How to Build a Self-Checking Testbench

Testbenches, especially those you will use often or for a number of similar projects, should be self-checking and time-agnostic. Here are some tips on the best way to construct them.

by William Kafig
Senior Content Development Engineer
Xilinx, Inc.
bill.kafig@xilinx.com

A testbench, as it's known in VHDL, or a test fixture in Verilog, is a construct that exists in a simulation environment such as ISim, ModelSim or NCsim. Simulation enables a unit under test (UUT)—typically, your synthesizable FPGA design—to connect to virtual (simulated) components such as memory, communication devices and/or CPUs, and be driven with a known set of stimuli. These stimuli cause the UUT to react and interact with the virtual components. You can view both the stimulus and the reaction as waveforms in the simulation environment.

Here's quick example to illustrate how to implement a testbench using a simple 8-bit up/down with reset as the FPGA design (UUT). The testbench provides clock, up/down, enable and reset control signals. Figure 1 shows how to connect the UUT (central gray box) to a testbench.

The various functions on the left side of the diagram provide stimulus for the UUT which, in turn, produces a series of waveforms displayed in the simulation environment. Figure 2 shows a quick screen shot of the waveform view, both zoomed in and zoomed out. How clearly do you see the results? Do you see the values ascending, then descending on the count_out_pins? What if we zoom in (blown-up circle)?

Now we can see the values of the outputs, but we also need to validate each and every single value. That would be 256 values for increment and another 256 for the decrement, along with the special cases of enable off and reset and how these control signals affect the count value. While tolerable for a small design, this is a painful and ineffective means of verifying a larger, more sophisticated design.

Indeed, verifying a significant FPGA design can be more difficult and time-consuming than creating the synthesizable design. Given a known set of stimuli, a design should *always* produce a predictable collection of results—regardless of whether the simulation is behavioral, netlist or full timing (post place-and-route).

There are certainly many ways to create a stimulus set and drive it into the simulation—direct input from the simulation console, input from a text or binary file, scripted stimulus (.do or .tcl) and so on (Figure 3).

The question becomes, after the simulation has run, how do you verify that it was successful—that the generated output matches the predicted output as driven by the stimulus? Certainly you could pore over many dozens to hundreds (or thousands) of waveforms, but this is extraordinarily tedious and time-consuming, not to mention error-prone. You could dump thousands of lines of text to disk, thus requiring someone to examine each and every message—likewise tedious and error-prone.

You could even write a testbench that painstakingly describes each output waveform and tests specific values of the waves at strategic points in time. This type of testbench could then "automatically" detect errors. A wonderful thing, yes? Unfortunately, not only would you have to significantly modify this testbench each time the input was changed, but time does not always hold a fixed meaning in the context of simulation. Behavioral simulation has little concept of how a design may actually be implemented in hardware and no concept of routing delays; netlist simulation knows how the design is implemented, but is likewise ignorant of routing delays. Only post-place-and-route simulation (full timing simulation) can factor in all the delays for an accurate timing model, but this type has the longest simulation time by far. Thus, it is easy to see that the *exact absolute* time cannot hold constant through all these types of simulations.

This means that if a testbench looks for a specific value (output) at a specific point in time, that value is likely not to be in the same place in time in the behavioral simulation as it is in the post-place-and-route simulation.

Writing a separate verification testbench for each type of simulation is possible, but demands a significant time investment, not to mention the time spent in figuring out what is "correct" to begin with. Additionally, each time the stimulus (test vectors) change, you will need to recode the waveform checker module. Certainly you can use data files as the metric for the waveform checker, but the tedium of calculating the various values and times is still extremely costly.

Clearly, this static approach is not the path to take.

THE 'TIME-AGNOSTIC,' SELF-CHECKING TESTBENCH

We've just looked at a number of ways *not* to write a testbench. Let's now consider some specifications for an "ideal" time-agnostic, self-checking testbench.

"Time-agnostic" in this context means that the type of simulation (behavioral, netlist, post place-and-route) and hence the specific timing has no impact on the verification of the UUT. Self-checking means that the testbench is capable of generating its own valid output for each set of stimuli without the user having to explicitly enter it.

