Design Advantage™
We build them for YOU.℠
Comit Systems, Inc. Newsletter Volume 3 Number 3 May – Jun. 2000

Supercharging Your Simulation with a Verilog-Tcl Bridge

Clever coding can enhance your design options and speed up time to completion.

By Venkat Iyer

Integrating a scripting language into your simulation environment adds tremendously to testing functionality by providing a mechanism to observe actual output, and not just waveforms. Tcl is an extension scripting language that can be integrated with Verilog, forming a Verilog-Tcl bridge, providing access to your design in a standardized Verilog simulation environment. A "Tcl'ed" Verilog, requires significantly less effort to design, build and deploy than with a language like C.

Dynamic Testing: Without changing your Verilog, Tcl provides a way of doing configurable testing. Tcl's interpreter lets you process files generated by other flows, redefine procedures, unload packages and reload new ones. You can often build a professional-looking user interface within a day, with Tk. Tk delivers an accurate feel of the simulation, because the user is able to sit at the simulation and control it manually. For example, when testing your PCI target, you write a PCI-transaction dialog-box in Tcl, which allows you to do various PCI bus operations (with options). You could then click on the operations and see any of the results immediately.

A Verilog-Tcl bridge is necessary for each platform that you use. This allows you to use all of your Tcl scripts, unmodified, in all of those simulation environments. It is not necessary to rebuild the executable or recompile your Tcl scripts: since it works on the post-implementation Verilog, the bridge isn't specific to an FPGA vendor. This type of bridge has been used successfully with designs targeting various FPGA vendors, including Xilinx and Altera.

If a vendor's tool already has Tcl built into it, an interface Verilog XL might be necessary. Since Tcl and the programming language interface (PLI) in Verilog are both targeted towards C, C is the obvious choice for building this Verilog-Tcl bridge. Typically we use the most basic level of simulation interaction intially,and then build higher-level layers. Performance is usually not an issue—the bottleneck is usually the simulator. Though the compilation and linking options are radically different between simulators, the general idea is the same.

Issues with Tcl and Verilog: When builidng a simple, low-level Tcl application-programming interface (API), the main issues generally center around the control flow. As Verilog XL defines `main` and you can't override it, you need to do the Tcl/Tk initialization at a different time in the flow. There are two ways to handle initialization issues. The easier process is to check in all of the Tcl PLI calls and then do the initialization, if it hasn't as yet happened. Tcl doesn't yet (as of V 8.3) support threads very well (especially Tk), so the Tcl interpreter as well as the Verilog simulator need to be run in the same thread. This means that you have to find a way to keep the user interface interactive while letting the simulation run. In your simulation, you will need to call the Tcl `update` command often—the point at which the GUI commands are recognized and the GUI is refreshed.

The PLI: Verilog calls Tcl functions via the PLI. With Verilog XL, you need to first write up the `veriuser.c` (see Listing 1). Tcl provides a command `eval`, which can call any other command. We define one PLI task called `tcleval`, which will allow us to do anything inside `tcleval` as a C function. This function takes one string as the first parameter and passes it to the Tcl interpreter. Other parameters can be passed too; accessible through the PLI routines.

Listing 1

```
s_tfcell veriusertfs[] =
{
   { usertask, 0, 0, 0, tcleval, 0, "$tcleval", 0 },
   {0} /*** final entry must be 0 ***/
};
```

A set of commands, translating to Verilog, are added to Tcl, and are defined in the interpreter at initialization time (see Listing 2). Each C routine uses PLI routines to translate the Tcl command for the Verilog simulator. As many routines as needed may be added from the Verilog PLI. We use a simple subset for this task (see table, below).

