ETH zürich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

VLSI I:
From Architectures to VLSI Circuits and FPGAs

227-0116-00L

Exercise 4

Testbenches & Verification

Prof. L. Benini
F. Gürkaynak
M. Korb

Last Changed: 2023-10-23 14:59:12 +0200

1 Introduction

In this exercise, you will learn the fundamentals of verification and creation of testbenches. Testing and verification is arguably the most time consuming stage in a typical VLSI project life cycle. A successful verification effort can save a fortune as it uncovers all possible bugs in the design *before* product deployment. A tremendous amount of internal and external costs are associated with respinning a hardware project. It is therefore important to know the art of verification to complete the projects on time. While there are several different aspects to verification, in this exercise we will focus on 'functional verification' which ensures the logic behavior of the circuit works as per the specification.

This exercise spans for two afternoons. We will guide you through the following tasks during these two sessions:

1. Getting to know the anatomy of a generic testbench infrastructure.
2. Writing scripts for simulation with ModelSim - an advanced HDL simulator.
3. Debugging a design with ModelSim.
4. Different approaches to functional verification, pros and cons of each.
5. Developing components of a testbench using SystemVerilog and MATLAB.
6. Making use of SystemVerilog's assertions.

As always the assistants are at your disposal to help you with the task and discuss solutions.

2 Preparation

To get access to materials required for the exercise please follow the task below:

Student Task 1 (Setup): Setup the working directory for this exercise by calling our install script and changing to the newly created directory:

```
sh> /home/vlsi1/ex4/install.sh
sh> cd ex4
```

3 Anatomy of a Testbench

The environment created for verification of a Design Under Test (DUT) is called 'testbench'. A typical structure of a basic testbench is schematized in Figure 1. The DUT is the digital circuit that is written by the design engineer using one of the Hardware Description Languages (HDL) (e.g., VHDL, Verilog). As shown in the figure, the DUT is instantiated in the testbench along with other testbench components. Most or all of the testbench components are written using a HDL so that these can interact with the DUT. As illustrated in Figure 1, some of the testbench components can be described using another procedural programming language such as MATLAB, Python, or C/C++, giving flexibility to write complex functional specification, arguably in a shorter period of time. Meaning *all* testbench components (except for the DUT) can be *non-synthesizable* to optimize simulation runtime.

The following testbench components are visible in Figure 1.

1. `Stimuli Generator`:
   This module takes care of stimuli generation for the given testcase. The output of `Stimuli Generator` is a sequence of simulation vectors which is fed to the `Application Driver` on the input side interface of the DUT.
2. `Golden Model`:
   This module contains an accurate representation of the DUT functional specification.

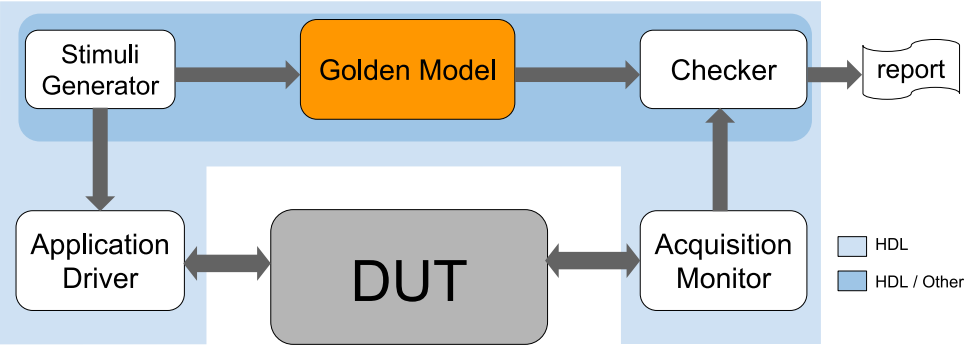

Figure 1: Generic Testbench Architecture

3. `Application Driver`:
   This module is responsible for feeding the simulation vectors to the DUT. It takes care of passing the data in compliance to the specified protocol which can sometimes consist of complex transactions lasting multiple cycles and requiring internal states. In cases where the interface protocol is simpler, it is also possible to merge the `Application Driver`'s functionality with the `Stimuli Generator`.
4. `Acquisition Monitor`:
   Monitoring output responses and controlling externally-driven signals on the DUT's output interface is the job of the `Acquisition Monitor`. It also collects the output vectors from the DUT and sends them to the `Checker` module.
5. `Checker`:
   The `Checker` module compares buffered reference output vectors from the `Golden Model` against the actual responses from the `Acquisition Monitor`. It will report any mismatches in the output.

We will first have a look at a basic *Gray counter* example design to learn the basic tools and aspects of verification. Later, we will look at an advanced image filter design to get in touch with advanced verification techniques such as constrained random verification. [1]

4 Introductory Example

In this first assignment we will be working with a *Gray counter*. As you may recall, Gray code is a technique used to order unique binary bit patterns such that any two consecutive bit patterns will have only **one** bit position different. To get you started with the assignment, an implementation of a Gray counter developed by one of your colleagues [2] is provided for you in **`1_introductory/sourcecode`** directory. However, the code is not yet tested and your task in this assignment is to find a way to successfully test the design, and find bugs if there are any.

Student Task 2 (Open Source Files): Open **`gray_counter.sv`** in **`sourcecode`** directory using a text editor of your choice (we recommend using `sublime_text` editor) and familiarize yourself with the code. ***Hint**: Sublime Text allows opening whole folders instead of only single files, and will allow you to browse the files in its side bar. If you are in the `ex4` folder, you can open the folder for the first exercise with the following command:*

```
sh > sublime_text 1_introductory
```

`gray_counter` has two inputs and one output. In a nutshell, the specification of this DUT is to output 8-bit Gray codes on `gray_count_o` port in every clock cycle (the clock is fed via `clk_i` port) when the reset phase is inactive (when `rst_ni` input is HIGH).

[1] For a standardized verification methodology, you might want to check Universal Verification Methodology.
[2] Take it with a grain of salt ;)

A testbench code (which is under development) to test the Gray counter is provided in **1_introductory/sourcecode/tb** directory. In the **tb** directory you will be able to find three different testbench modules: **gray_count_tb.sv**, **clk_rst_gen.sv** and **rst_gen.sv**.
The main testbench module, which is **gray_count_tb**, is incomplete and your help is required to make it complete.

