

# TCL\_PLI, a Framework for Reusable, Run Time Configurable Test Benches

Stephan Voges and Mark Andrews,  
Electronics For Imaging, Inc.

## Abstract

*As ASIC complexity keeps increasing, the time spent in design and maintenance of test benches has grown to become a disproportionately large part of the total design effort. Verilog test benches have become slow to compile and cumbersome to maintain.*

*This paper discusses a scripting approach to managing the test bench complexity issue. Partitioning the functionality of a test bench between Verilog and a scripting language allows for a significant reduction in compile times during ASIC verification. If done correctly, partitioning also offers great potential for re-use of test bench components.*

*The Tcl language was chosen as a basis for implementing a library of PLI routines that allow fully customizable interpreters to be instantiated in Verilog test benches. This library allows multiple Tcl interpreters to be instantiated in a Verilog simulation. The Tcl interpreters can interact with the simulation and cause tasks to be executed in the Verilog simulation.*

*We have found the TCL\_PLI library to be extremely valuable in speeding up our verification efforts on multi-million gate ASICs and have decided to make the library available to the general design community.*

## 1 Introduction

Our traditional approach to test bench development and ASIC verification was to write a single test bench that contains a number of tasks that exercise different aspects of the designs. Team members would add new tasks as the verification effort continued, and tests would be run by calling different sets of tasks in different order, depending on the functionality being tested.

The problem with this approach was that the test bench had to be recompiled for every new simulation. Even though compiled simulators offer features like incremental compilation, in which only the code that changed was recompiled, this still meant that a lot of time was spent compiling test benches.

Our initial efforts to reduce this problem involved using “configuration files.” In this approach, we would try to make the test bench code to be “all things to all people.” Test benches typically contained complicated case statements and if-then-else sequences. These were controlled by parameters that were read from a text configuration file when the simulation was launched.

This approach eliminated the need to repeatedly recompile test benches, but introduced an even worse problem: test benches became extremely complicated, very difficult to maintain, and very application specific. Very often when new functionality was added to a test bench, it had side effects that caused other tests to break. At the end of the project we had a test bench that very few people understood, and was so convoluted that there was no possibility of reuse.

## 2 Scripting as a solution

We decided that the solution to our dilemma was to make the “configuration files” more powerful, so that some of the complexity that was coded in Verilog could be shifted over to something that was evaluated at run time. In essence, we needed a scripting language that could interact with the simulation.

With this approach, we could design a simple test bench, which would contain a number of tasks that handle low level interaction with the device under test. If these tasks could then be called from a scripting language, we could develop different verification scripts; each tailored to verifying a specific aspect of the design.

The key difference from the configuration file approach is that the intelligence that determines the ordering of events is moved from Verilog to a script file that is interpreted at run time. Different people could then use the same test bench to test a design in completely differing ways. They would have the benefit of significantly reduced need for recompilation, and different people on the verification team would be completely insulated from one another. Each would be developing their own verification script, and changes to that script would affect only their own simulations.

### 3 Tcl as a choice of scripting language

We started out with the idea to extend the configuration files that we were already using to handle simple conditional and looping constructs, and to extend the PLI routines that we were using to process these configuration files to become a simple interpreter.

Fortunately, very early in the project, we decided to investigate the feasibility of using Tcl as our scripting language. This turned out to be a very good idea. It was immediately obvious that John Ousterhout, the inventor of Tcl, had been in our situation many times. It was after many attempts to write custom interpreters for various tools, that he decided to write the ultimate, reusable and embeddable interpreter[1].

Tcl had all the characteristics that we were looking for: It was already developed, it was free, it was easily extendable, and it was easy to embed in an application. In a nutshell: it was invented for this specific purpose.

### 4 The obstacle

We only had one problem to overcome: The Verilog language is extended through function calls: If a user needs new functionality, he implements it in C, and uses the PLI API to make it available as a new function that can be called from Verilog. Tcl is also extended through function calls: new functionality is implemented in another compiled language (typically C) and is linked into Tcl as a new Tcl function.

