Lab Worksheet 03

Part 1:

Main Objective:

To learn techniques for testbenches that verify combinational circuits.

Sub-Objectives:

1) **To become familiar with consulting the official VHDL document to resolve questions about VHDL.**
2) **To become familiar with Behavioural VHDL and coding styles applied to testbench code.**
3) **To create scripts that make design tasks efficient and uniform.**
4) **To develop a systematic procedure for producing coherent waveform displays in ModelSim and/or Questa.**
5) **To write sequential code loops in VHDL.**
6) **To construct arrays of records containing test vectors.**
7) **To detect differences between actual signal response and expected signal response and communicate messages to the transcript.**

Part 2:

Main Objective:

To study and apply synthesizable and non-synthesizable VHDL code.

Sub-Objectives:

1) **To create a behavioural model for an Entity that detects whether a given integer signal is divisible by SEVEN.**
2) **To design a synthesizable combinational circuit that detects divisibility by SEVEN.**
3) **Create a testbench that uses a behavioural (intent) model in parallel with a DUT to functionally verify the DUT.**

Lab Worksheet 03

Introduction

VHDL:

VHDL is not a programming language. It is a language that was originally created to facilitate discrete-event simulation of high-speed digital circuits. While it was not the original intent, VHDL is also used to facilitate the synthesis and realization of these circuits.

The definition of the VHDL language is provided by the IEEE standard 1076, in the "language reference manual", **LRM**,

"IEEE Standard VHDL Language Reference Manual – IEEE-STD-1076-1987",
"IEEE Standard VHDL Language Reference Manual – IEEE-STD-1076-1993",
"IEEE Standard VHDL Language Reference Manual – IEEE-STD-1076-2002",
"IEEE Standard VHDL Language Reference Manual – IEEE-STD-1076-2008",
"IEEE Standard VHDL Language Reference Manual – IEEE-STD-1076-2019"

Notice that the standard was recently updated. The 20.1.1 version of Quartus does not support the VHDL-2019 standard. We shall use the VHDL-2008 standard.
(*refer to the document "LWS-01.1-SW-Install-Setup-350-xxxx"*)

The LRM is not free.
As SFU students, you (legally) have access to the document through the SFU library.
When you need to learn new aspects of VHDL you should refer to the LRM. Searching internet resources often produces incorrect results resulting in badly designed code. You are training to become a professional engineer and thus should learn the methods of professionals. Learn to use the LRM to resolve any questions you may have about VHDL.
(*This document references the LRM-VHDL-2008 document. If you are just learning VHDL you may wish to simultaneously reference the LRM-VHDL-1993 as the older text is not obscured by newer features such as callbacks and PSL.*)

While VHDL is NOT a programming language, it can be used in such a manner so as to appear like a programming language.

Read Chapter 14 of the LRM.
You should understand the general concepts of elaboration, initialization and executing simulation cycles. In addition, you should understand the fundamental structure of a simulation model containing processes and nets.

In general, code that is written like a "computer program" is either not synthesisable or if it is, results in a badly designed circuit.

- A testbench is a top-level Entity that coordinates verification of a circuit. The circuit of interest is referred to as the "device under test", **DUT**. (sometimes "unit under test", UUT). An instance of the DUT is instantiated within the testbench.
- A testbench is never synthesised. Testbench code is a typical example of when you would see VHDL code that resembles a computer program.

Lab Worksheet 03

Testbenches for Functional Verification:

Questa is compatible with ModelSim

Conventions:

To keep organised let's adopt some simple conventions;
refer to the document "LWS-01.x-SW-Install-Setup-350-xxxx"

1) Use a directory called "Simulation" as the project folder for ModelSim/Questa. Place this directory in the root level of your main project. The main project directory also acts as the project directory for Quartus.

2) Place the sourcecode for all of your designs in the "SourceCode" directory which should already be in the root level of the main project. Both Quartus and ModelSim/Questa will need access to these files.

