ECE337 Lab 1 – Introduction to VHDL Simulation via Modelsim and Digital Logic Design Refresher

The purpose of this first lab exercise is to help you become familiar with the VHDL synthesizer, Design Compiler (by Synopsys), and the VHDL simulator, Modelsim (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 VHDL 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 VHDL model via Modelsim.
- Synthesize the VHDL 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.

Required Background Knowledge and Introduction to Lab Format

Throughout this course it is assumed that the student has rudimentary programming skills that are taught in a freshman or sophomore level programming class, such as EE264. Also a detailed understanding of digital logic design, both sequential and combinational, is required in order to complete some of the laboratory exercises and a necessity when work begins on your respective semester projects as you, the students, will be creating designs of your own**. The digital logic design requirement includes, but is not limited to, Karnaugh Maps (K-Maps), Truth Tables, and State Machine design, both Moore and Meally 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 the user entering the commands via the menu options. Through this exposure it is hoped that the student 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 the course webpage http://cobweb.ecn.purdue.edu/~ece337/ and Blackboard https://blackboard.purdue.edu/. **In order to follow class discussions, announcements and your grades, you are encouraged to use Blackboard for this course**. It is highly recommended that you download and print out the lab prior to arriving at class. **Also, 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 labs should be brought up at this point. The following 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 or VHDL code.
`QUESTION`	This indicates a question being posed to the student 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 insure 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 the student 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 the student should use the button to bring up a context sensitive menu in the current design tool.

1. 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 Potter 360 VLSI design please let your TA know immediately. Please login using the **Java Desktop Environment** session. This desktop is more user-friendly and easier to customize. If it does not appear automatically, please ask your TA for assistance.

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 locate the item called “Tools” in the menu select the arrow to the right of it. Now select the item “terminal”. This should bring up a new window containing a command prompt. You can run any application by typing “grid” followed by the name of the application.

In your terminal window, issue the following command (make sure you are in your home directory):

```
~ece337/setup337
```

If the above command gives an error, try:

```
/home/ecegrid/a/ece337/setup337
```

This command is actual an alias to a batch/script file in the ece337 directory that will copy the VHDL source code for the 16-bit Comparator to you ECE337/Lab1 directory. This batch/script file will also copy the necessary setup files required to make Synopsys work properly into your root directory. Moreover it will allow you to change your password from the given random password to you own password. **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. Upon completion of the previous command, you should open a new terminal and go to your ~/ece337/Lab1 directory. Verify this by issuing the following command at the UNIX prompt:

```
pwd
```

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 to do 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 VHDL 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 VHDL source code will be placed. Essentially, this directory will contain VHDL files that describe your synthesized or mapped circuits. A synthesized or mapped circuit is simply the results of linking or mapping your VHDL source code to a given set of logic gates in a design library. This mapped form of the VHDL 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 your VHDL 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 notlocated all in one directory, the search path for some scripts will not be able to find the code.

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 entity. 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 VHDL 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 VHDL 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.**

2. Compiling and Verifying the VHDL 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. At the prompt type:

```
grid vsim –i &
```

If you are using a Sun Blade, simply type

```
vsim –i &
```

The “&” in the end of the command is to prevent Modelsim from locking up your terminal. This is a UNIX symbol to prevent any program from locking up your terminal.

This will launch the program called ModelSim. 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 ModelSim. You only need to do this once per directory. You must now compile your VHDL code for the comparator. In order to do this, type the following command at the ModelSim command prompt:

```
vcom source/comparator.vhd
```

The compiler will now run. When the compiler finishes you will see several errors, indicated by the lines in the ModelSim 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 VHDL code in order to correct the errors. One way is a standard text editor, such as Pico, emacs, or vi. However, ModelSim 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 VHDL language (note that some versions of emacs and vi support VHDL highlighting also). This color coding helps one locate some common typographical errors that are encountered by students when creating VHDL source files.