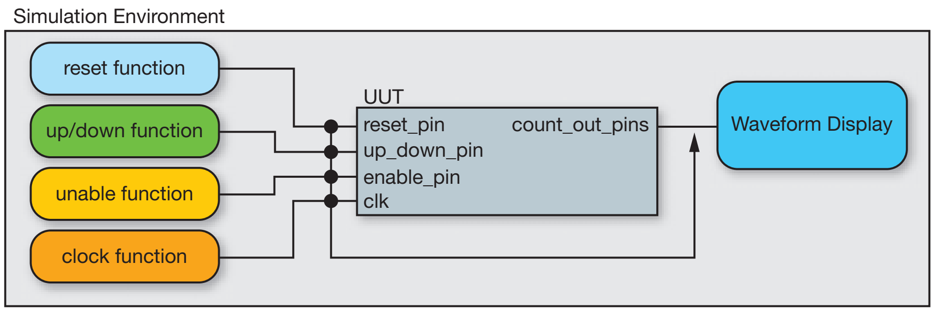


Figure 1 – Simple example of a UUT—that is, your synthesizable design– and how it connects in a testbench

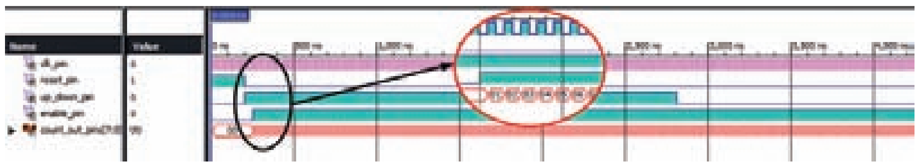


Figure 2 – Zoomed-out waveform shows "big picture," but lacks details. Zoomed-in waveform shows details, but no context.

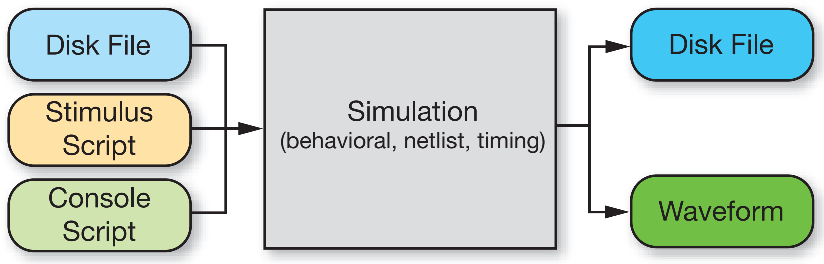


Figure 3 – Generic testbench construction

To use the language of specification, the testbench: shall be valid regardless of the type of simulation being performed; shall operate with any set of valid test vectors; and shall report the status of the various tests in the stimulus set in a clear and easy-to-read format.

You can achieve these specifications by using a technique that creates a parallel flow to the UUT. Stimulus, regardless of its source, is fed to both the UUT and the parallel path. This parallel flow comprises a *behavioral model* of the UUT followed by an asynchronous FIFO that adjusts for any time discrepancies between the behavioral model and the UUT.

A "waveform comparator" detects the new output of the UUT and pulls the next value from the behavioral model's FIFO. These two values are then compared and the result is registered, clearly and unambiguously, highlighting the matching and nonmatching behaviors between the paths. Figure 4 illustrates a typical model of this type of testbench.

THE CORE OF THE DESIGN

The core of the design consists of the UUT and the known-good behavioral model (KGBM) of that UUT. The UUT is simple enough—it is the synthesizable design you are trying to implement. The KGBM behaves according to the specification of the UUT, but is coded using behavioral constructs.

By constructing a "perfect" version of the UUT using behavioral modeling techniques—*which don't have to be synthesizable*—you (or rather, your waveform comparator) can quickly identify any differences between the outputs of the UUT and the KGBM. Behavioral modeling is generally easier (and faster) to code, as it doesn't need to meet the rigors of synthesis and timing closure.

Construct the behavioral model hierarchically. Once you have created the lowest levels, you can simulate them piecemeal with their own small testbenches to verify proper behavior of the overall model, in the same way as you would simulate a synthesizable design. These small testbenches generally do not need to be time-agnostic or self-checking. Ad hoc (manual inspection of the waveforms) is usually sufficient to validate them.

CREATING STIMULUS

You can create stimulus in a number of ways, starting with console input. Many simulation environments allow the user to "force" values into signals. While this technique may be appropriate for some designs or certain signals, larger, more complex designs benefit from easily repeatable inputs so that it's easier to perform analysis and debugging.

You can also script stimulus using "tcl" or "do" scripts, depending on which simulation tool you are using. While this approach has a number of benefits, you need to take into account the type of