3) Store all testbench sourcecode in the root of the "Simulation" directory. Quartus will never need to access testbench files.

4) All testbenches will contain only one instance of the "device under test", DUT.
   Name all testbenches using the same name as the DUT but beginning with the letters "TB".

Preliminary Verification:

```
library ieee;
Use ieee.std_logic_1164.all;
Use STD.TEXTIO.all;
Use ieee.numeric_std.all;
```

Create a testbench that instantiates the Entity OddParity.

In addition to the usual package from the ieee library,

- include the package, numeric_std.
  - ✓ This package is necessary for signed and unsigned datatypes.
  - ✓ You should get into the habit of always including this package.

- Include the package STD.textio. We will use this package later.
  - ✓ The STD library is implicitly declared in all VHDL code.

- Do Not include obsolete old packages.
  Warning: The internet is full of badly written VHDL code using obsolete libraries.
  - ✓ Using code with obsolete libraries will not save you time.
  - ✓ Instead it will waste time resolving numerous unnecessary compatibility issues that have nothing to do with your design task.

Lab Worksheet 03

General Procedures for Functional Verification:

Part 1:

OddParity Verification:

1) Create a ModelSim/Questa project using the “Simulation” folder as the project folder.

2) Create a new file and declare an `Entity` called TBOddParity that has no inputs and no outputs.
   Include a `Generic` parameter called “N” , that is of type natural and defaults to 16.
   (LRM-2008 §6.5.6.2, page 78, & LRM-2008 §6.5.7.2, page 84.)

3) Within the `architecture`, add an instance of the `Entity` OddParity(Tree) directly.
   - For now, use direct Entity instantiation. We will change to components later.
   - Label the instance **DUT:**
   - Use the convention that local signals connecting to the DUT, have that same name as the formal port parameters but beginning with the letters “TB”.
     example: **`port map ( sigX => TBsigX, sigY => TBsigY )`**

4) Add the files to the ModelSim project and compile to verify that there are no syntax errors.
   - You should arrange the compile order so that leaf nodes compile first.
   - Warning: if you copy a project hierarchy that includes a project file, the references to sourcecode will be in the original locations, NOT the copied sourcecode files.

In addition to the instance of the DUT, a testbench contains statements that perform two tasks.
1) Values for the inputs signals must be created and applied to the DUT.
2) The outputs of the DUT must be observed and compared to known correct values.
There should be a reasonable mechanism for communicating these observations to the user.

When values for all inputs are chosen, these values are referred to as a stimulus. Corresponding to given stimuli are the correct values for all outputs. Let us call these the expected response.

Concurrent statements in VHDL are generally either
1) instantiations,
2) signal assignments or
3) VHDL `process` statements.

Instantiations eventually elaborate to leaf-node signal assignments or VHDL processes. After elaboration, each leaf-node statement corresponds to a single simulation process. Notice the terminology: a signal assignment is a simulation process and also a `process` statement is a simulation process.

A testbench may specify multiple concurrent `process` statements to perform various tasks.
- Let's use a one `process` statement to perform all tasks that communicate with the DUT.

Almost all simulation processes begin execution during initialization.
- A VHDL `process` statement with a sensitivity list suspends (sleeps) when it reaches the end.
- A VHDL `process` without a sensitivity list is an infinite loop. As such VHDL `process`es without sensitivity lists must contain `wait` statements. (LRM-2008 §10.2, page 145.)

`Wait` statements cause a `process` to suspend (sleep) and also specify event conditions that resume (wake) the `process`.

5) Add a VHDL `process` that assigns "unknown" to all input bits, sleeps for PREPTIME, then assigns all '0's to the input bits and then sleeps for MEASTIME.
- Label the process MAIN:
- Declare the PREPTIME and MEASTIME as `constants` of 40 picoseconds and 200 picoseconds respectively. (*The default resolution is picoseconds*.)