We wanted to implement a system in which the Verilog simulation could start up a Tcl interpreter and instruct it to run a script. This implied a PLI function call. At some point the Tcl interpreter would encounter a function that is mapped to a certain Verilog task. It then has to pass control back to Verilog so that the task could be executed. This implies that the PLI call that invoked the interpreter has to return, leaving the problem of how

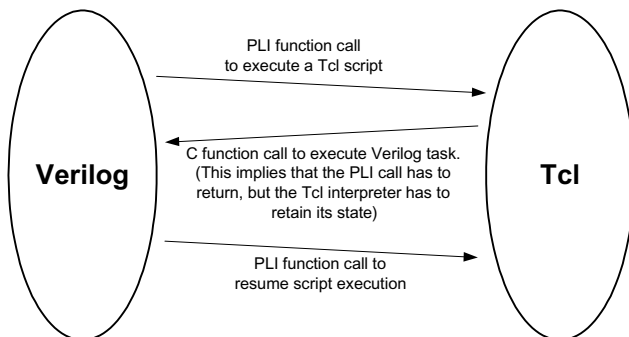


Figure 1: Required interaction between Verilog and Tcl

to resume execution of the Tcl script after executing the Verilog task.

We had to figure out a system through which we could pass control between Verilog and Tcl, through function calls on either side, so that the Verilog code controls when Tcl interpreters are invoked, but that the Tcl interpreters could cause Verilog tasks to be executed (implying that a PLI call needs to return), while retaining the state of the Tcl interpreter so that it could resume execution of the Tcl script after the Verilog task completed. Figure 1 illustrates the required interaction between the Verilog simulation and a Tcl interpreter.

### 5 The solution

Our solution to this problem was to implement a simple client-server model, in which the Tcl server runs on a separate thread from the Verilog simulation. Synchronization between the Verilog simulation and the Tcl server is done via a set of semaphores. This allows control to be passed freely between Verilog and Tcl. The Verilog code determines when Tcl interpreters are invoked and Tcl interpreters can randomly cause Verilog tasks to be executed.

The TCL\_PLI library allows any number of Tcl interpreters to be instantiated in a Verilog simulation. Every interpreter is completely customizable. PLI functions are used to initialize interpreters and define new Tcl functions that are mapped to Verilog tasks. PLI functions are also used to start running scripts on the Tcl interpreters and to pass control between Verilog and Tcl.

The Verilog tasks have access to arguments passed to the Tcl functions that invoked them, allowing information to be passed from Tcl to Verilog. The Verilog tasks also have the ability to control the return values of their Tcl counterparts, allowing information to be passed from Verilog to Tcl. PLI routines are also provided that allow direct sharing of information between Tcl and Verilog, as well as between different Tcl interpreters.

### 6 Interaction between Verilog and Tcl

At the core of the TCL\_PLI library are five PLI functions: `$tclInit`, `$tclExec`, `$tclGetArgs` and `$tclClose`.

`$tclInit` is used to create and initialize a new Tcl interpreter. It defines the new Tcl functions that will be used to invoke Verilog tasks, maps them to specific tasks, and defines how many arguments they will take.

`$tclExec` is used to pass control from Verilog to the Tcl server. It is used to launch a new script or to resume execution of a script that was stalled when the interpreter encountered a function that was mapped to a Verilog task. `$tclExec` will return under one of three conditions:

when an error occurs, when the script ends or when a function is encountered that is mapped to a Verilog task.

`$tclGetArgs` is used to access the argument values that were passed to an extended Tcl function. `$tclClose` is used to destroy a Tcl interpreter and free the resources associated with it.

The following code snippet shows how an interpreter is created and initialized in Verilog. The interpreter will have three extended Tcl commands (`b_write`, `b_read` and `b_wait_irq`) that are mapped to Verilog tasks. These commands will allow scripts running on the interpreter to write data on a bus, read data from a bus or wait for an interrupt to occur on the bus.

```

1 parameter
2   BUS_WRITE      = 1,
3   BUS_READ       = 2,
4   BUS_WAIT_IRQ   = 3;
5 initial
6   begin: processor_model
7     integer tcl_handle,
8     tcl_command, tcl_return_value;
9     // Initialize interpreter that
10    // knows about the three
11    // extended Tcl commands:
12    tcl_handle = $tclInit (
13      "processor", tcl_command,
14      tcl_return_value,
15      "b_write",   BUS_WRITE,   2,
16      "b_read",    BUS_READ,    1,
17      "b_wait_irq",BUS_WAIT_IRQ, 0
18    );
19    if (tcl_handle == 0)
20      begin
21        $display ("Init error.");
22        $finish;
23      end

```

On lines 1 to 4, three integer parameters are defined that represent the three extended Tcl commands in Verilog. Lines 7 and 8 define three variables that are used to communicate between Verilog and Tcl. `tcl_handle` will be used to store the handle of this interpreter. Each interpreter will have a unique handle. It is returned by the `$tclInit` PLI function and is used by all other TCL\_PLI functions to identify the interpreter that is being addressed. `tcl_command` will be used by TCL\_PLI to indicate to Verilog which extended Tcl function had been encountered while executing the script. It is also used to indicate the completion status of the Tcl script when execution of the script ends. `tcl_return_value` will be used by Verilog to communicate the return values of tasks back to the calling Tcl functions.