Student Task 3 (Open Testbench Files): Open **gray_count_tb.sv** in **sourcecode/tb** directory using the text editor and familiarize yourself with the code.

In `gray_count_tb` testbench you will be able to recognize code blocks marked by:

```
initial begin : <block identifier>
...
end
```

These code blocks have matching components in the generic testbench architecture presented in Figure 1. Thus, they are dedicated to describe the function of each of the related component.

The `gray_counter` module which is the subject of this assignment turns out to be a special case which does not require a `Stimuli Generator`. Apart from clock and reset, no input stimuli is specified. For this reason, the code does not contain a `Stimuli Generator`. Furthermore, since the application of clock and reset signals is managed by subordinate module, the code block related to `Application Driver` is also left empty. The code block related to `Acquisition Monitor` describes how the output response is repeatedly captured in a register element in every clock cycle after the reset phase, while marking this event by raising the `acq_valid` flag. We will discuss the `Golden Model` and `Checker` blocks later in this exercise.

The first lines of the `gray_count_tb` module are

```
timeunit 1ns;
timeprecision 1ps;
```

The `timeunit` directive specifies the time unit used during simulation. That is, whenever you encounter a time delay statement with a unit-less number, e.g.,

```
#3;
```

simulation at this point is delayed by the number times the time unit.

The `timeprecision` directive specifies the minimum time step of the simulation. No delay can be smaller than the minimum time step. Delays that fall outside the steps specified by `timeprecision` are rounded to the nearest step. For example

```
#3.0009ns;
```

would be rounded to a delay of $3.001\,\mathrm{ns}$ when `timeprecision` is $1\,\mathrm{ps}$.

Having understood the source files, let us try to simulate the design. For this assignment we will be using ***ModelSim*** simulator, which is one of the most powerful and fastest HDL simulators[3] with many advanced simulation features, some of which you will learn during this exercise.

As the first step, we need to compile the DUT files and the testbench files. During the compilation process all source files are compiled into a '*library*' for which a directory will be created. Even though it is possible to perform the next steps using a graphical user interface (GUI), we will teach you how they could be done using a reusable script so that the process could be automated and reusable in other designs as well.

Student Task 4 (Completing the Compile Script):

1. Open the compile script **compile.tcl** in **1_introductory/modelsim** directory using the text editor.

[3] Opinions may vary. The original author was not payed to write that.

2. Notice how a variable `LIB` for the name of the library is set as `work`. A directory is then created using `vlib` command (N.B., prior to that it will be cleaned if the directory already exists).
3. Below that, there are two `vlog` commands with placeholders to compile design files and testbench files separately. Update the placeholders in both the commands with the relative path to the design file list and the testbench file list respectively.
   - path to the design file list: **`../sourcecode/file.list`**
   - path to the testbench file list: **`../sourcecode/tb/file.list`**
4. Have a look at both file lists to understand how file names are specified using a relative path with respect to the compile script directory and multiple file paths are separated by newlines.

The `vlog` commands in the script carry several arguments with different purposes. For example, the `sv` argument indicates the files are written in SystemVerilog. The `work` argument specifies the library where compiled simulation models should be stored. The `f` argument specifies design files by means of a file list. Thereby, when working on another design with the same directory structure, it is possible to use the same compile script, and only update the file list. Furthermore, the file lists could be shared with other tools such as Vivado and Synopsys (a tool for IC design), such that the design is consistent across EDA tools. We have also used the `pedanticerrors` switch in the **`runsim.tcl`** script to enforce more strict compilation rules to make sure the design is clean. You are encouraged to have a look at the *ModelSim reference manual* to get to know more about all the commands used in this exercise. The document is available in the following path: **`/usr/pack/questa-2019.3-kgf/questasim/docs/pdfdocs/questa_sim_ref.pdf`**

The top testbench is incomplete and it needs to have the instantiation of the DUT.

Student Task 5 (Instantiating the DUT in the testbench):

1. Open the top testbench located in **`sourcecode/tb/gray_count_tb.sv`** using the text editor.
2. Add the instantiation code snippet to match the port list in **`sourcecode/gray_counter.sv`** file. **Use *'dut'* as the name of the instance**

The simulation script is now complete and the testbench is ready for the first compilation.

Student Task 6 (Compiling The Design):
In a terminal, navigate to the **`modelsim`** directory and execute the following command:

```
sh>  questa-2019.3 vsim -do compile.tcl
```

Here, we use the latest ModelSim version available, which is 2019.3. If the script worked as expected, ModelSim GUI should open with no errors in the *'Transcript'* window. If you see errors or something else is wrong, contact an assistant.

The next step is to complete and launch the simulation script.

Student Task 7 (Completing And Launching The Simulation Script):

1. Open **`runsim.tcl`** using the text editor and have a look at its content.
2. Specify the name of the top entity (i.e., the name of the top level testbench) in the placeholder provided.
3. Notice the use of arguments to enable debug features (`-voptargs='+acc'`) and specify the working library where the compiled models are stored. More information about the `vsim` command can be obtained from the reference manual.
4. Now launch the simulation script. In the ModelSim command prompt type:

   ```
   QuestaSim>  do runsim.tcl
   ```

Make sure there are no issues in the simulation steps. If everything works fine, the simulation should pass and stop at the `stop()` command in the testbench at $13\,360\,\mathrm{ns}$ time (time is shown at the left-bottom corner of th main ModelSim window). However, since our testbench does not yet check the output, the *PASSED* does not really say something about the design.

Now that the simulation is running, let's inspect if the output matches the Gray code specification.

Student Task 8 (Manual Inspection Of The Output):

1. Select the waveform window, which includes all internal signals of the DUT.
   (The `add wave -r /dut/*` command is responsible for automatically adding all the signals into the waveform window. If you did not name the instance of the Gray counter *dut*, this command will fail to add the waves to the waveform window. Signals can also be added from the GUI. For that, select the block in which the signal is present from the `Instance` window → select and right click the required signal in the `Objects` window and → Select `Add wave` or press `ctrl+w`.)
2. Find the first erroneous output on `gray_count_o` port. After how many correct outputs does the first error occur? (You may use the hints below.)