How do you assign all '0's to a std_logic_vector using "`others`"?
(LRM-2008 §9.3.3.3, page 133)

6) Compile and then Start a simulation of the testbench.
- Notice that the compiler created your entities in the work library. (*see the library pane*.)
- Notice that the simulator loaded (elaborated) the entities. (*see the transcript pane*.)

Try this experiment: Remove the wait statements to form an infinite loop. Determine whether initialization occurs when the model is loaded or whether it occurs when the simulation is run.

OddParity Verification (continued):

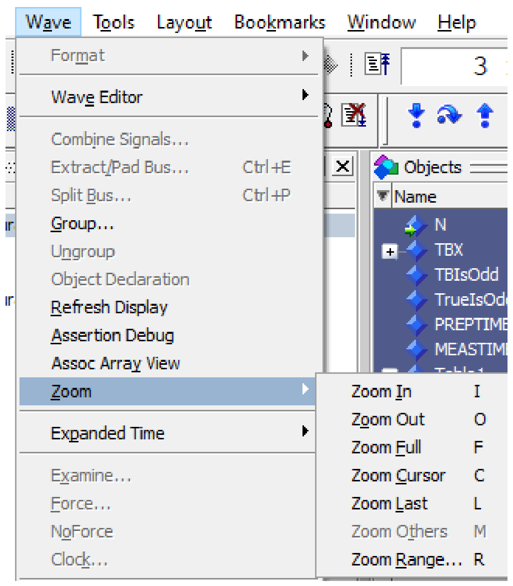
At this point we have coded a very simple stimuli engine. Now we want to accomplish two tasks.

- Setup ModelSim/Questa to display simulation waveforms.
- Make the testbench check to see if the output is correct.

7) Setting up the Wave Window. The waveforms should be displayed for the convenience of the person viewing the results. (It is very rude to neglect the viewers feelings.)

- Assign all signals connected to the DUT to the wave window. (*Internal Testbench Signals*)
- Learn how to set the row height, display radix, etc
- Learn how to insert dividers between collections of related signals.
- Set the row height to 45 and display the input vector in Hexadecimal.
- Insert a divider named "DUT Signals" above the actual DUT signals.
- Choose Useful Names to display on the left of the signal waveforms.

Activate the wave window. Now look at the menu item Wave->Zoom. You can easily zoom in and out by hovering the mouse over the wave window and then typing the letters shown.



OddParity Verification (continued):

8) Using Scripts to Efficiently Conduct Repetitive Tasks.

