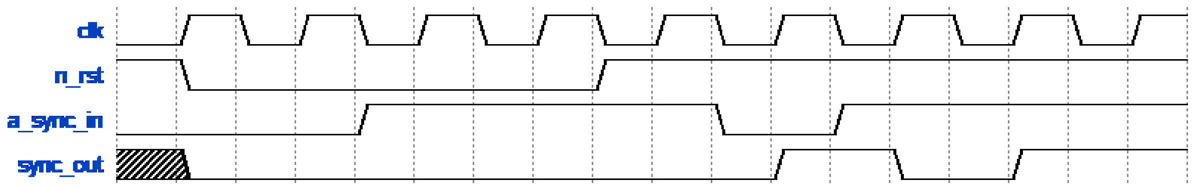

Figure 1: Timing waveform for 2-stage synchronizer for an active high input

Create a Schematic diagram of a 2 Flip-Flip synchronizer. This diagram should be signed off by a TA for points, as you will be using it in Lab 2.

7.3.2. Most Significant Bit First Serial to Parallel Shift Register

Serial-to-parallel shift registers are a vector of flip-flops arranged so that:

- All flip-flops use the same clock signal and only update their values on same clock signal edge (rising edges for this class)
- Each bit to get the value of its less significant neighbor bit (or the serial input in the case of the least significant bit) while the shift enable is active
- Each bit retains its current value while the shift enable is in-active.
- All flip-flops must asynchronously reset to a predetermined value while the reset signal is active (usually the idle/default value for the serial input of the shift register which is normally a logic '1')

This allows for a most significant bit first serial stream of bits to be accumulated into a parallel form for easier use by other modules. The standard practice in this class (as well as good practice in general) is to have the reset signal for a serial-to-parallel shift register reset the value of each bit to idle/default value of its serial input, which is usually a logic '1'.

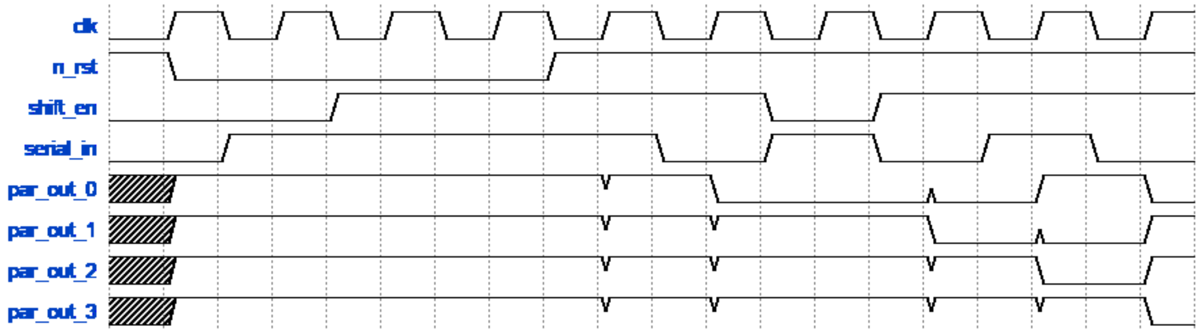

Figure 2: Timing waveform diagram for 4-bit MSB serial to parallel shift register

Create a RTL diagram for a most significant bit first serial-to-parallel shift register that holds four (4) bits of data and uses a single bit active high shift enable, and an active low reset signal.

Create a schematic diagram for a most significant bit first serial-to-parallel shift register that holds four (4) bits of data and uses a single bit active high shift enable, and an active low reset signal.

7.3.3. Most Significant Bit First Parallel to Serial Shift Register

Parallel-to-serial shift registers are a vector of flip-flops arranged so that:

- All flip-flops use the same clock signal and only update their values on same clock signal edge (rising edges for this class)
- Each bit will change to the value of its corresponding bit in the parallel input vector when the load signal is active
- Each bit will change to the value of its less significant neighbor bit (or the reset value in the case of the least significant bit) while the shift enable is active and the load signal is inactive
- Each bit will retain its current value while both the load and shift enable signals are in-active.
- All flip-flops must asynchronously reset to a predetermined value while the reset signal is active (usually the idle/default value for the serial output of the shift register which is normally a logic '1')

This allows for a most significant bit first serial stream of bits to be generated from a parallel form for transmission to other designs or modules.

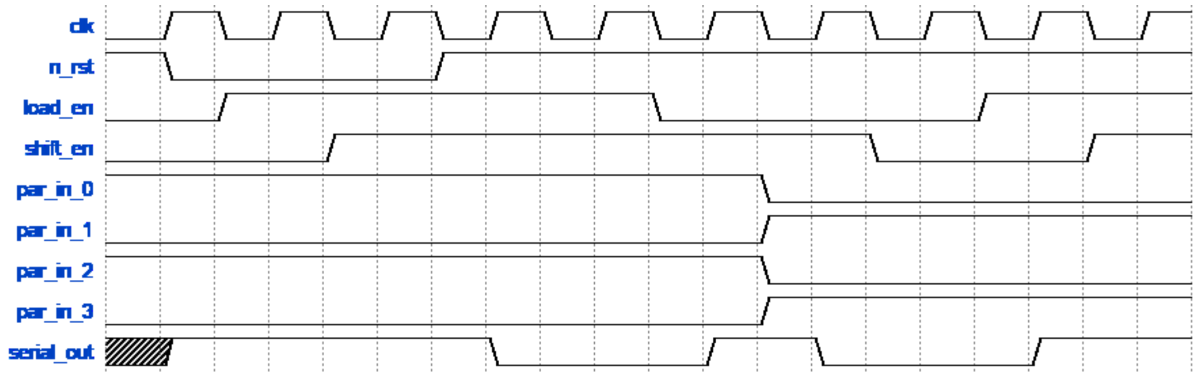

Figure 3: Timing waveform diagram for a 4-bit MSB parallel to serial shift register

Create a RTL diagram for a most significant bit first parallel-to-serial shift register that holds four (4) bits of data and uses a single bit active high shift enable, a single bit active high load signal and an active low reset signal.

Create a schematic diagram for a most significant bit first parallel-to-serial shift register that holds four (4) bits of data and uses a single bit active high shift enable, a single bit active high load signal and an active low reset signal.

8. Deliverables

- **Turn in your check-off sheet at the beginning of Lab 2**.
- **All Digital Refresher Solutions must be submitted as digital documents (no scanned hand-drawn or hand-written documents) in PDF format via the 'submit Lab1p' command.** Standard image file formats are acceptable for diagrams as well. **All documents must be put in your 'ece337/Lab1/docs' folder**
- You will need your solutions to the Digital Refresher problems to do labs 2-4.