Integrating Perl, Tcl and C++ into Simulation-Based ASIC Verification Environments

Michael D. McKinney
Senior Member of the Technical Staff
DLP™ Products ASIC Development
Texas Instruments, Inc.
Dallas, TX

Abstract

As ASIC designs become more complex, it follows that the complexity of the verification environments for such designs increases dramatically as well. However, while System-on-Chip methodologies and thought processes have been strongly accepted and utilized for the HDL design, there has not been a concurrent type of strong process taking place for verification environments. That is, the HDL of an ASIC design can be divided, even sub-divided, into understandable but reasonably sized components whose behavior can be comprehended in a reasonable amount of time. However, any verification environment that is created or generated for these design sub-blocks remains highly complex, whether written in HDL or any of the various verification or scripting languages now available.

This paper will address issues faced and lessons learned by an ASIC design team whose product is a highly complex SOC-based design. The team's desire was to integrate C++, Tcl and Perl together in a coherent, highly intelligent and usable verification environment for the ASIC. This effort was highly successful (although there have been some less encouraging moments along the way) and the resulting simulation environment is being used now with acceptable results.

1. Introduction

Although there are several examples of "verification languages" in the market today, and still unproven new ideas and methodologies (like formal methods), it seems that circuit verification at any level, FIFOs to ASICs, still most often uses simulation as the methodology of choice. Naturally, simulation results are coupled with the outputs of other tools, such as code and functional coverage and equivalence checkers to quantify a point in time

A common example of a mixed-language verification environment includes compiled C++ objects via the FLI of VHDL or PLI of the Verilog HDL. With many of the verification tools now supporting (or being written in) Tcl, verification environments can easily integrate this language as well (for example, to control the action of the simulator). In the case to be discussed in this paper, C++, Tcl and Perl are integrated as part of the simulation environment for the verification of a complex SOC-based ASIC. Many lessons were learned in creating this level of integration, and many nice ideas were implemented. This effort sometimes resulted in good controllable behavior in which visibility into the design was maintained, and sometimes resulted in simulations that not only could not be easily controlled, but also could not easily be debugged, either. This paper will briefly address these issues in an effort to inform others about this level of integration.

2. The Big Picture

Before discussing details, it is appropriate to take a look at the "big picture". That is, how all of these languages, the design HDL and the simulator all fit together into a coherent whole. Please refer to Figure 1 below for further information.

The simulator is a modern, HDL-independent single-kernel, event-driven simulator that has extensive debugging capabilities and the Tcl interpreter built in. The simulatable test environment is written in VHDL in the traditional way, and optional components are selected by a 'generic' test name and a test-specific VHDL configuration. However, rather than have the stimulus patterns driven by compiled VHDL or Verilog components, they are driven by Tcl code. Members of the verification team have written a large number of Tcl procedures, all of which are available to the persons writing test-specific verification code.

The simulator is invoked and told to load and execute a core Tcl subroutine after a successful elaboration phase. When called, this procedure performs several front-end tasks such as gathering hierarchical paths to selected components, sensing what optional

components have been chosen in, sensing whether the simulator is running is graphical or batch mode, and setting default conditions in the VHDL verification environment. At the end of these tasks, this procedure loads the test-specific Tcl file as in-line code, executing it line-by-line through the Tcl Interpreter (step (A) in Figure 1). After completing the test-specific Tcl, the procedure permanently stops the simulator. In batch mode, the simulator process also ends.

At some point or points in the execution of the test-specific file a Tcl subroutine is called whose entire task is to execute the Perl interpreter containing algorithmic code representing the function of the HDL design (the 'reference' model) and with data representing expected stimulus patterns up to that point (step (B) in Figure 1). It is quite possible that the simulator has not yet 'run' at all, yet. The reference model works on the set of expected stimulus patterns and produces a set of expected result patterns. Referring to Figure 1, stimulus patterns are generated in path (1), expected results are generated in paths (2) and (3). When necessary, the Perl reference model may execute some set of pre-compiled object code representing parts of the ASIC function. Compiling C++ files with cpp generates this object code.