On lines 12 to 18, the call to `$tclInit` initializes the interpreter. It identifies the variables `tcl_command` and

`tcl_return_value` to TCL\_PLI. It also defines the extended Tcl functions `b_write`, `b_read` and `b_wait_irq`. `$tclInit` associates an integer value with each extended Tcl function and stores the number of arguments that each extended Tcl function should expect. `$tclInit` can take a variable number of arguments to allow definition of any number of extended Tcl functions.

The return value of `$tclInit` will contain the handle of the newly created Tcl interpreter. If the value is zero, this is an indication that an error occurred while creating and initializing the interpreter. The test on line 19 confirms that the call to `$tclInit` completed successfully.

The next code snippet indicates how a script is executed on a Tcl interpreter and how control is passed between Verilog and Tcl.

```

24   while ($tclExec (tcl_handle,
25     "example.tcl"))
26     begin
27       tcl_return_value = 0;
28       case (tcl_command)
29         BUS_WRITE:
30           bus_write (tcl_handle);
31         BUS_READ:
32           bus_read (tcl_handle,
33             tcl_return_value);
34         BUS_WAIT_IRQ:
35           bus_wait_irq;
36       endcase
37     end // while
38     if (tcl_command != 0)
39       begin
40         $display (
41           "Error in Tcl script!");
42         $finish;
43       end
44     $tclClose (tcl_handle);
45   end // processor_model

```

When `$tclExec` is encountered the first time (line 24), the Tcl server is idle. This is an indication that it should start executing the script "example.tcl". As mentioned earlier, `$tclExec` will return under one of three conditions: if an error occurs, if the script ends or if an extended Tcl function is encountered. A non-zero return value indicates that an extended Tcl function had been encountered. A return value of 0 indicates that the script has ended or that an error occurred. In the case where an extended Tcl function was encountered, the Tcl interpreter will stall until the next call to `$tclExec` informs it that the Verilog task has been executed.

The while loop (lines 24 to 37) will keep looping as long as the return value of `$tclExec` remains positive. This means that the simulator will keep executing Verilog

tasks as they are encountered in the Tcl script that is being executed. If `$tclExec` is called when the Tcl server is not idle, it assumes that execution of the current script should continue. `$tclExec` will check that the script name passed to it matches the name of the script being executed and will return an error value if this is not the case.

Not all Verilog tasks have meaningful return values. Therefore `tcl_return_value` is initialized to zero to ensure that an undefined value is not eventually returned to the Tcl interpreter (line 27).

Whenever `$tclExec` returns a non-zero value, the `tcl_command` variable will contain the integer value corresponding to the extended Tcl function that was encountered in the script. The case statement on line 28 is used to call the Verilog task associated with this Tcl function.

Once the Verilog task completes, the while loop continues with another call to `$tclExec`. `$tclExec` will again return when it encounters an extended Tcl command, reaches the end of the script or encounters an error. The result is that the while loop will cause the whole Tcl script to be executed, and Verilog tasks will be called in the order determined by the execution of the Tcl script.

When the while loop terminates, it is appropriate to check that no error occurred (line 38). When `$tclExec` returns zero (causing the while loop to terminate), the `tcl_command` variable will contain an exit code indicating either normal termination (end of script reached) or abnormal termination (due to an error in the script).

At this point, the Tcl interpreter can be deleted if it is no longer required (line 44). It should be noted that the same interpreter can be used repeatedly. Scripts can also be run on the interpreter from different points in the Verilog code. `TCL_PLI` will generate an error if an attempt is made to run multiple scripts on one interpreter simultaneously. Interpreter state information (like variable values and procedure definitions) is preserved until the interpreter is destroyed through a call to `$tclClose`.