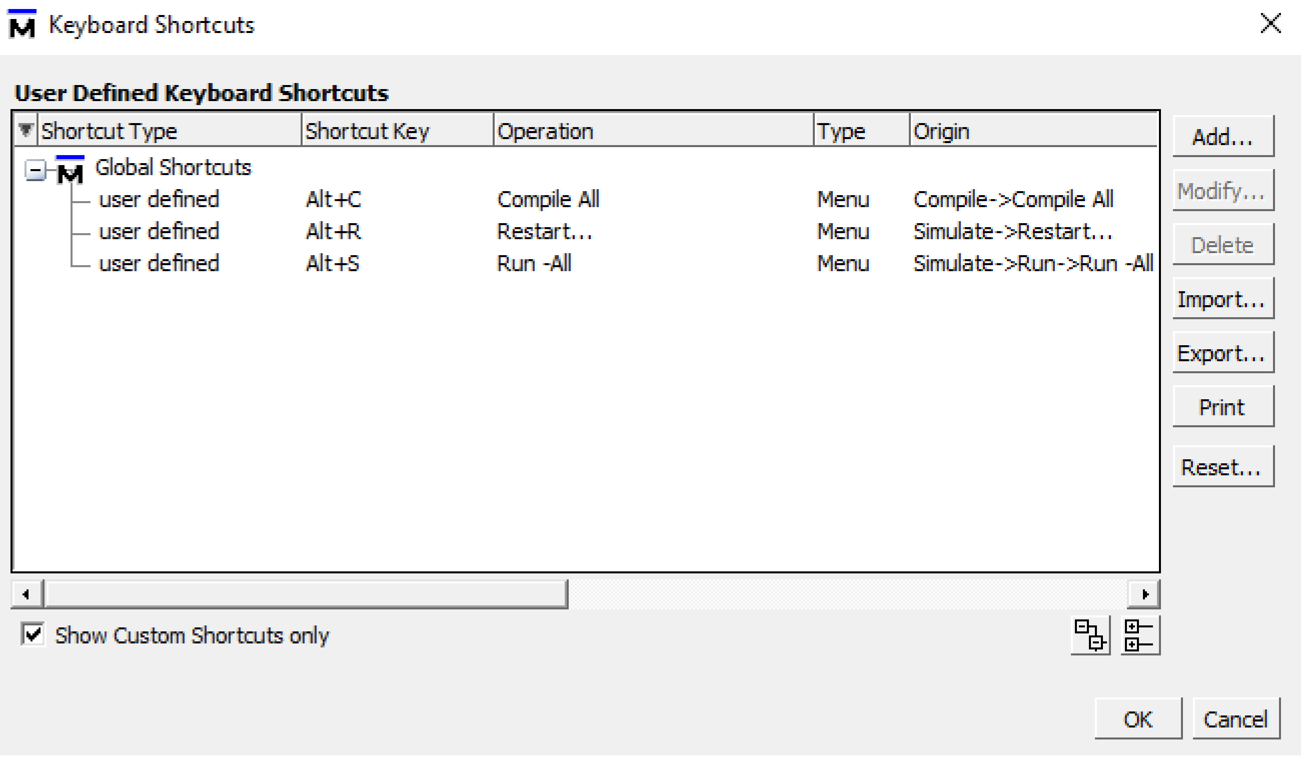
- Activate the Wave window, then use the menu File->save format (ctrl-s).
  Using a text editor, look at the file.

The file "Wave.do" contains the commands that are needed to create your window.
You can execute a script by typing in the transcript window "do scriptfilename.do"
Once you have decided how best to display the waveforms, you should save a script that creates the wave window. Whenever you make changes to the wave window you should update the script.

- To initially remove all objects attached to the wave window, insert the command
  `delete wave *`

You can also efficiently re-compile code and restart a simulation using keyboard shortcuts.

- Look at the menu item, Window->Keyboard Shortcuts …
- Make your own shortcuts as you find yourself repetitively selecting the same menu items.
- Refer to page – 7 of "LWS-01.x-SW-Install-Setup-350-xxxx"



Shortcut Type	Shortcut Key	Operation	Type	Origin
Global Shortcuts				
user defined	Alt+C	Compile All	Menu	Compile->Compile All
user defined	Alt+R	Restart…	Menu	Simulate->Restart…
user defined	Alt+S	Run -All	Menu	Simulate->Run->Run -All

9) Run a simulation for enough time that you are able to verify that the stimuli are correct and that the constants, PREPTIME and MEASTIME are correct.

OddParity Verification (continued):

Similar to a `struct` in C, VHDL allows data objects to be aggregated into a Record. It seems reasonable to collect stimulus/expected response pairs together in a record.

10) Declare a `record` type, used to hold the input and output pairs for the OddParity circuit.
- Call this datatype "TestVectorOP". (LRM-2008 §5.3.3, page 51.)