At this point, the same stimulus patterns given to the reference model are given to the HDL pattern generators or driven directly to HDL signals from Tcl (path (4) in Figure 1), and the simulator is released to 'run' (step (C) in Figure 1). Finally, when a stopping point is reached, the output from the HDL design is compared, line-by-line, bit-by-bit and clock-by-clock if necessary, for congruence with the expected results generated by the reference model (path (5) in Figure 1), and the entire scenario stops.

2. The Reference Model

The simplest explanation of circuit verification in general is that the designed circuit is compared against a golden or reference model. Verification using simulation naturally assumes that the circuit design is expressed in an HDL of some kind, and assumes that identical stimulus patterns will be applied to the HDL and to the reference model. Furthermore, the actual

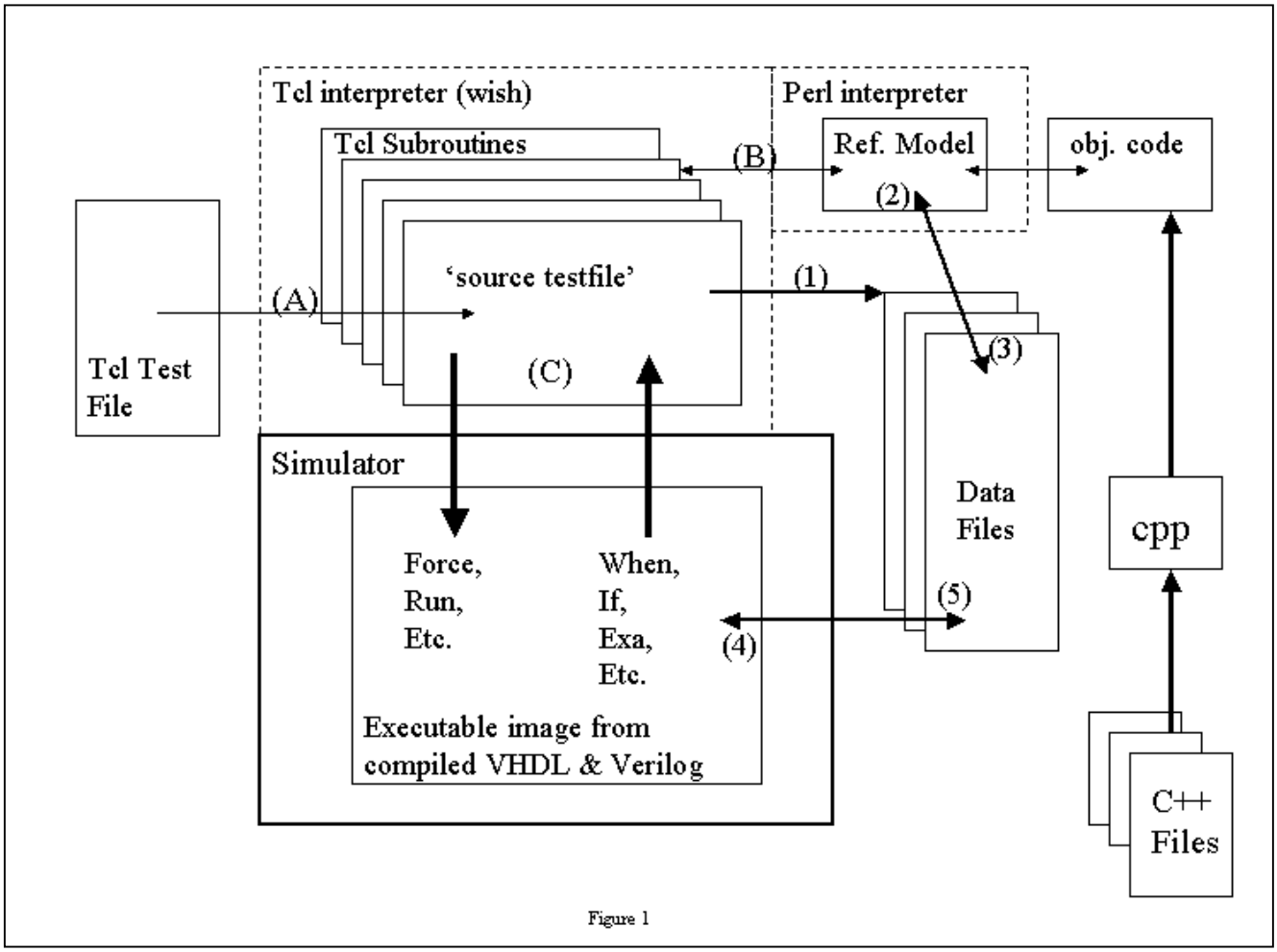

Figure 1

verification checks occur when the outputs of each model are compared.

It is understood that a large percentage of design teams now kick off their design project with the definition of a specification document for the circuit being designed. Two different persons or groups then use this specification document: one writing the HDL design code, one writing the reference model. Without this separated effort, the circuit designer is faced with creating verification patterns and checks for his/her own design – naturally, the circuit will easily verify to a very high level (however, may not have detected enough corner cases). With the separated effort, the verification process will likely take a little longer, but will be more robust because of the individual interpretations of the specification.

The reference model for the design being addressed by this paper was written in Perl, and will execute within the Perl interpreter. Naturally, this code could have been written entirely in C++ or another separately compilable and available language, or even Tcl itself, or possibly even "behavioral' style of VHDL or Verilog. Perl was the choice based on speed of execution when compared with HDL, but the reason for the decision against any other compiled language has been lost in the history of the project. The Perl code for the reference model was generated from the specification document for the design, in some (but obviously not entirely perfect) isolation from the HDL designers. The reference model furthermore is bit-accurate but not necessarily clock-cycle accurate. That is, very large sets of input vectors may produce fairly small sets of output data, measurable only during specific strobe events. The group that developed the model is still responsible for the maintenance of the model.