   ***Hint**: `gray_count_o` values are displayed in hexadecimal format. It is easy to find the error by visualizing the values in binary form, because an error can be spotted when the number of bits changing between consecutive values is not equal to 1. To convert the format, right click on the signal in the left column in wave window (shown by "H" in Figure 2). → `Radix` → `Binary`*
   ***Hint**: Last bits in `gray_counter_o` may not be visible in some intervals, because of the compactness of the time axis. To zoom in and out on the time axis use icons shown by "D" in Figure 2. Alternatively, you can use the middle mouse button to zoom into a specific time interval by clicking and dragging, or you can use shortcuts. When the waveform window is the active window use:*
   *`f` to **f**it the time axis to the whole simulation span.*
   *`o` to zoom **o**ut and `i` to zoom **i**n.*
   *`c` to zoom in on the location of the active **c**ursor.*
   ***Hint**: The values displayed in "G" region correspond to the time where the cursor shown by "E" sits in the time axis. Therefore, it is possible to more clearly see the value changing in a fixed position by dragging cursor. However instead of manually moving the cursor, the cursor could be hopped to the next value change for a selected signal. For that, select the signal in "H" region and click on icons in "C" region in Figure 2 .*
   ***Hint**: Finally, once the first error is located, counting the number of preceding correct outputs could be a time consuming and error-prone task. It is possible, however, to measure the time between first output and the first error by placing multiple cursors. To place a second cursor click on icons in "B" region. After placing the two cursors at the required locations, the time between the cursors appears in "F" region. It is also possible to lock cursors to avoid accidentally moving again. For that use icons in the "I" region.*

Manually inspecting errors of a simulation is indeed a cumbersome effort as you may have experienced in the previous task. Let's assume we had a design that performs the basic functionality and bugs were introduced later on after adding an enhancement feature. Or, the same design, but at two different stages in the life cycle (e.g., behavioral and synthesis netlist). ModelSim provides the feature to compare waveforms from different versions of a design. In this particular case, we have saved the simulation database of a working `gray_counter` module in **`gold.wlf`** file in the **`modelsim`** directory.

Student Task 9 (Comparing Waveforms):

1. Click on `Tools` → `Waveform Compare` → `Comparison Wizard`
2. In the Comparison Wizard window provide **`gold.wlf`** as the *Reference Dataset*. For Test Dataset select *Use current simulation*. Click *Next*.
3. In the next page, select *Compare All Signals*. Click *Next*.
4. In the next page, select *No* to *Would you like to add more signals to the comparison?*

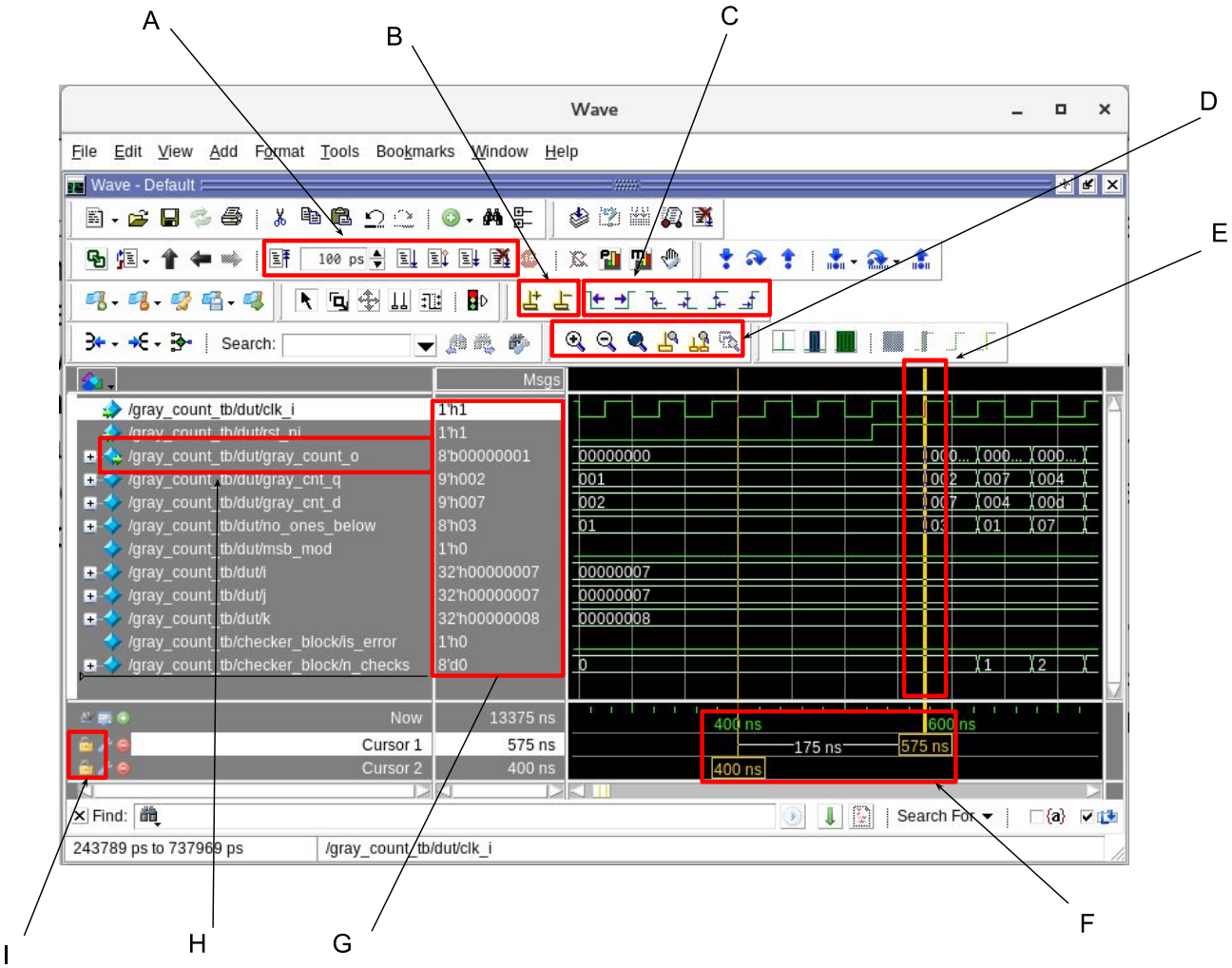

Figure 2: ModelSim wave window

5. In the last page, click *Compute Differences Now*.
6. Now check the wave window for new signals. You may be able to see mismatching regions in red. Signals which have at least a single mismatch are indicated by a red cross in the "H" region in Figure 2; a "diff" label will appear in the "G" region if, at the cursor's position, there is a mismatch.