11) Creating a table of values for verifying the DUT. (LRM-2008 §5.3.2, page 44.)
- Declare an `array` type called "TestDataTable" containing records of type TestVectorOP.
- Declare that the array contains 5 elements. (indexed from 1 to 5)
- Create an instance called Table1. Make this instance a constant and choose values to be placed in the table. (*Your choices are called Test Vectors.*)

To create std_logic_vectors from integer constants, use the conversion functions,
`std_logic_vector( to_unsigned(` someIntegerConstant, N))

12) Now modify the main loop so that it sequences through the 5 measurements. There are both `for` loops and `while` loops in VHDL. When using `for` loops it is not easy to see the loop control variable in the wave window.
- Declare an integer variable called MeasurementIndex, initialized to 1. This is the time to learn the difference between `variables` and `signals` in VHDL.
- Enclose your previous statements that applied the stimuli using a `while loop` that sweeps through the table. (LRM-2008 §10.10, page 166.) Notice that the code is starting to look like a computer program. This region of code does not necessarily correspond to a circuit.
- Alter the statement that applies the measurement value so that it reads the value from the table.
- Increment the MeasurementIndex as the last statement in the loop.
- Precede the loop with a statement that explicitly initializes `MeasurementIndex := 1;` This is necessary since if you bypass the loop you will never hit a wait statement and get an infinitely looping process.
- Alternatively, you could add a wait statement after the loop.

13) Experiment using different combinations of
- using a `wait` outside the loop, at the end of the process,
- using a variable assignment statement to initialize the variable, MeasurementIndex, and
- initializing the variable, MeasurementIndex in it's declaration.

Each time, run a simulation for at least 3 ns to verify that you can see the correct stimuli being applied to the DUT.

OddParity Verification (continued):

Now that we are able to conveniently apply stimuli to the DUT, it is time to focus our attention on the output.

14) Add a signal to the wave window that shows the expected response.
Use the convention:
All signals in the testbench containing expected responses have the prefix, **`TBtrue`**.

- Within the measurement loop, assign the expected output value to a signal called TBtrueIsOdd. The expected output should be stored in each record of the array, Table1.
- The TBtrueIsOdd signal is simply for convenience to see the values in the wave window. Add TrueIsOdd to the wave window, between the input stimuli and the actual DUT output.
- Add the MeasurementIndex to the top of the wave window. (you may need to activate the locals window and then select the MAIN process to see the variable.)
- Format the displayed waveforms to be considerate to the person viewing and then update the wave.do script.
- Update the script wave.do

VHDL contains a facility for sending messages to the transcript and deciding whether to halt execution of a simulation run. This feature is implemented using the keywords, `assert`, `report`, and `severity`. (LRM-2008 §10.3, page 147.)

We have introduced the interval PREPTIME to conveniently begin each measurement from a stable state. The DUT may have some propagation delay and thus we should wait until the DUT output is stable before beginning the next measurement.

15) Place an assertion that executes when the measurement is to end. This should occur MEASTIME after a stimulus is applied to the DUT.

- Insert an assertion that compares the expected result to the DUT output.
- You should choose a reasonable message to send to the transcript.
- Severity levels are typically, NOTE, WARNING, ERROR, FAILURE or FATAL. ModelSim/ Questa defaults to halting execution on ERROR.
  (inspect the settings using the menu "simulate->runtime options…"
- Check that the assertion works correctly by temporarily placing some bad expected output values into Table1.

It would be useful to provide numerical information within the reported message. Briefly look at examples using the string concatenation operator and the 'image conversion function.

16) Modify your `report` statement to include the measurement index, the input stimulus and the actual output within the reported message string.

OddParity Verification (continued):

17) Modify your code for the primitive 2-input XOR created in the previous assignment.
Model a propagation delay of 20 ps and contamination delay of 15 ps, by adding a delay-mechanism to the signal assignment. (LRM-2008 §11.6, page 174.)
(*Delay mechanisms are used by simulators but are ignored by synthesisers*.)

```
y <= 'X' after 15 ps, a xor b after 20 ps;
```

- Observe the difference between the chain and the tree architectures.
- Observe the effect of changing the PREPTIME stimuli from ‘X’s to ‘0’s.
- Observe the effect of temporarily extending the MEASTIME from 200 ps to 400 ps.