Writing the reference model so that the handshake and file interface with the controlling Tcl was complete and thorough was a burden in the beginning of the project, simply because that code portion represents 65-70% of the effort. The code segments that represent the functions of the designed ASIC were generated in a straightforward manner from the specification. Although the reference model as written serves our needs very well now, new ideas for the model involving streamlining or another Tcl handshake methodology would require a complete rewrite or at least major re-structuring of the entire code block.

3. Stimulus and Response Handlers

A second problematic issue involving the Perl reference model that the team faced was how to tell the model about the stimulus patterns being generated by the test-specific Tcl file. These patterns would necessarily be different based on what the specific patterns were verifying in the HDL design. Since Tcl and Perl are not the same language and therefore cannot share the interpreter, internal variable or strings, etc., the only viable alternative was to share files. This has caused (still causes) additional overhead because of having to create and write the files from one process, and then open, read and parse them in another process. Furthermore, the two processes had to agree exactly about the format of the files being written and read. With large stimulus files this can become very (very) (very, very) cumbersome.

The answer for this team has been to transform very short Tcl subroutine calls:

id_data x"000a" x"1234"

into fully expanded file records for the reference model:

id_data b"0000000000001010" b"000100100110100"

and also into simple sets of 1/0 characters to be read and used as std_logic values by the VHDL stimulus handler:

0000000000001010000100100011010 0

In special cases a subroutine might convert to an entire set of 1/0 vectors, to be sent to the component being verified on each clock cycle, rather than just one vector as shown. The resulting files could be hundreds or even thousands of lines long.