The editor window will appear in the upper right portion of your screen. 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.vhd,* 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 ModelSim Window and see what errors you have, the line number of the error is indicated in parenthesis in the ModelSim window. Begin to fix the errors that are present in the code based upon the line number information that is present in the ModelSim window. An appendix of VHDL syntaxes has been provided for you in this lab handout.

After each error has been fixed, save the file and return to the ModelSim window where you will need to recompile. 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:

```
vcom source/comparator.vhd
```

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

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

When your code is error free and compiles correctly, **have a TA check off your work.**

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

Return to your ModelSim 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 '-'; make sure that a behavioral architecture appears. Select the comparator entity (marked by an "E" 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 Modelsim 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 ModelSim 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 ModelSim command-line:

```
Add Wave *
```

You should see all of the highlighted signals appear in a new window, labeled 'wave-default'. Note: Modelsim 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.

TESTING PART A:

We are now ready to perform a simulation on the VHDL source code function to ensure that it works properly. Return to the ModelSim 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 ModelSim 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 ModelSim 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 ModelSim 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 proceeding two lines are very important, and the force command is a ModelSim command that you will use quite extensively throughout this course. The first of the two lines is instructing ModelSim 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 ModelSim. The final portion of the command '-repeat 75 ns' instructs ModelSim 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 ModelSim 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 ModelSim.

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? Modelsim 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 ModelSim command terminal, issue the following command:

```
run 150 ns     (by default the time step is picoseconds)
```

This command tells ModelSim 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:

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 have the TA verify that your waveforms are correct.**

TESTING PART B:

You re-test the functionality of your circuit using a different set of input test vectors. 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 ModelSim 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 table. 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 ModelSim. In order to exit ModelSim, select the following Menu option:

`File` → `Quit`

Or type the following command at the ModelSim terminal prompt:

```
quit
```

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

NOTE: For future reference, if you wish to view the help documentation for ModelSim 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 ModelSim. This would prove useful if you wish to explore the various commands available in ModelSim.

4. 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 VHDL 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 the user to turn their VHDL 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 VHDL 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 VHDL 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 provided a script called `scriptgen`, which will help you create 'comparator.scr' which will be used to synthesize the circuit. This script can be invoked in the following form:

```
scriptgen [options] <Design_Name>
```

Where: [options] are several flags that can be set to tell the script to perform certain functions that are not part of its default operation.

<Design_Name> is the name of the VHDL source file, without the ".vhd" extension on it.

This script will generate a file named: <Design_Name>.scr in the `scripts' directory. You can then run the script that results from this command. The script that is generated by scriptgen by default is an executable file that launches the command line version of DC Shell, dc_shell-t, and performs the operations of analyzing, elaborating and compiling the design. The script also causes DC Shell to create a timing and area report for the design and store the resulting information in the `reports' directory with the filename <Design_Name>.rep. This script will also produce the mapped version of the Source VHDL code, the result after you compile the source code, and write this result to the `mapped' directory with a filename of <Design_Name>.vhd. In addition, the script output from scriptgen will produce VERILOG version of the mapped version and write this file to the `mapped' directory with a filename of <Design_Name>.v. Finally this script will produce a postscript file that contains the schematic of the design that was created. This postscript file will be placed in the `schematic' and will be named <Design_Name>.ps.

To get familiar with `scriptgen`, in a terminal window get the help menu by doing the following command:

```
scriptgen -h
```

QUESTION: How many options are there available to be used with scriptgen? (Please put your answer on the Evaluation Sheet)

QUESTION: What do you think the command should be to create our 'comparator.scr' 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 `scriptgen` command line to create your script file (note you need to be in your ~/ece337/Lab1 directory). In you terminal window view the file that was created by the following command:

```
cat ./scripts/comparator.scr
```

You should see something similar to the following:

```
#!/bin/csh -f
/package/eda/synopsys/syn-V-2004.06-SP2/linux/syn/bin/dc_shell-t -f ./scripts/comparator.fcr |
tee comparator.log
```

Now you are ready to synthesize the circuit**. Before synthesizing the circuit you might want to check the comparator.scr file. Try to understand the Compiler commands. Can you figure out their functions?** Again in you ~/ece337/Lab1 directory type the following command:

```
grid ./scripts/comparator.scr
```

or, on the Sun Blades, simply

```
./scripts/comparator.scr
```

You might be wondering why we go through the trouble of synthesizing your VHDL 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 some time this semester that just because your source code works, does not mean that your synthesized version will.