The next code snippet shows the definition of the Verilog tasks that are called under control of the Tcl interpreter.

```

49 task bus_write;
50 input [31:0] tcl_handle;
51 integer address, data;
52 begin
53     $tclGetArgs (tcl_handle,
54         address, "i",
55         data, "i"
56     );
57     // Simulate the write cycle

```

```

58     // on the bus
59 end
60
61 task bus_read;
62 input [31:0] tcl_handle;
63 output [31:0] data;
64 integer address;
65 begin
66     $tclGetArgs (tcl_handle,
67         address, "i"
68     );
69     // Simulate the read cycle
70     // on the bus
71     data = data_read_from_bus;
72 end
73
74 task bus_wait_irq;
75 begin
76     @(bus_irq);
77 end

```

A Verilog task can gain access to the arguments of the Tcl function that invoked it. For this purpose the handle of the interpreter is passed to these tasks (line 50). A call to `$tclGetArgs` is used to transfer this information from Tcl to Verilog (line 53). `$tclGetArgs` can handle integer or string arguments, and will do the appropriate conversions.

The `bus_read` Verilog task illustrates how the return value of a Tcl function is set up in the Verilog task (lines 71 and 33). `TCL_PLI` assumes that the return value is an integer. The `bus_wait_irq` task illustrates a simple case where the Tcl interpreter can be stalled while waiting for an event in the simulation (line 76).

Following is a sample Tcl script that can be run in the Verilog example shown above. It illustrates how the execution order of the Verilog tasks are completely controlled from Tcl. In essence, the Tcl script is in complete control of the simulation in the same way that software controls the hardware on which it is run.

```

1 # Write to a register, wait
2 # for an interrupt and read back
3 # a cause:
4 puts "$::vname @ $::vtime: \
5 Writing 0xaa to address 0x05:"
6 b_write 0x05 0xaa
7 puts "$::vname @ $::vtime: \
8 Waiting for interrupt..."
9 b_wait_irq
10 puts "$::vname @ $::vtime: \
11 Interrupt received"
12 puts [format "Address 0x05 now \
13 contains %x" [b_read 0x05]]
14
15 # Write to another register,

```

```

16 # then poll until bit 1 changes:
17 puts "$::vname @ $::vtime: \
18 Writing 0xff to address 0x0a:"
19 b_write 0x0a 0xff
20 set bit_1 0x0
21 while {$bit_1} {
22 # Read 0x0a and check the LSB:
23 set bit_1 \
24 [expr [b_read 0x0a] && 0x01]
25 puts "$::vname @ $::vtime: \
26 Bit 1 value is $bit_1"
27 }
28 puts "$::vname @ $::vtime: \
29 Value changed!"

```

```

21 break;
22 case TC_ADDCOMMAND:
23 /* Add new commands to
24 interpreter */
25 break;
26 case TC_LINKVARS:
27 /* Link with Verilog
28 variables */
29 break;
30 case TC_SHAREVARS:
31 /* Link with variables from
32 other interp */
33 }
34 }

```

The variables `vname` and `vtime` are defined by the `TCL_PLI` library. `vname` contains the name of the interpreter (passed as the very first argument to `$tclInit`). It is useful to determine the source of a message. `vtime` contains the current simulation time, which is very useful for tracing simulator messages back into waveforms.

## 7 Behind the scenes

As mentioned previously, the Tcl interpreter is run on a separate thread from the Verilog simulation. A call to `$tclInit` will cause a secondary thread to be created, on which the Tcl server will run. The Tcl server creates and initializes a new Tcl interpreter, and then enters a loop in which it waits for and executes commands received from the PLI functions.

The following C code snippet is a simplified version of the command loop in the Tcl server.

```

1 /* Wait for and service requests
2 to run scripts */
3 runServer = 1;
4 while (runServer)
5 {
6 /* Pass control to the Verilog
7 thread */
8 sem_post (&t->t2v);
9 /* Wait for an instruction from
10 the Verilog thread */
11 sem_wait (&t->v2t);
12 switch (t->serverCommand)
13 {
14 case TC_RUNSCRIPT:
15 t->serverStatus =
16 tclServer_runScript (t);
17 break;
18 case TC_CLOSE:
19 runServer = 0;
20 t->serverStatus = TS_DONE;

```

Once the Tcl server has initialized, it enters the command loop and immediately posts the `t2v` semaphore to the PLI to indicate that it is ready to accept a command. It then waits for the `v2t` semaphore. Upon receipt of the `v2t` semaphore, it examines the `serverCommand` member of its defining structure to determine what command was issued and executes the command. When the command is completed, the loop starts again.