The reference model reads and parses the files and sends the input patterns through its functional algorithm. This action produces files containing long records of 1/0 data representing the expected HDL output. This data is taken (much like the stimulus data is taken) by a VHDL checker mechanism, using the data to verify the output of the HDL design bit-by-bit. It is clear that several files will be needed for each test, all of which will be reread / reparsed / regenerated each time a simulation is run for each test. Altogether a large overhead cost involving Tcl, Perl and HDL elements that must work together error-free during a simulation run.

For this mechanism to work well HDL stimulus and response pattern handlers must be constructed for the test environment that have common attributes:

1) They are designed as test architectures of normal ASIC sub-blocks, to be used anywhere in the HDL design that component-specific tests are desired. This means that many HDL code elements, perhaps very similar ones, may have to be written. It may also mean that careful use of a VHDL configuration for the ASIC design is a requirement for any successful simulation.

2) The HDL code in the stimulus and response mechanisms must support the specific vector length required by the component being verified. This aspect will be controlled by the entity definition for the component being verified. A 37-bit stimulus generator

for component A cannot be used to drive the 83-bit vector required for component B. Most often this requirement is hard-coded into the HDL.

3) The components being verified are unique, and therefore may have unique strobe times and strobe events for input drive and output comparison. The HDL for the stimulus and response mechanisms must be aware of how Tcl will give or receive data for specific component tests. Therefore, they must be setup with specific signal names, generic names, file names etc. to accommodate handshaking with Tcl for the specific component being verified.

The design team has been working on this issue for some time and now has a substantial number of stimulus and response mechanisms written and debugged. Team members have found ways to reuse some of this coding for multiple tests, and have found and implemented ways to speed up execution and ensure a stronger connection between HDL and Tcl. Additionally, some idea are being expressed about using "behavioral" style VHDL coding with access types and linked lists to load larger packets, etc. The path getting to the current point, however, has not been easy so far, and the firm outlook is that these mechanisms can only be improved.

4. Debugging HDL

A tenacious and problematic issue in this configuration of simulator and languages is that of debugging the HDL design code. Stated a different way, it is that by using Tcl to control the simulator, the very powerful HDL debugging features of the simulator are not available. Most of this difficulty is based on the way the Tcl code itself is written; however, the Tcl is written the way it is because of a decision early in the project to give the Tcl full control of the simulator.

Activities that normally would be done in a VHDL stimulus package using assignment statements, or by the user entering commands directly in the simulator window (like 'force', 'examine', 'stop', 'break', 'resume' or 'run' commands) are done entirely by executing specific Tcl subroutines. As stated above, a core subroutine containing test-specific code is executed as soon as the simulator has loaded and elaborated the design. This core Tcl must determine, for example, whether the simulator has been invoked in interactive mode (with the GUI) or in command-line mode (output only to STDOUT, and no Tcl/Tk coding allowed). With this infrastructure, only a forced 'break' command is recognized with enough strength to pause the simulator (this would be something like a user entering 'Ctl-c'). Any other induced break, such as is caused by setting a breakpoint on a line of HDL code, is treated as temporary and automatically causes the execution of the 'resume' command. Therefore, no HDL breakpointing, single-stepping, or temporarily stopping to get current signal levels is possible.

The only solution (and its considered only a partial one) has presented itself through the careful evaluation of the Tcl code to determine exactly how the 'break' is handled. It turns out that the 'break' command is detected by an autonomous subroutine that waits for the 'break' event. When the event is detected, the subroutine determines the operating mode for the particular simulator invocation ('batch' or 'interactive') and performs the proper function. That is, a 'batch' mode simulation must stop and abandon the simulator on detection of a 'break', while 'interactive' simulations may 'pause', allowing users to 'resume' after debugging. The actual coding methodology used disables the autonomous subroutine while in 'pause' and re-enables it upon 'resume'.

The core Tcl upon its initial call must set up this autonomous subroutine after the simulator has successfully loaded and elaborated. While this solution is workable, it also greatly increases the complexity of the Tcl being loaded and used. In addition, any new Tcl must conform to the strict rules for enabling and disabling the autonomous subroutine in the core Tcl. There must be a better way.