We will now try to locate the error in the source code using the waveform comparison tool.

Student Task 10 (Locating The Error Using Waveform Compare):

1. Slide the cursor until the first mismatch is found. Which signal(s) exhibit the earliest mismatches?

   ---

2. Click on the "+" sign on the left side of the signal(s) to expand and see the waves from both versions. Clicking "+" again on the signal from each version reveals bit level variations.
3. Which bit position(s) of the error signal(s) identified above have mismatches at the time of the first mismatch?

   ---

4. Can you localize the bug in the DUT to a single line? Which one is the erroneous line?

***Hint**: In ModelSim it is easy to find lines related to drivers of a signal. Right-click on the erroneous bit/wave in the Wave Window (you can click at the time that the first error occurs) and select 'Event Traceback (@ X ns)' → 'Show Driver'.*
Alternatively, you can right-click on the signal in the 'Wave Window' relating to the current simulation (with prefix "sim:") → 'Object Declaration' → this will open a text editor (check the task bar if the window is in background) with the cursor located at the line in which the signal is declared → Right click on the signal name on the text editor → 'Event Traceback' → 'Show Driver'. This should ideally highlight line(s) that have drivers for the selected signal.
If the 'Event Traceback' or 'Show Driver' option does not appear or work, end the comparison (Click on `Tools → Waveform Compare → End Comparison`) or restart the simulation (`restart -f\ ; run -a`). If this does not work, try to recompile and restart the simulation and try again without repeating the waveform comparison (`do compile.tcl; do runsim.tcl`).
***Hint**: The error originates from the line of code where the concerned bit position of the interested signal is driven exclusively by specifically selecting that bit as the target.*

Student Task 11 (Fixing The Error):

1. Apply the most appropriate fix on the erroneous line of the code. (You may use the hint below.)
   ***Hint**: In the incorrect version of the design, it was revealed that the logic that is in the erroneous line of code has one input coming from a different bit position of the registered version of the signal. This should be corrected as the same bit position of the registered version of the signal.*
2. Repeat the above tasks 6, 7, 9 to verify that your fix is correct. You do not need to close ModelSim to recompile the design, just rerun the compile script from the ModelSim command line.

```
QuestaSim> do compile.tcl
```

As explained earlier, waveform comparison tool can be used to compare all available internal signals, but it is helpful only if there exists a simulation database of a correctly working version of the design. A self-checking mechanism becomes useful in all other cases. In fact, developing a self-checking testbench is essential to speed up the verification effort.

We will now attempt to write a piece of code that provides self-checking functionality to our existing testbench. A **Golden Model** is usually an important part of a self-checking testbench. However, the specification of our current design is not complete enough to generate a golden model which can determine each and every response of the DUT. Instead of developing a golden model, we will write a SystemVerilog task which checks whether the responses in the sequence match the Gray code specification.

Student Task 12 (Adding self-checking testbench features):

1. Write the content of the `next_gray_check` task such that it updates the `is_error` signal to value 0 only if the current Gray code (`acq_gray_count`) has exactly one bit position different from the previous Gray code value captured (`acq_gray_count_q1`).
   ***Hint**: For this purpose you may use **for** loop constructs, **if** constructs and additional variables local to the task. Refer to lecture notes to find references to coding constructs.*
2. Call `next_gray_check` task in the `checker_block` module within the space provided. Increment the error count `n_errs` variable for each erroneous output.
3. Rerun the simulation again (repeat tasks 6, 7 above or simply run `do compile.tcl; do runsim\ .tcl`) with the **`gray_counter.sv`** code ***before*** fixes. If a window pops up warning about the sourcecode change, please select 'Reload' option. How many errors are you able to capture during the whole simulation?

5 Assertions

Our design now passes the testbench, and we are (rather) confident that it is working correctly. However, to make everyone's life easier when using this Gray counter in a large system, we will also add an assertion. Specifically, we will add an assertion making sure that at every clock cycle, exactly one bit changes. The main purpose of this assertion is to check the correct behavior of our design. If we missed a bug while testing the design with our testbench, and our module misbehaves when inserted into a much larger design, this assertion will hopefully catch the invalid output and tell the designer that something is wrong with the Gray counter. This can save a lot of debugging time as the designer would otherwise search for the problem in their own module or whatever they changed last in the design. Furthermore, an assertion can communicate intent. It will immediately tell everyone looking at the design that the output is only supposed to change one bit at a time.

Student Task 13 (Adding Assertions):

1. Add an assertion to the `gray_counter` module, asserting that exactly one bit of the output changes each clock cycle.
   ***Hint**: In the appendix of this exercise you can find a cheat sheet briefly explaining the syntax and the most important features of assertions.*
   ***Hint**: You may use SystemVerilog's `$onehot(expression)` function which returns true only if exactly one bit in the expression is one.*
   ***Hint**: Use the **`xor`** operation, which is true if two bits are different, and false if two bits are the same.*
2. Test that your assertion works correctly by rerunning the simulation **with** and **without** the errors in the design.

$\mathcal{E}$ **You have reached the end of the first afternoon.**
Please discuss your answers and any open questions with an assistant. $\mathcal{E}$

In our previous assignment we looked at the debugging aspects and developed self-checking testbenches for a simple design. In this assignment we will look at a more advanced problem, which is a `rgb_saturator`, to learn about the missing pieces of the generic testbench architecture presented in Figure 1. In particular, we will look into different approaches that can be used in input stimuli generation and constructing a golden model for a given problem. By the end of the exercise you will also learn some useful techniques in improving the test coverage for a given number of simulation cycles.