Listing 2

```
void tcleval()
{
  static Tcl_Interp *my_interp;
  char *cmd;

  if (my_interp == NULL)
  {
            my_interp = Tcl_CreateInterp();

   ....all tcl init, tk init and application init....

            if (Tcl_CreateObjCommand(my_interp, "verprint", verprint,
                  NULL, NULL) == NULL)
            {
               tf_error("Couldn't add command verprint");
               return;
            }
   .... add commands verputp, vergetp, vertime, verprint ....
  }
  if (tf_nump() < 1)
  {

            tf_error("$tcleval requires at least one string"
                  "arg for the command");
            return;
  }

  if (tf_typep(1) != tf_string)
  {
            tf_error("First argument to tcleval must be a string");
            return;
  }
  cmd = acc_fetch_tfarg_str(1);
  if (Tcl_Eval(my_interp, cmd) != TCL_OK)
  {

            tf_error("tcleval error found\n, %s\n",
                        Tcl_GetVar(my_interp, "errorInfo",
TCL_GLOBAL_ONLY));
            Tcl_ResetResult(my_interp);
  }
}
```

Routine Name	*Routine Description*
Verprint	Use io_print to print a string so that the message will show up in the simulator logs as well as the standard output
Vertime	The current simulation time (this ignores the high 32 bits)
Verputp	Use tf_putp to put a value to one of the parameters to this routine
Vergetp	Use tf_getp to get the value of one of the parameters to this function

Commands are added to the simulator at Tcl initialization, in the `tcleval` function. The command `vergetp` (see Listing 3) due to the use of Tcl objects and error checking, looks quite complex.

Listing 3

```
int vergetp(ClientData cd, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int i;

  if (objc != 2)
  {
            return setTclError(interp, "%s takes exactly only one integer "
              "as the parameter to get the value of.",
                                        Tcl_GetString(objv[0]));
  }
  if (Tcl_GetIntFromObj(interp, objv[1], &i) != TCL_OK)
            return TCL_ERROR;
  else
  {
            Tcl_Obj *io = Tcl_NewIntObj(tf_getp(i));
            if (!io)
              return setTclError(interp, "Couldn't create object "
                "for int return");
            Tcl_SetObjResult(interp, io);
            return TCL_OK;
  }
}
```

You would write all your design specific Tcl code in one file or a set of files (see Listing 4).

Listing 4

```
initial begin
        $tcleval("source counter_tb.tcl");
end
```

Then you could call $tcleval anytime to do your tcl tasks. For example, in the code shown here, the counter_tb.tcl had one proc clockedge defined that incremented the second parameter passed in (see Listing 5).

Listing 5

```
integer i;
reg clk;
always @(clk)
begin
        $tcleval("clockedge", clk, i);
        $tcleval("update");
end
```

The first argument to $tcleval is the name of the procedure, the parameters that follow are just signal names that Tcl will access as described in the next section. If you have a Tk GUI built, then make sure you call `update` or the GUI won't display correctly and won't respond. (see Listing 6).

A procedure that was defined in the file sourced above (see Listing 5), counter_tb.tcl is enacted prior to the procedure named "clockedge" (see Listing 6). The important things to note are that vergetp uses numbers starting from 2, because parameter 1 will be the script passed in Verilog to $tcleval. You can still use the standard Tcl puts to print things onto the screen, but remember that this will not go into the simulation log. Use verprint where possible.

Listing 6

```
proc clockedge {} {
  set clk [vergetp 2]
  set i [vergetp 3]
  verprint "Clock is $clk and i is $i at time [vertime]"
  incr i
  verputp 3 $i
}
```

A Sample Application: We used an application that involved designing a real-time video-processing chip with an RGB input and an RGB output. The true test of the chip was to see the input image and the output image simultaneously, and to monitor whether the chip was processing correctly. The simulation was taking a long time to run, so the user wanted to be able to see the output image as it was being generated, and to debug the image by simultaneously zooming to various parts of the image, visualizing pixel values, or other parameters.

Building a Verilog-Tcl bridge allowed the passing of data from the Verilog simulation to Tk, where it was displayed on the screen and updated pixel-by-pixel as the simulation proceeded. You could also control the simulation by either pausing or quitting it directly from the GUI (which also displayed the current time). Additionally, to use this with your FPGA design, use the FPGA vendor's tool to generate Verilog for the implemented FPGA design and, finally, use Verilog-XL with the bridge to exercise the design.

The source code of the Verilog-Tcl bridge, along with building instructions, is downloadable free of charge for a limited time at www.comit.com. Check under FREE STUFF. ■