When the synthesis process finishes you should be able to find two newly created files in your `mapped` directory. It is also recommended to look at the log files and check for any errors. In your ece337/Lab1 directory enter... less comparator.log. Use the 'b' key to navigate backwards and the spacebar to navigate forward.
If you have any errors, call over you TA.

When you have an ERROR-FREE synthesis of your comparator, **have a TA check off your work.**

5. Simulating the Synthesized VHDL 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. If you have closed down ModelSim already then you need to open it back up. But first remove your work directory and recreate it. Remove it with the following command:

```
rm -rf work
```

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

1. Compile your synthesized circuit (the resulting synthesized VHDL code is in your `mapped` directory)
2. Load the synthesized design into ModelSim (using the **`Simulate`** → **Start `Simulation...`** from the menu and select the comparator entity)
3. Add the appropriate waveforms
4. 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 VHDL code.

Once you have simulated your synthesized circuit and it works properly, **have a TA check off your work.**

6. Digital Design Refresher

This next section of the lab is to refresh your memory on digital logic design. Write down all your solutions on separate sheets, attach them to your check-off sheet and turn them in with your check-off sheet.

6.1. Functional Protein Detector

Given the following specifications of a Functional Protein Detector:

The Functional Protein Detector System that you are designing relies on the assumption that the protein of interest is only functional when there are more than two different amino acid characteristic properties present in the input vector. There will be a five bit input vector to the system logic, with each bit coming from a different amino acid sensor. The amino acids, and their corresponding sensors, are organized into two groups, A and B. The three lower significant bits of the vector represent group A. The upper two significant bits of the vector represent group B. Group A is made up of three different amino acids with different characteristic properties. Group B is made up of two different amino acids with the same characteristic property. A logic '1' in the input vector means the appropriate amino acid is present. If more than two different amino acid characteristic properties are present then the output should be a logic '1', else it should be a logic '0'. Consider two examples: 00111 indicates that three amino acids with different characteristic properties from group A are present, thus the system output would be '1'. However, 11010 indicates that two amino acids from group B and one from group A are present, resulting in an output of '0' because only two *different* characteristic properties are present.

- Generate the Truth Table for the Functional Protein Detector Circuit.
- Illustrate the K-Maps that were used to determine the Optimal 2- Level Logic Circuit, in SUM OF PRODUCTs form. The K-Maps should be for the next-state and output.

6.2. "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 entity declaration is:

```
RST: in STD_LOGIC;
CLK: in STD_LOGIC;
I: in STD_LOGIC;
O: out STD_LOGIC
```

- The input is serial.
- When the sequence "1101" is detected, '1' should be asserted.
- 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?)
- For both cases, Moore and Mealy, generate the Present State – Next State Table.
- For both cases, illustrate all necessary K-maps for the design of Moore and Mealy versions.
- Illustrate the K-Maps that were used to determine the optimal 2- Level Logic Circuit, in SUM OF PRODUCTs form (next-state and output).

- Every state machine will always consist of three components: A state register, next state logic and output logic. Draw 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.

Turn in your check-off sheet along with your solutions to the digital design refresher problems and prelab 2 at the beginning of Lab 2. You will need your solution to the Functional Protein Detector problem to do your prelab 2.