6 File-based Verification

In this assignment we will develop a file-based testbench infrastructure to test a `rgb_saturator`. The functionality of the DUT is such that it applies point-wise transformation functions on each of the R, G and B channels of the pixels in an image to get a saturated look on the output image. The operations are mathematically represented in equation 1.

$$\begin{aligned}
\texttt{sat_rpixel} &= clip(((\texttt{inp_rpixel} - \texttt{gray_pixel}) * \texttt{SATURATE_SCALE} + \texttt{gray_pixel}), 0, 255) \\
\texttt{sat_gpixel} &= clip(((\texttt{inp_gpixel} - \texttt{gray_pixel}) * \texttt{SATURATE_SCALE} + \texttt{gray_pixel}), 0, 255) \\
\texttt{sat_bpixel} &= clip(((\texttt{inp_bpixel} - \texttt{gray_pixel}) * \texttt{SATURATE_SCALE} + \texttt{gray_pixel}), 0, 255)
\end{aligned} \tag{1}$$

A high-level view of the file-based testbench infrastructure is given in Figure 3.

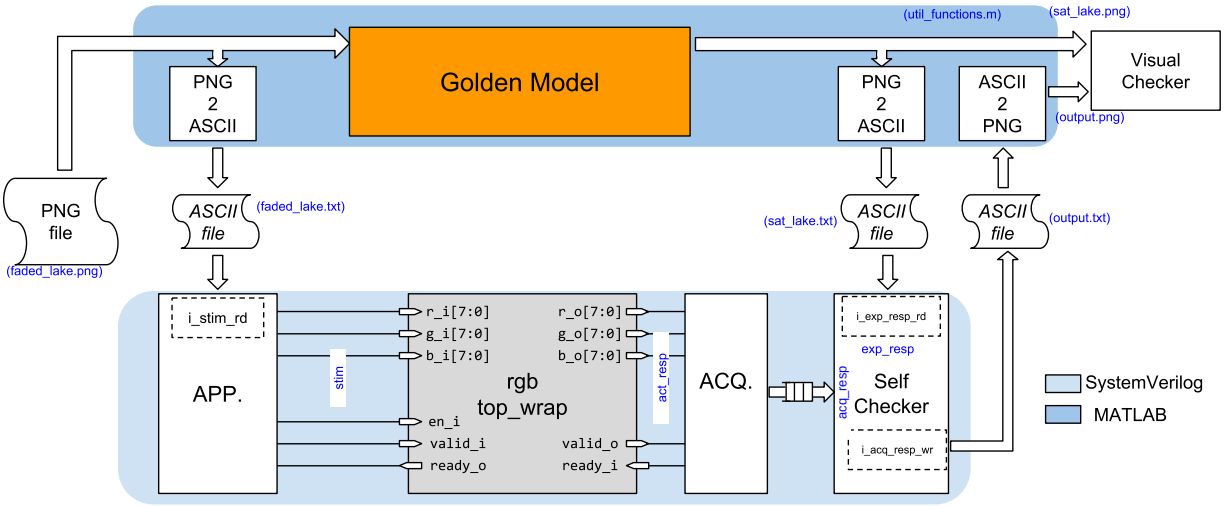

Figure 3: File-based testbench infrastructure for RGB Saturator

As shown in Figure 3, some pieces of the testbench are implemented in MATLAB while the rest is implemented in SystemVerilog and the image data exchanged between the two environments are stored in ASCII files (comma-separated value format). We will first have a look at the MATLAB functions.

Student Task 14 (File preparation):

1. Open MATLAB (type `matlab` in the terminal) from the **`2_advanced/matlab`** directory, have a look at `util_functions.m` and read the descriptions of the functions.
2. Open `script.m` in the same directory and run it. It uses the `util_functions` and input image to create **`faded_lake.txt`** in the **`simvectors`** directory, which will be the stimulus for the SystemVerilog testbench. Furthermore, it will create the golden image **`sat_lake.png`** and the reference values for the checker block, **`sat_lake.txt`**.

We have fulfilled the prerequisites to run the file-based testbench now. A partially completed testbench for this task is provided in `rgb_sat_fb_tb.sv` file in **`2_advanced/sourcecode/tb`**. Open the file and notice the following features in the 'application' block:

1. Using a file descriptor named `stim_fd` the application block accesses **`faded_lake.txt`** you have just created.
2. New stimuli are read from the file and applied to the DUT until the end of the file is reached.
3. The application of `inp_valid` signal is randomized[4].
4. The `$fscanf(fd, format, vars)` function[5] reads the file content line by line and stores the values in `vars` according to the format string. The application block passes this stimulus along with the `inp_valid` and `filt_enable` signal to the DUT.
5. The application of all input signals are delayed by `APPL_DELAY` time units. In reality, inputs to the DUT can arrive with some time delay with respect to the clock due to propagation delays despite them being synchronized to the clock. `APPL_DELAY` tries to model this delay on the input signals.
6. Once applied, `stim` is held until the `inp_ready` signal is sensed as HIGH.

Let's now try to fill the other parts of the testbench to get started with the simulation.

Student Task 15 (Filling missing parts in the main testbench):

1. Instantiate the top level DUT `rgb_top_wrap` within the demarcated area and name it `dut`. The module definition can be found in **`2_advanced/sourcecode`** directory.
2. Complete the function of the `acquisition` block. The following features must be satisfied by the `acquisition` block:
   a) Once the reset period is over, wait for `APPL_DELAY` time units after the positive edge of the clock. Then, randomize the `oup_ready` signal with `randomize(signal)`.
   b) After a further `(ACQ_DELAY-APPL_DELAY)` delay, push the actual response `act_resp` to `queue_mon2chk` queue, when the handshaking is valid, that is, when both `oup_ready` and `oup_valid` are HIGH. Use the `queue_mon2chk.push_back(value)` method for this. SystemVerilog queues are generic and dynamic non-synthesizable data structures that can hold a variable number of elements of a certain data type. It is one of the most useful testbench features to pass data asynchronically between blocks. In this case they pass data from `Acquisition monitor` to `Checker`. Notice how the queue is declared in the testbench.
3. Complete the missing part of the checker block. The following features need to be covered:
   a) Wait until the `queue_mon2chk` queue is non-empty, then pop the value at the other end of the queue and assign it to `acq_resp`. You may use the `size()` and `pop_front()` method on `queue_mon2chk` queue object for this purpose.
   b) Load the next expected value to `exp_resp` by calling `$fscanf(fd, format, vars)` function with the `exp_resp_fd` file descriptor (similar to `stim` in the application block).
   c) Increment `n_checks` and compare `exp_resp` against `acq_resp`. Increment `n_errs` in case of a mismatch, while printing a suitable message regarding the error (expected/actual values and time stamp).
   d) Additionally store the acquired response into **`output.txt`**. Use the `acq_resp_fd` file descriptor with the **`$fwrite`**`(fd, format, vars)` function for this purpose[a]. Note, use the same format that is used for the input stimuli to stay compatible with the MATLAB scripts.