On the Verilog thread, a PLI function sends a command to the Tcl server by setting up the `serverCommand` member of the server's defining structure and then posts the `v2t` semaphore. The PLI function then waits for the `t2v` semaphore as an indicator that control is being passed back to Verilog.

The following sequence will take place if the Tcl script in the previous example is executed: When `$tclExec` is called from the Verilog simulation the first time (Tcl server is idle), it will instruct the Tcl server to start executing the script, and then wait for the `t2v` semaphore. The Tcl server will execute lines 1 through 5 of the script. When it reaches line 6, which contains the extended command `b_write`, it will call the function `tclServer_verilogCall`. This function is called for all extended commands that are mapped to Verilog tasks. `tclServer_verilogCall` will save the relevant information in its defining structure, post the `t2v` semaphore and then wait for the `v2t` semaphore.

When `$tclExec` receives the `t2v` semaphore, it will synchronize linked variables and return to Verilog. This allows the simulator to enter the while loop in which it executes the tasks associated with extended Tcl functions (lines 24 to 37 of the Verilog example). When the task associated with `b_write` completes, `$tclExec` is again called. This time the Tcl server is not idle, so `$tclExec` assumes that execution of the script should resume. It will synchronize linked variables, post the `v2t` semaphore and wait for the `t2v` semaphore.

`tclServer_verilogCall` will receive the `v2t` semaphore and return, allowing execution of the Tcl

script to continue. This process will repeat itself until the script completes. When this happens, the Tcl server will indicate this to the PLI when posting the `t2v` semaphore. This time, when `$tclExec` returns, the Verilog while loop will terminate.

The following C code snippet shows a simplified version of `tclServer_verilogCall`, the function that is called by the Tcl interpreter when it encounters an extended command.

```

1 int tclServer_verilogCall (...)
2 {
3     /* Store the command value for
4     the Verilog thread */
5     t->commandValue =
6     command->value;
7     /* Store the argument array
8     and count in the interpreter
9     struct to make it accessible
10    to tclGetArgs: */
11    t->argc = argc;
12    t->argv = argv;
13    /* Semaphore verilog */
14    sem_post (&t->t2v);
15    /* At this point control has
16    been passed back to Verilog,
17    where the functionality is
18    being simulated. Once done,
19    the main thread will
20    semaphore this thread to
21    continue. */
22    /* Wait for semaphore from
23    verilog */
24    sem_wait (&t->v2t);
25    /* Set up the return value */
26    sprintf (message, "%d",
27            t->retVal);
28    Tcl_SetResult (t->interp,
29                  message, TCL_VOLATILE);
30    return TCL_OK;
31 }

```

It is important to note that, even though `TCL_PLI` is multi-threaded, and that every interpreter is run on a dedicated thread, the essential single threaded nature of Verilog simulations is maintained. Only one call to `$tclExec` can be reached in the Verilog simulation at any given time. The Verilog simulation stalls until this call returns, which will happen when the Tcl interpreter calls `tclServer_verilogCall` or when the script completes. `tclServer_verilogCall`, on its part, will only return when the next call to `$tclExec` is encountered in the Verilog simulation. This means that, even though many Tcl scripts may be in the process of execution at any given moment in time, only one of them

or the Verilog code itself will be running at that moment. All event scheduling and execution order is still under the control of the simulator.

It should also be noted that the Tcl server executes scripts in zero simulation time. Simulation time doesn't advance for the duration of a call to `$tclExec`. Simulation time advances normally while the Verilog tasks invoked from Tcl are executed

## 8 The TCL\_PLI library

Due to a lack of space, we will only give a brief overview of the PLI functions available in the `EFI_PLI` library. The source distribution will contain a more complete set of documentation. The `$tclInit`, `$tclClose`, `$tclExec` and `$tclGetArgs` PLI functions have already been discussed in detail and are not listed again in this section.

### 8.1 \$tclLinkVariables and \$tclShareVariables

`$tclLinkVariables` allows direct sharing of variables between Verilog and Tcl. It links a list of Verilog variables with a list of Tcl variables. The `TCL_PLI` library will then automatically keep these variables synchronized until the interpreter is deleted. `$tclLinkVariables` has support for integer and string variables, and can mark variables as read-only in the Tcl interpreter, meaning that they can be modified in Verilog, but not in Tcl.