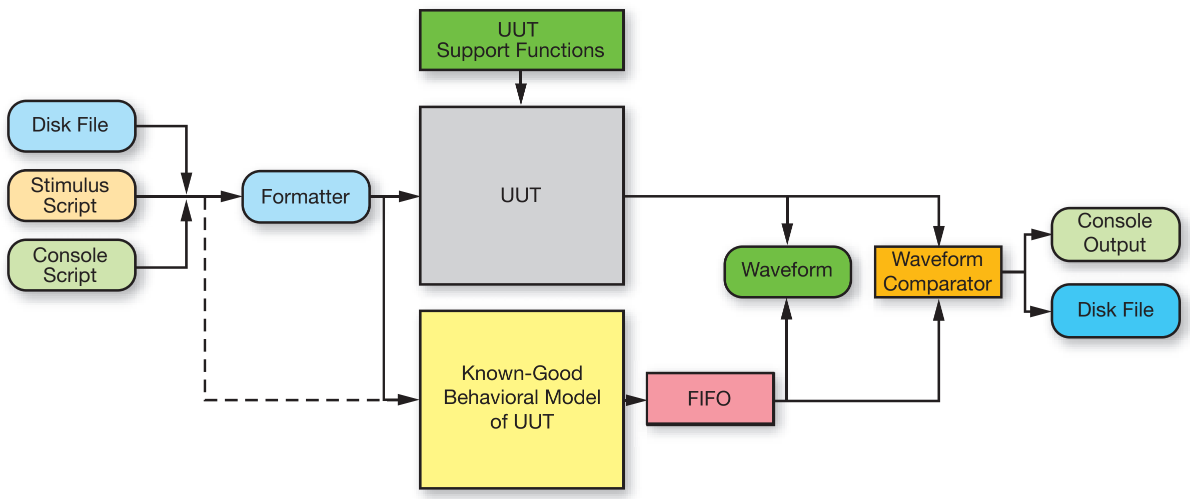


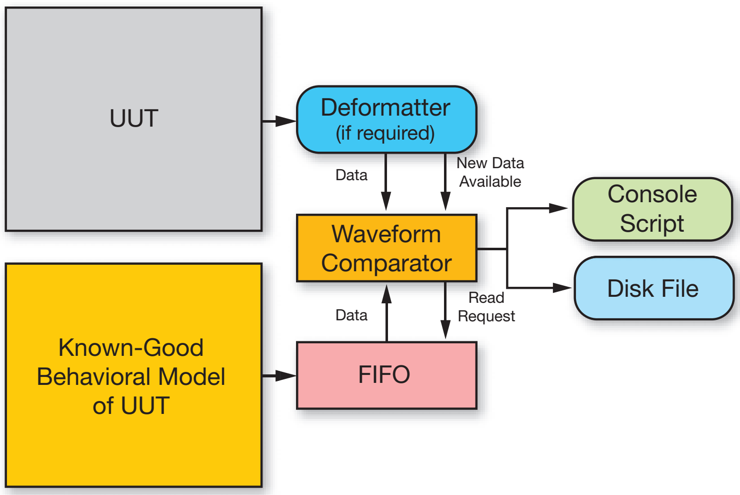
Figure 4 – Generic block diagram for a time-agnostic, self-checking testbench

data that you are pumping into the UUT/KGBM. Scripting is especially attractive in cases where there are a number of inputs that can be simulated in an unpredictable way, such as jitter and pseudo-random noise or data.

The disk file stimulus, provided by a simple text file, is an ideal mechanism for loading simple data streams. An ASCII text file merely contains the values that will be pumped into the design. As an example, let's use the WaveGen design, which Xilinx provides as a reference design with the ISE® tool suite and uses in many customer education classes.

Regardless of the source of the stimulus, you must convert the information into a form that the UUT can digest. You may need a "formatter" of some sort to handle this task. The KGBM, because it needs to model only the *behavior* and not the actual process of the UUT, may be able to use the input stimulus directly. For example, if the UUT needs to process data contained in an Ethernet packet, then the formatter must collect the data from the stimulus source, create a packet and then send the packet to the UUT. The KGBM, on the other hand, doesn't need any header information (unless it needs to make decisions based on this information), but rather just the data itself. This means you can feed the "raw" stimulus directly to the KGBM and use a formatter to package the stimulus for the UUT.

The UUT support-functions block that you've coded for your specific design supplies the necessary clocks, resets and other control signals required to keep the UUT running so that it can process the incoming data. Since the KGBM is not synthesizable, all timing and reset signals can be contained within the model or shared with the UUT.



```
if (New Data Available) then {
   issue Read Request to FIFO
   compare Data from UUT with Data from FIFO
   if match {
      report success to Console Output
      report success to Disk File (test name or data)
   } else {
      report failure to Console Output
      dump failure information to Disk File (time, UUT data, KGBM data, etc.)
   }
}
```

Figure 5 – Block diagram details and pseudo code for the waveform comparator

THE ASYNCHRONOUS TIMING FIFO

The UUT is constrained by the speed of its clocks and the latency of the architecture; the KGBM is not. Since the KGBM is constructed as a behavioral model, it is the *function* that is important and not necessarily the timing (the exception being any output that other devices use, such as RS-232, Ethernet and the like). Since the KGBM will produce an output before the UUT, some mechanism must be present to align the outputs of the UUT and KGBM so that they are presented to the waveform comparator block simultaneously. Asynchronous FIFOs excel for this type of task.