[a] The **`$fwrite`** function does not have a return value

[4] The `randomize` function has a return value of 1 if it was successful or 0 if it was not. If you ignore the return value, you will get a warning from ModelSim that looks something like this: `(vlog-2240) Treating stand-alone use of function 'randomize' as an implicit VOID cast.`

[5] The `$fscanf` function returns the number of variables it decoded (three in our case) or -1 if an error occurred. If you ignore the returned value, ModelSim will likely give you a warning.

It is now all set to launch the simulation in **2_advanced/modelsim_1** directory. Let's check the **compile** and **runsim** scripts.

Student Task 16 (Compiling and launching simulation):

1. Open **compile.tcl** in **modelsim_1** directory. Make sure the switch DEBUG is ON and COVERAGE is OFF. Notice the addition of *-cover* switch in *vlog* command to compile with options required to do a coverage analysis. All the other switches and arguments remain the same as from the gray_counter example.
2. Open a terminal in **modelsim_1** directory and enter the following command:

```
sh>  questa-2019.3 vsim -do compile.tcl -msgmode both
```

   The -msgmode both flag will allow us to see assertion messages in the transcript as well as in the wave window. If you encounter compilation errors, it means there is something wrong with the piece of code you added in the testbench or the way that you updated the scripts. Please fix the errors with the help of an assistant. If compilation works without errors execute the following command in the ModelSim console.

```
 QuestaSim>  do runsim.tcl
```

If everything worked fine, you should be able to observe roughly 56000 mismatches out of 57206 checks between the actual and the golden model output after 85488 stimuli as printed in the *ModelSim* console. Here, 85488 stimuli corresponds to all the pixels in the image of resolution 274x312. If there is a significant inconsistency please consult an assistant. We will now try to visually inspect the output in the form of PNG image and see if we can find problems.

Student Task 17 (Visual checking and comparing the output):

1. Open the script.m again in MATLAB and uncomment the command to create **output.png** using **output.txt** generated by the simulation. Run the script.
2. Compare **output.png** created inside **simvectors** directory against **sat_lake.png**, which is the expected output image. What are your observations? What does the black area in the output image represent?

   _____

3. The pixels in the picture are streamed in the raster scan order (pixels in the frame in row order starting from the top row and pixels within a row from left to right). Can you identify one existing issue in the DUT, given that number of checks (57206) is significantly lower than the number of stimuli provided (85488)? Can you approximately identify where the pixels corresponding to top, middle, and bottom row of the input are mapped to in the output image?

   _____

   _____

   _____

As you would have guessed, the DUT doesn't produce an output pixel for every input pixel it receives. This means there is a problem in the *control path* which includes ready/valid handshake signals. A block diagram of rgb_top_wrap explaining the handshaking is shown in Figure 4. There are two pipeline stages, at the input and at the output of the design, which means that there are three handshaking interfaces with their own ready/valid signals.

The first interface is at the input to rgb_top_wrap, i.e., to the input register. If valid_i and ready_o are valid, the data on the r_i, g_i, b_i, and en_i channels are transmitted and stored in the input register.

The second handshaking interface consists of the `valid`, `ready`, `inp_q`, and `en_q` signals. Specifically, the input register indicates that the input to the `grayscaler` and the `saturator` (and therefore also the output) are valid by setting `valid` to high. The output register sets `ready` to one if it can accept new data, thus fulfilling the handshake. Note, the `en_q` signal is only used as input to the functional blocks and not stored by the output register.

Finally, the output register and the testbench form the third handshaking interface. When the output data `r_o`, `g_o`, and `b_o` are valid, the output register sets `valid_o` to HIGH and the testbench controls the `ready_i` signal.

Each pipeline stage has one bit of state, storing whether the register has valid data or not (`oup/inp_full`). To be able to stream data in every clock cycle, the pipeline stages must be able to accept new data from the upstream source in the same clock cycle as the downstream handshake is successful. Therefore, there is a combinatorial path from the downstream ready signal to the upstream ready signal. For example, `ready` directly depends on `ready_i`.

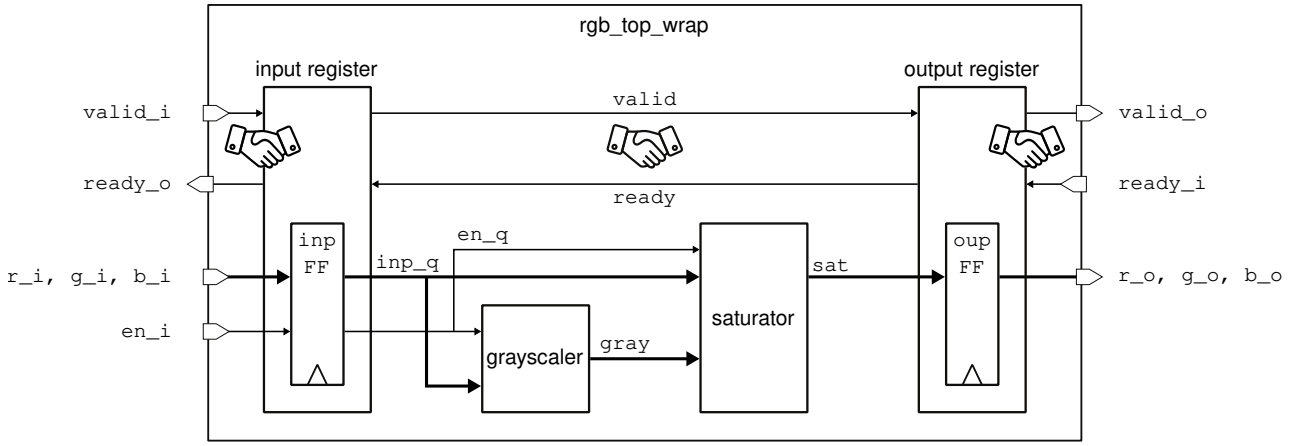

Figure 4: Block diagram of `rgb_top_wrap` with handshaking signals.