`$tclShareVariables` allows direct sharing of variables between two different Tcl interpreters, without any connection to Verilog. After a call to `$tclShareVariables`, the list of Tcl variables in both interpreters will be automatically synchronized by the `TCL_PLI` library, until one of the interpreters is deleted.

### 8.2 \$tclSetMCD and \$tclAddMCD

`$tclSetMCD` and `$tclAddMCD` allow the Tcl interpreter access to multi-channel descriptors in the Verilog simulation. This allows the user to redirect messages from the Tcl interpreter into log files that also record messages directly from the simulation, which is critical for preserving the order in which messages were generated. Any Verilog multi-channel descriptor can be associated with a Tcl interpreter. If an interpreter has an associated MCD, its built-in `puts` command will be modified, causing all output to `stdout` to be redirected to the multi-channel descriptor.

This makes it very easy to open a log file and set up an MCD that will cause all messages from the simulation

(both from Verilog and Tcl) to be printed to stdout as well as being recorded in a log file.

### 8.3 \$tclSetErrorReg

\$tclSetErrorReg allows the user to identify one register in the Verilog simulation that is linked to any error occurring in any interpreter or in TCL\_PLI. If any error occurs, the value of this register is changed, allowing the simulation to immediately react to the error.

### 8.4 \$tclWarnOnX

Like any software language, Tcl has no concept of X or Z values. The default behavior of TCL\_PLI is to cause execution of the Tcl script to be terminated under any of the following conditions: If the return value of an extended Tcl function is X or Z, or if any linked variable is X or Z at a point where it is evaluated in the script.

It is the opinion of the authors that this is correct and desirable behavior. In their opinion, X and Z values should never propagate into the software domain, because software doesn't know how to deal with these values. However, some users of TCL\_PLI do not share this opinion, and hence the existence of \$tclWarnOnX. Calling \$tclWarnOnX will cause TCL\_PLI to print a warning message under any of the previously described conditions. Execution of the script will continue. If the variable in question was an integer, its Tcl value would be considered to be zero. If it was a string, its Tcl value would be "Zz".

Note that only a warning message is generated. In a simulation that prints many messages, this message can easily scroll off the screen before being noticed. The misinterpretation of the Verilog value in Tcl can ultimately cause all sorts of strange behavior and cause much heartache to the user who decided to use \$tclWarnOnX.

## 9 Example: PCI\_TCL

We have designed a module that instantiates Synopsys LMC source models for a PCI master and a PCI slave together with two Tcl interpreters. The tasks supplied with the PCI models were mapped to extended Tcl functions, allowing us to execute Tcl scripts that interact with other devices on a PCI bus.

The Verilog code causes one of the Tcl interpreters to start executing a script when it senses an interrupt on the PCI bus. This allows easy modeling of interrupt service routines written in Tcl. Execution of a Tcl script can be linked to an interrupt by simply waiting in the Verilog code for the interrupt to occur before entering the while loop calling \$tclExec.

The two interpreters in the PCI master have to compete for access to the bus. This is accomplished through a simple gating mechanism that checks whether the bus is busy before calling a task that will start a transaction on the bus. If the bus is busy, it simply waits for the current transaction to complete before starting the new one. This arrangement models actual software behavior very well, where several processes running on a CPU have to compete for access to the PCI bus, with no guarantee on the order in which accesses will take place.

The PCI\_TCL module allows both interpreters to load data files directly into the LMC slave memory (in zero simulation time). Data can also be dumped from the slave memory to a file. Both interpreters can execute memory or I/O transactions on the bus. For bursting commands, we use two extended Tcl functions, the one executes the address phase of a burst while repeated calls to the other executes one data cycle per call. We wrote an encapsulating Tcl procedure that takes an address, byte count and byte array and does a corresponding burst read or write on the PCI bus by appropriately calling the underlying extended Tcl functions.

The PCI\_TCL module allows us to do extensive testing of any PCI based device without writing a single line of Verilog code for the test bench. We have also developed a library of Tcl procedures that simplifies tasks like configuration of PCI devices.