Make sure that you
1) can observe the difference between chain and tree outputs, and
2) completely understand the need to change the PREPTIME stimulus and MEASTIME interval.

There are other methods for detecting bad responses from the DUT.
- We could create an independent process that receives the same stimuli as the DUT. This process would calculate the expected results in a manner similar to a computer algorithm. This VHDL code would be a high-level behavioural model. Not necessarily synthesisable.
- Alternatively we could use a programing language like C++ , Python or MATLAB to construct the stimuli/expected response pairs (Test Vectors). The computational results would be stored in a file, that is simply referred to as the Test Vectors. The VHDL testbench could read values from this file, similar to reading values from Table1.
- We will investigate these methods later.

Our final task will be to learn as little bit more about using scripts. Using a mouse to select menus and toolbar elements is inefficient when you are running through the same sequence of operations many times. (it’s also bad for your body)

Each time you perform some action within ModelSim/Questa/Questa, you can see the equivalent command in the transcript window. It is a simple matter to copy the command from the transcript window and then paste the command into a script file. In some cases you may wish to modify some of the command arguments.

To execute a script: at the command prompt in the transcript window, type “do ScriptFileName.do”
(*The up/down arrow keys can be used to scroll through the history buffer*.)

18) Conclude this part of the assignment by constructing a script that
- Closes the simulator. (just in case its already open)
- Compiles all files in the simulation project.
- Starts the simulator
- Sets up the wave window – by running the script wave.do
- Runs the simulation

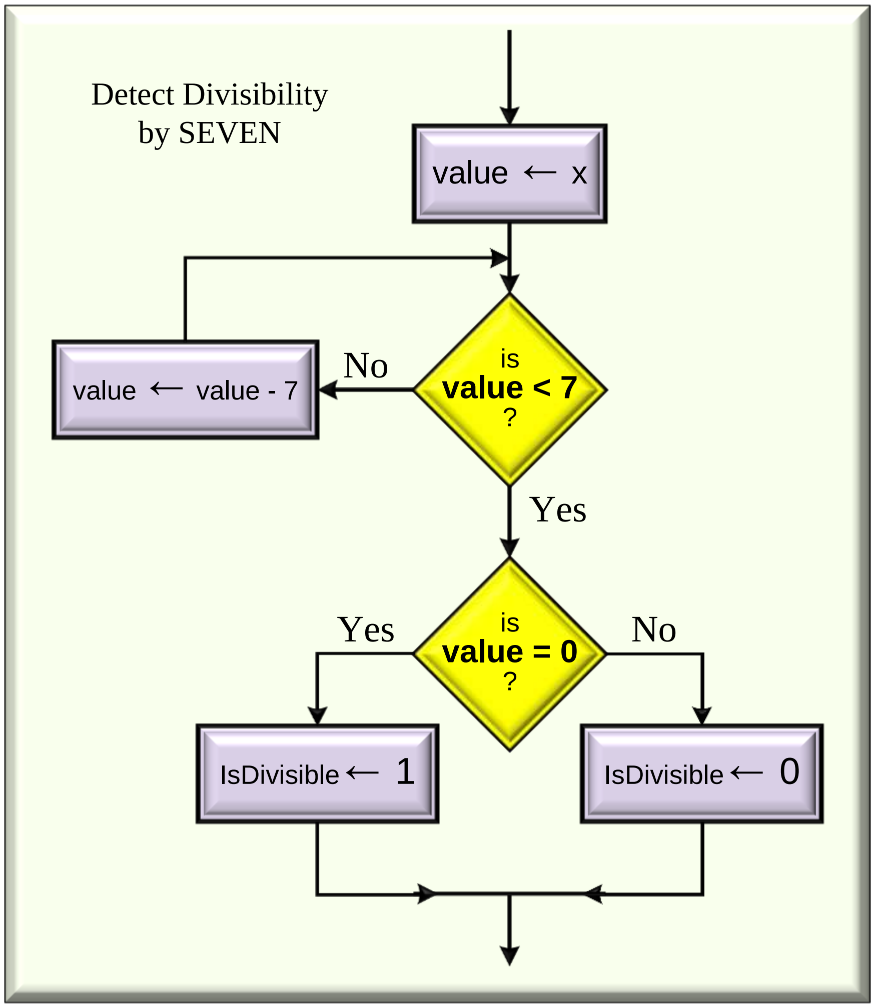
Synthesizable vs. Non-Synthesizable Code: **Part 2:**