Appendix: Quick Reference to VHDL

Entity declaration

```
entity (entity-name) is
     port((signal-names) : (mode) (signal-type);
          (signal-names) : (mode) (signal-type);
           …
          (signal-names) : (mode) (signal-type));
end (entity-name);
```

(Reproduced from Wakerly, Digital Design Principles & Practices, 3rd edition, Prentice Hall 2000, table 4-27)

Architecture definition:

```
architecture(architecture-name) of (entity-name) is
  type declarations
  signal declarations
  constant declarations
  function definitions
  procedure definitions
  component declarations
begin
  concurrent-statement
  …
  concurrent-statement
end (architecture-name);
```

(Reproduced from Wakerly, Digital Design Principles & Practices, 3rd edition, Prentice Hall 2000, table 4-28)

Assignment Operator

- for the type `signal` the assignment operator is <=
- for the type `variable` the assignment operator is `:=`

Declaring signal/variable

- to declare an object of type `signal`, it must be declared inside the architecture before the `begin`, in the form of `signal foo : std_logic;`
- to declare an object of type `variable`, it must be declared in a process statement before the `begin`, in the form of `variable bar : integer;`

Vectors (examples will use: `foo_vec : std_logic_vector(2 downto 0)`)

- individual bits in a vector can be access like so: `foo_vec(1) <= '0';`
- the vector can be assigned by individual signals, for example to do a right circular rotate: `foo_vec <= foo_vec(0) & foo_vec(2 downto 1);`

Dataflow Style:
Dataflow style directly describe how the inputs flow to the outputs through the various logic functions and apply only to combinational logic. You can think of each line as describing a logic gate or collection of logic gates. All dataflow statements are "concurrent".

- Conditional Dataflow Statements Syntax Examples:

```
o  foo <= '1' when (condition)else '0';
o  with (select signal) select
      ▪  foo <= (assigned signal) when '1',
                (assigned signal) when '0',
                '0' when others;
```

Behavioral Style:
Behavioral style of coding is similar to C programming style. Behavioral style allows constructs such as "if", "for", "case", etc. Need to be emphasized that VHDL is not C! Therefore care should be applied when using the behavioral style. Behavioral style is easy to write but also easy to foul up. Behavioral style is best used for complex combinational logic and the code has to translate into combinational logic. For simple design, it is suggested that dataflow style should be used, unless instructed otherwise.

Behavioral style consists of sequential statements contained in a process block. Process block will translate to a block of circuitry during synthesis. As a whole, a process block is a concurrent statement that merely contains sequential statements (e.g. the if/then/else). Constructs such as "if", "for", "case" can only be used inside a process block. The Process block, as a whole, is a "concurrent" statement

Reminder: concurrent statements are interpreted as parallel, while sequential statements are interpreted in order

- syntax for a process block:

```
process(sensitivity list)
   type declarations
   variable declarations
   constant declarations
   function definitions
   procedure definitions
begin
   sequential statement
   …
   sequential statement
end process;
```

(Reproduced from Wakerly, Digital Design Principles & Practices, 3rd edition, Prentice Hall 2000, table 4-55)

Sensitivity list consists of signal names. Any time a signal in the sensitivity list changes states, the code in the process is re-evaluated. This is only for source simulation purposes and means very little for synthesis. Missing signals from sensitivity list can cause difference between source, synthesized and hardware behavior.

Other sequential statements:

- if statements must use the following structure:

```
if (condition) then
...
elsif (condition) then
...
else
...
end if;
```

- case statement:

```
case (condition) is
     when (choices) => sequential statements
     …
     when (choices) => sequential statements
end case;
```

(Reproduced from Wakerly, Digital Design Principles & Practices, 3rd edition, Prentice Hall 2000, table 4-59)

- for loop:

```
for (identifier) in (range) loop
     sequential statement
     …
     sequential statement
end loop;
```

(Reproduced from Wakerly, Digital Design Principles & Practices, 3rd edition, Prentice Hall 2000, table 4-62)