Student Task 18 (Insert Assertions):

1. At the end of `rbg_top_wrap`, add assertions for all three handshaking interfaces, to check that the data stays stable during the handshake. Specifically, whenever valid is HIGH but ready is still LOW.
   ***Hint**: You may use the `$stable(expression)` function. It returns true if the expression's value did not change with respect to the previous clock cycle.*
2. Rerun the simulation (`do compile.tcl; do runsim.tcl`).
3. Which handshaking interface violates this assertion? Which register implementation is buggy?

   ---

4. Fix the issue in the register stage and rerun the simulation to verify your fix.
   ***Hint**: If only one handshaking interface is buggy, the other one might give you a hint on how to fix it.*
   ***Hint**: Remember that an `always comb` block should either read or write a variable, but not both. Check if someone mixed up the variable names.*
5. What is the fix for the error found?

   ---

6. *Optionally, repeat Task 17 to visually verify that the output is correct.*

Now we have verified that the design properly works for the provided input image. Does it mean that the design is $100\,\%$ correct? We can not be sure about that because it is possible that some parts of the design are not tested by the given input image. To get a quantifiable idea about the level of coverage of the design achieved

with the given input, we can rerun the simulation with the coverage option on. [6]

Student Task 19 (Rerunning the test with coverage option):

1. Edit the **compile.tcl** script to set COVERAGE variable to ON. Note how COVERAGE changes the **compile.tcl** and **runsim.tcl** scripts. In particular notice that the COV_TYPE variable is set to "bcst", which stands for the types of code coverage: "branch, condition, statement, toggle".
2. Rerun the test by re-executing **compile.tcl** and **runsim.tcl** scripts in ModelSim shell
3. You can now browse the coverage data in ModelSim GUI. A summary report for dut instance is automatically created in **modelsim_1** directory, named **myreport.txt**.

Student Task 20 (Logging simulation statistics):

1. Please enter in the space provided the following figures using the "coverage report by instances" from the above simulation:
   Number of pixels checked, percentage statement coverage, percentage branch coverage, percentage condition coverage, percentage toggle coverage.

   _____

   _____

   _____

   _____

   _____

7 Constrained Random Verification

The file-based testing method that we followed in the previous example is also called *'directed testing'* because we applied a known testcase directly into the DUT without trying to rigorously test for the whole input state space. We will now look at a different verification approach with the intention of increasing the above coverage numbers and uncover more potential bugs in the *same* design. To this end, we will use constrained random verification. A partially completed testbench is provided in **rgb_sat_crv_tb.sv** in **2_advanced/sourcecode/tb** directory. Figure 5 graphically illustrates the composition of the testbench. As shown in the figure, all components are implemented in SystemVerilog, with stimuli generation and application driving being combined into a single module called random stream master.

Student Task 21 (Filling missing parts in the main testbench):

1. Familiarize with the rand_stream_mst instantiation in the testbench. MinWaitCycles and the parameter MaxWaitCycles are set to 0 and 5 to ensure driving of valid data will occur at random intervals constraint between 0 and 5. Hence the name **constrained random verification**. Look inside rand_stream_mst to familiarize yourself with the SystemVerilog syntax of randomizing a given variable.
2. Complete the missing parts in the golden model block. Example code for the usage of the task rgb_gray_task is already given. You may use the rgb_sat_task available in the main testbench scope to calculate the saturated output. Golden model calculation should be initiated if inp_valid and inp_ready are HIGH, ACQ_DELAY timeunits after the positive edge of the clock. The calculated golden value gold_val should be pushed to exp_resp_queue using the push_back method.
3. Complete the missing part of the checker block. The following features must be included:
   a) At the event of positive edge of the clock, check whether both queues, acq_resp_queue and exp_resp_queue, are non-empty.

[6] Find a detailed description of each coverage type in: **/usr/pack/questa-2019.3-kgf/questasim/docs/pdfdocs/questa_sim_user.pdf** from page 921 onwards.

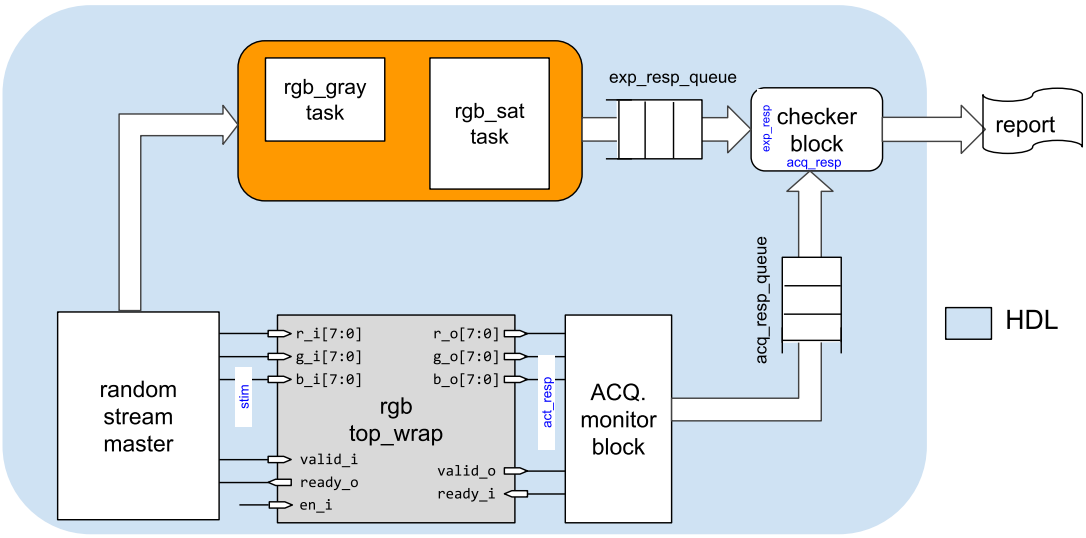

Figure 5: Constraint Random Verification Infrastructure

b) If the above condition is satisfied, increment `n_checks` and pop values out of the queues. You may call `pop_front()` on both queues to get the next-in-line outputs from the queues.

c) Compare the popped values `exp_resp` and `acq_resp`.

d) If they do not match, increment `n_errs` and print an appropriate message. An example error message can be found in the code.