5. Tcl / Simulator Interaction

There are several issues involved with the interaction of Tcl and the simulator that cause a lot of heartache among the design team. Among these issues are 1) debugging the Tcl code, 2) detecting syntax errors in the Tcl code ahead of time, 3) what happens when the Tcl is busy for several seconds or minutes and 4) what happens when the simulator is finally released to simulate in the normal way.

1) In the currently implemented system the process of debugging the loaded Tcl during a simulation – using breakpoints, stepping over or into subroutines, evaluating variable values and generally following the execution flow of the Tcl code – is impossible. While the Tcl code is easily seen and manipulated in a UNIX window, it cannot be seen at all during a simulation. It is an internal and background piece of the simulator. This action has been and continues to be very tiresome, especially when developing new Tcl subroutines to support the ASIC verification, or developing new test-specific Tcl code itself. There is no straightforward way to determine that you have written the correct thing unless it is already correct.

2) Because the Tcl cannot be compiled or loaded ahead of time (before the simulator is invoked or outside of the simulator), then it becomes highly likely that syntax errors will be present when the Tcl is loaded, which in turn will cause the simulator to crash – but note,

only *after* a successful load and elaboration phase. This is a pure waste of time, very frustrating and a strong damper on productivity while using the system. Attempts to load Tcl directly into 'wish' fail because built-in simulator subroutines are being called that do not exist outside of the simulator.

3) With the current system the Tcl is responsible for opening, closing, parsing, creating and otherwise manipulating several files during a typical simulation. These files may contain stimulus or response data for processing by the ASIC design, or log data, and may be very long. This means that, for many simulations, the Tcl is busy for many real-time seconds or perhaps minutes with these files. Unfortunately, the simulator is hung during those times: unresponsive to button pushes, GUI windows will not update, commands cannot be typed in, restored windows are blank, etc. Even direct output messages to the simulator windows do not show up. This state of affairs remains until Tcl is finished with the files and provides at least one 'run' command. At that point the simulator wakes up and updates its environment.

This is an annoying problem because most users of UNIX and the graphical user interfaces associated with most modern EDA tools begin to get very anxious when the GUI is frozen for more than a few seconds. That this action is not the fault of any one process but rather the particular combination of processes used in this environment gives an excuse, but does not make using the environment any easier.

4) Finally, when the simulator is released to run in the normal way the Tcl interacts with the environment minimally, but it is the Tcl that is responsible for submitting the 'run' commands. As with other like tools, the simulator will update its set of GUI windows at the termination of each 'run' command. When the Tcl code submits many very short commands, such as 'run 1 ns', the GUI windows update very often and very rapidly, producing an annoying flicker in the monitor. Furthermore, the simulator is slowed in this process by having to re-draw the windows very rapidly before processing the next 'run' command. Longer 'run' times produce the window update at longer separations, and naturally it is expected that a command like 'run –all' will update only once at the end of stimulus or at some time that the user interrupts the simulation.

6. Solutions

For the annoying GUI window flickering produced by short 'run' commands, the solution has been to re-code some of the Tcl to reduce the number of commands submitted, and, where possible, to make the 'run' commands of longer length. This strategy has been moderately successful, resulting in a faster simulation with less flickering, but the problem will not be eliminated until *all* of the Tcl is examined for this activity. One technique for doing both updates at the same time has been to write an accessible and re-triggerable VHDL process (in a special component architecture so that it may be configured in several different locations in the design) to take over a sequence of events from Tcl. The Tcl code initiates the process by setting a signal, then submits one 'run –all' command and waits for a completion signal. When this was accomplished in the current environment to supplant a very often-called Tcl subroutine, the effect was dramatic and very satisfying.