Div7 - Verification:

1) We wish to implement a combinational circuit that has 18-bits at it’s input, called x, and a single 1-bit output called IsDivisible. Declare a new **`Entity`** called Div7 in a separate file.

```
Entity Div7 is
        Generic ( N : natural := 18 );
        Port ( x : in unsigned( N-1 downto 0 );
                IsDivisible : out std_logic );
End Entity Div7;
```

2) Create an **`Architecture`** called behavioural, using the following algorithm.
You should take this opportunity to enhance your knowledge of the difference between **`variables`** and **`signals`**.
Notice that you could also eliminate the loop using the VHDL modulus operator.



Div7 - Verification (continued):

3) Create a testbench called TBDiv7.
   Use your testbench from the previous part as a starting template.
   Modify the code appropriately so that
   - there are ten measurements using the input values,
     { 0, 20, 196609, 100, 70, 12345, 38572, 65541, 2344, 43535 } and
   - an instance of your behavioural entity is the DUT.

4) Run the testbench and convince yourself that the model functions correctly.
   - You should experiment with other input stimuli.
   - Try running the loop using consecutive integer values.
   - Notice what happens when the signal X is displayed in decimal, or unsigned. Consider the virtue in displaying the signal in octal.
   - If you see a warning "metavalue detected, returning 0", determine the cause and experiment with ways to remove the warning.

5) Now modify the loop body so that the input stimulus is a random 18-bit positive integer.
   Manually, convince yourself that the result is correct. A quick google search will provide many examples describing how to use the function uniform().

6) Now comes the real hardware engineering.
   Create an `architecture` called **structural**, for the entity Div7.
   Your circuit should be implemented to meet the following constraints.
   The final circuit should;
   1) be a purely combinational circuit,
   2) synthesize correctly using Quartus's synthesizer.
   3) be optimized to use minimal LUTs when built using a Cyclone-IVE FPGA.

   Verify these properties using the Quartus RTL viewers and reports.
   - The final circuit should have less than 40 LEs and be as fast as possible.
   - You should verify the speed when performing the tasks for LWS04.

   HINT: Notice that if a number is divisible by NINE, then the sum of it's decimal digits is also divisible by NINE.

7) We can verify that the structural circuit is correct by comparing it's output to the output of the behavioural model. Of course, we are assuming that the behavioural model is correct.
   Modify the testbench to use the behavioural model as an intent model and automate the verification procedure.
   - Instantiate the structural model as the DUT.
   - Instantiate the behavioural model as an entity called Intent. *This object is called an intent model.*
   - Apply the 100 random input stimuli to both entities and use an assertion that compares both outputs.
   - Report meaningful errors when the two outputs are not the same. Temporarily create errors the check that the error check works correctly.

Lab Worksheet 03

Task ID	Show to TA during Lab Period for feedback and Lab Tick	Demonstration Required Signup for a demo time on Canvas	Documentation Required Canvas Submission	Documentation Required for feedback and Lab Tick
LWS-03-P1 (01-09)	☒	☒	☒	☒
LWS-03-P1 (10-18)	☑	☒	☒	☒
LWS-03-P2 (01-05)	☑	☒	☒	☒
LWS-03-P2 (06-07)	☑	☒	☒	☒