It is now all set to launch the simulation in **`2_advanced/modelsim_2`** directory. Let's check the **`compile`** and **`runsim`** scripts.

Student Task 22 (Compiling and launching simulation):

1. Open **`compile.tcl`** in **`modelsim_2`** directory. Make sure switch `DEBUG` is ON and `COVERAGE` is OFF.
2. Open a terminal in **`modelsim_2`** directory and enter the following command:

```
sh> questa-2019.3 vsim -do compile.tcl -msgmode both
```

   If you encounter compilation errors, it means there is something wrong with the piece of code you added in the testbench or the way that you updated the scripts. Please fix the errors with the help of an assistant.
3. Run the simulation.

If everything worked fine, you should be able to observe about 224 mismatches out of 1000 checks between the actual and the golden model output printed in the *ModelSim* console. If there is a significant inconsistency please consult an assistant.

This indeed means that there are further bugs concealed in the design that the file-based testbench has failed to find.

Student Task 23 (Debugging the datapath path error):

1. Using the debugging techniques learned during the Gray counter example, find the data path error in the design.
   ***Hint**: You may compare R, G, and B channel data from the golden model and the actual data to first*

localize the erroneous channel(s) and then use the 'Event Traceback' feature to quickly check how signals are driven through the design in that particular channel.
***Hint**: Refer to Equation (1) if you are unsure about the correct output behavior.*

2. What is the fix for the error found?

Student Task 24 (Verifying correctness of the fix): Rerun the test with the **`compile.tcl`** and **`runsim.tcl`** scripts in the ModelSim shell and make sure the self checking testbench reports zero mismatches compared to 224 mismatches before.

Now we have verified that the design works properly for randomized pixel inputs fed into the DUT at randomized time intervals. Does it mean that the design is $100\,\%$ correct? We can still not be sure about that, because it is possible that some parts of the design are still not tested for all possible cases and case transitions, etc. To get a quantifiable idea about the level of coverage that the design has achieved with the given input, we can add some coverage assertions and additionally enable the coverage option.

Student Task 25 (Add coverage assertions):

1. Use a coverage assertion to `rgb_top_wrap` in order to test how often all three handshakes were successful in the same cycle. In other words, how often data was actually properly streamed through the design without stalls.
2. Add a second coverage assertion to measure how often all handshakes were successful (just like in the first task) for at least three consecutive cycles. This tests whether a data element passed the full design pipeline.
3. Rerun the simulation, check out how often these cases in your coverage assertions occurred by navigating to the *Cover Directives* tab ('View' $\rightarrow$ 'Coverage' $\rightarrow$ 'Cover Directives'). How often did your testbench cover the two scenarios? How good is the test coverage of our design?

---

4. Increase the number of random stimuli that the testbench applies until both scenarios were tested at least a couple of times.
***Hint**: Simply modify the test bench's `TOT_STIMS` parameter.*
5. *Optionally, you can add coverage directives to the wave view by dragging them to the wave window. To restart the simulation, simply use `restart -f; run -a`.*

Student Task 26 (Rerunning the test with coverage option):

1. Edit the **`compile.tcl`** script to set COVERAGE variable to ON. This prompts the simulation to perform an analysis of basic code coverage of the design. A more advanced and effective approach is to perform *functional coverage*, a topic we will keep for a later date.
2. Rerun the test by re-executing **`compile.tcl`** and **`runsim.tcl`** scripts in ModelSim shell.
3. You can now browse the coverage data in ModelSim GUI. A summary report is automatically created in **`modelsim_2`** directory, named **`myreport.txt`**.

Student Task 27 (Logging simulation statistics):

1. Please enter the following statistics related to the above simulation in the space provided:
Number of pixels checked, percentage statement coverage, percentage branch coverage, percent-

age condition coverage, percentage toggle coverage.

2. Which method (out of directed testing and constrained random verification) seems to provide better verification results? Why?

Discuss your results and any unclear part(s) in the exercise with an assistant.

You have reached the end of this exercise.
Please discuss your answers and any open questions with an assistant.
Finally, please provide us with feedback by filling out the form on our website. Thank you!

8 Assertion Cheat Sheet

This page gives some example assertions and useful operators for the exercise. For more information on assertions, check out the lecture slides.

Immediate Assertion

Immediate assertions have to be written inside procedural blocks (e.g., **`initial`** or **`always`** blocks).

```
// optional_label: assert ( EXPRESSION ) ACTION;
assert (APPL_DELAY < ACQ_DELAY)
  else begin
    $error("Aquisition delay must be larger than application delay!");
  end
```

Concurrent Assertion

Concurrently running assertions with their own sampling event. They check that certain properties and sequences hold during verification or issue a message.

```
// optional_label: assert property (@(posedge clk) CONDITION ) ;
// Make sure that b is true whenever a is false
example_assertion: assert property (@(posedge clk) !a |-> b)
  else begin
    $error("b was false while a was false!");
  end
```

Assertions may fail during reset period. Consider the example above and assume that both, a and b, will be reset to zero. In this case, the assertion will fail even though it might not be relevant during the reset period. To disable an assertion during reset (or any other event) you can use `disable iff ( EXPRESSION )`.

```
// Make sure that if a is true, so is b in the next cycle, unless rst_n is 0
assert property (@(posedge clk) disable iff (!rst_n) a |=> b);
```

Coverage Assertion

Coverage assertions count how often a certain condition occurred. This is useful to evaluate how often some events took place or if some corner cases were even triggered at all by the test bench.

```
// optional_label: cover property (@(posedge clk) CONDITION ) ;
// Count the number of handshakes
example_assertion: cover property (@(posedge clk) valid && ready);
```

Assertion Operators and Functions

Operator	Description
`a |-> b`	If a is true, b must be true in the **same** cycle.
`a |=> b`	If a is true, b must be true in the **next** cycle.
`a ##N b`	True if b is true **N** cycles after a is true.
`a ##[N:M] b`	True if b is true **N**, **N+1**, ..., **M-1**, or **M** cycles after a is true.
`a [*N]`	True if a is true for **N** consecutive cycles.
`a [*N:$]`	True if a is true for at least **N** consecutive cycles.
`$stable(a)`	True if a did not change compared to the previous cycle.
`$onehot(a)`	True if exactly one bit in a is HIGH.

Table 1: **Overview of important assertion operators**.