A complete and satisfying solution to the issues of debugging HDL and Tcl has not as yet been discovered. Recoding the current set of Tcl (other than by 'echo' commands for each line of code) cannot increase the debugging capability of the embedded Tcl. However, documentation relating to the newest version of the simulator indicates that a Tcl debugger is part of the package, and may provide that part of this solution. Because debugging the HDL code relies on 'break' events it is feasible that the Tcl can be rewritten to accommodate this activity. To some extent this has been done in the environment, and under careful control some minimum debugging can be done now. However, because a 'break' may be generated by a large set of other events, a complete Tcl rewrite to accommodate total HDL debugging will not be an easy one. The result has been that users simply do their debugging in some other way, such as viewing the waveforms to detect invalid behavior.

How to pass information back and forth between the simulator, Tcl and the Perl reference model has also only been partially resolved. Files are still used because there simply is no sharable address space between the Tcl and Perl interpreters. Several ideas have been brought forward, including a) Re-writing the reference model in Tcl, b) re-writing the reference model totally in C++, and allowing Tcl to invoke it directly (eliminating the Perl interpreter), c) compiling the Perl or Tcl (perhaps compiling *all* of the embedded Tcl), and d) somehow compiling all of the Perl, C++ and Tcl into one piece of object code so that all parts can share internal memory. None of these ideas is viable for various reasons, but the most prevalent reason is manpower and schedule.

7. Conclusions

The primary conclusion that ought to be formulated from this paper is that a simulation-based ASIC verification environment that includes Perl, Tcl, and C++ can be successfully built and used. Specifically, using Tcl gives users access to internal simulator events and variables and can make the 'comparison of values' phase of verification much more automated. Additionally, Tcl is becoming the language of choice for

developers of the user interfaces for more and more EDA tools, for activities such as waveform analysis, simulation and synthesis.

A fair and equitable balance of simulator control between the user and the embedded Tcl can and should be implemented in a system such as this one. Such a balance would allow the users to have control in GUI-based simulations for such things as HDL debugging, and allow Tcl to have control for non-graphical or regression simulations where user interaction is minimal or non-existent.

More of the embedded Tcl code could be written to supply data to and compare data from the HDL, and less written for the purpose of controlling the 'run' behavior of the simulator. This can be accomplished by giving control for long and complex stimulus activity to some HDL process, instead of giving total simulator control to the embedded Tcl.

Finally, readers may conclude that this paper presents a strong argument for *simplicity*. For example, much (if not all) of the activity being accomplished in Tcl in the current environment can also be done in HDL (VHDL or Verilog or a combination). In addition, the reference model itself could be written in an HDL and simulated concurrently with the UUT. Both of these ideas simplify the verification environment by removing the addition and interaction of Tcl and Perl with the simulator. Admittedly, the HDL code that resulted from such a decision would still be very complex, and would still have to be maintained by two separated and isolated groups. And, such a decision would be best made at the beginning of a project rather than in the middle. However, the overall experience with the environment could very well be smoother, and produce an equal level of design HDL verification.

8. References

[1] *ANSI/IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993. IEEE Press, New York, NY, 1993.
[2] *IEEE Standard Verilog Language Reference Manual*, IEEE Std 1364-1995. IEEE Press, New York, NY, 1995.
[3] Raines, Paul and Jeff Trantor, *Tcl/Tk In a Nutshell*, O'Reilly & Associates, Sebastopol, CA., 1999.
[4] Conway, Damian, *Object Oriented Perl*, Manning Publications, Greenwich, CT., 2000.
[5] Wall, Larry, et.al., *Programming Perl (Second Edition)*, O'Reilly & Associates, Sebastopol, CA., 1996.
[6] Christiansen, Tom and Nathan Torkington, *Perl Cookbook*, O'Reilly & Associates, Sebastopol, CA., 1998.
[7] Gilly, Daniel, *UNIX in a Nutshell*, O'Reilly & Associates, Sebastopol, CA., 1992.
[8] Flint, Cliff, *Tcl/Tk for Real Programmers*, Academic Press, San Diego, CA., 1999.