## 10 TCL\_PLI in practice

The TCL\_PLI library has been in use at EFI now for about 18 months. We have used this approach to verify a 600k gate design and we're currently using it on two designs, both of which are larger than one million gates. All these designs are PCI based ASICs. Using the PCI\_TCL module has allowed us to write elaborate scripts that interact with the device under test in very much the same way as software would eventually interact with the real hardware.

We also used Tcl interpreters in modules that interact with other ports on the device under test. In each case, we wrote Verilog tasks that know how to interact with a port at a low level. The higher level behavior of the test module is then controlled by a Tcl script. This allows us to radically change the behavior of test modules by running differing Tcl scripts on them.

In our test benches we typically have a master Tcl interpreter that controls all test modules that interact with the device under test. It determines which Tcl scripts get executed when and on what module. This approach gives us centralized control over an extremely configurable test bench.

## 11 Pitfalls

There are a number of pitfalls to watch out for when using the TCL\_PLI library. Most important are some issues related to the simulator. We have been using the TCL\_PLI library extensively with Synopsys' VCS simulator. We started off with VCS version 4.0.3. This version had the restriction that we had to compile through C code. VCS allows compilation through C, assembler or directly to object code. Compilation through C is the slowest, but Synopsys informed us that we had to compile through C if using multi-threaded PLI. Not doing this caused immediate core dumps, even when the TCL\_PLI library was linked in, but not used. We are currently using VCS version 5.0.1. This version does allow us to compile directly to assembler or object code. It should also be mentioned that, besides the penalty of slower compile times, VCS 4.0.3 worked perfectly well with TCL\_PLI. Following is a list of compile time options for VCS that are needed when using the TCL\_PLI library:

- `-Xstrict`: Prevents VCS from using resources that are used by the multi-threading library.
- `-lpthread -lposix4`: Link with the Posix threading support libraries.

We have also used TCL\_PLI with Cadence Verilog-XL version 2.5. We do not use Verilog-XL very much, but we haven't yet encountered any problems with it.

One of the most common problems encountered by first-time users, is that they forget to assign a default value for the return value register in the while loop that executes the Tcl script. TCL\_PLI has no way to distinguish whether a return value will be used, and always complains if a return value is X or Z. Therefore a default value should always be assigned.

A problem that we anticipated, but have not encountered too frequently, is with Tcl bugs in long running simulations. Because Tcl is an interpreted language, a syntax error is only detected when the interpreter encounters the statement that contains the error. We were concerned that we would often start off a simulation that would run very long, only to have it die near the end due to a bug in the Tcl script. The way that we have managed this problem, is to develop scripts incrementally, thereby ensuring that long-running simulations will be done with proven Tcl code.

## 12 Conclusion

The solution that we devised more than fulfilled our expectations: We managed to significantly reduce time spent recompiling test benches. Our test benches are a lot simpler, and consist of modular, reusable modules that are easy to maintain. New tests are implemented in new,

separate scripts, eliminating the problem where addition of new tests caused existing tests to break.

One of the greatest advantages was not anticipated: The Tcl scripts themselves are often reusable. When we do module level testing, we partition functionality between Tcl and Verilog so that the Tcl scripts will be reusable in system level testing. As an example, a Tcl interpreter interacting with a module, will use extended Tcl functions that take the same arguments and return values as matching functions in the PCI\_TCL module, even though it is not interacting with a PCI bus. This way the scripts developed for testing the module can be rerun without modification in system level tests.

In one case, we were able to reuse a complicated Tcl script written for a project that was cancelled! It was initially written to test a memory controller by doing random reads and writes and automatically verifying data integrity. When we later had to design logic that had to access system memory through a PCI bus, we reused the script without modification.

Another unexpected benefit was that Tcl suddenly became a mainstream language in the EDA world. When Synopsys decided that they were going to integrate Tcl interpreters into their synthesis and static timing analysis tools, we could immediately take advantage of this, because our whole design team was already familiar with Tcl.

We are currently investigating the feasibility of porting our Tcl scripts to real hardware. This will enable us to run our verification suite on our ASICs when they return from the foundry. For PCI based devices, this seems like it will be a simple task. We just need to write simple C functions that map our PCI read and write commands to real PCI read and write commands.

All in all, our decision to switch to scripting languages to manage test bench complexity proved to be a very good one. In our opinion, choosing Tcl as the scripting language meant the difference between success and failure.

[1] John Ousterhout, "Tcl and the Tk Toolkit," p. xvii, Addison-Wesley, 1994.