When the UUT presents its outputs (which will be later in time than the KGBM), the waveform comparator block reads the next piece of data from the FIFO buffering the KGBM data. In this fashion, the relative performance differences between the UUT and the KGBM become irrelevant; therefore, it doesn't matter if the UUT is simulated from the source-level code, from the netlist or from the post-place-and-route design. The performance of the UUT is automatically compensated for and no modifications to the testbench are required when the user changes the type of UUT simulation.

WAVEFORM COMPARATOR

As with the stimulus, you may need to compensate for the format of the outputs. The WaveGen example provides its key output in the form of an RS-232 stream. The KGBM can output in "character" form. This implies that there must be some type of UART receiver or equivalent "deformatter" built into the waveform comparator (Figure 5).

With the information from the UUT and the KGBM, the waveform comparator checks to see if there is a match. If there is, data can be marked as "successful" and written out to either the simulator console output or a disk file. If there is a mismatch between the UUT and KGBM outputs, then the comparator can display an appropriate error message and alert the user to this discrepancy. As with good data, you can see this mismatch immediately on the console or save it to an ASCII disk file for later analysis. The advantage to textual output, whether via the console or an ASCII file, is that of data filtering. Often the user just wants to know if the simulation was successful or not. When not successful, it is helpful to be pointed to the time index in the simulation where the mismatch was found so that you may glean further information from the waveform view. Generally, you should send only simple and highly relevant information to the console and more detailed information to the text file. In this way you can quickly scan the output of the console to see which tests passed and which failed. You can then reference details of both in the text file.

You may also elect to halt the simulation when a mismatch is found. This is also achievable through the waveform checker module.

For those who enjoy reading waveforms, the ISim simulator tool, along with most other simulators, will display what is happening graphically as a default. This graphical representation may yield additional information for debugging as it contains all the monitored information at every cycle of the simulator. It is because of this vast amount of data that the user benefits through textual messages to the console and disk to more quickly "see" a successful result, or pinpoint any mismatches.

WAVEGEN EXAMPLE

The WaveGen reference design is a simple design that illustrates several important concepts used in the development of an FPGA, including crossing time domains, synchronizing asynchronous data and multicycle paths, among others. The basic function of the WaveGen design is to load a pattern into memory and "play" it back out at varying rates, thus forming a simple waveform generator. For demo boards so equipped, the output can be played through a speaker at audio rates.

The WaveGen design receives an ASCII text command and data via an RS-232 input. Therefore, the stimulus file needs to contain a set of commands to store a pattern in the memory, read some of the pattern back out (as verification that data was properly loaded into memory), configure the "playback" rates and initiate the automatic playback. Table 1 shows the VHDL and Verilog modules needed for the WaveGen testbench.

'Ideal'	VHDL Module Name	Verilog Module Name
Formatter	tb_uart_driver.vhd	tb_uart_driver.v
UUT	wave_gen.vhd	wave_gen.v
KGBM	tb_wave_gen_model.vhd	tb_cmd_gen.v
FIFO	tb_fifo.vhd	tb_fifo.v
Waveform Comparator	tb_resp_checker.vhd	tb_resp_checker.vhd

Table 1 – Mapping of 'ideal' testbench to WaveGen testbench source code

EASIER CODING

Testbenches, especially those that you will use often or for a number of projects, should be self-checking and time-agnostic. Self-checking is derived by driving the stimulus (inputs) from either a file or script which provides consistent, repeatable and documentable input into the testbench; and the use of a waveform comparator that compares the output of a known-good behavioral model of the design with the UUT. The comparison can be automated and can report the success and failure of various sets of inputs to the console or a text file (or both), thus providing a quick mechanism for the designer to check success or failure.

You can code the KGBM using behavioral modeling techniques rather than strictly synthesizable constructs. Coding in this fashion is easier (hence the time developing the model is significantly shorter than developing the UUT) and produces outputs earlier than the UUT without a significant impact on simulation time. Theoretically, the output values of the KGBM and UUT should match for every input set; therefore, it is easy to detect and report differences.

Bill Kafig *is a senior content development engineer working in the Xilinx Global Training Solutions team. He is currently neck-deep preparing for the release of the Zynq™ training material and has previously been involved with Xilinx's partial reconfiguration, PCIe®, C-language, VHDL and other classes.*

Prior to his joining Xilinx more than four years ago, Bill logged 25+ years of experience spreading the gospel of programmable logic throughout North America. He has done time as a project manager, systems engineer, consultant, FAE and digital grunt for a certain government agency.

Reach him at williamk@xilinx.com.