

IF YOU DESIGN IN VERILOG, USING THE HDL'S PROGRAMMING-LANGUAGE INTERFACE IS VALUABLE FOR INVOKING A C FUNCTION FROM VERILOG. THIS ARTICLE PROVIDES THE INFORMATION YOU NEED TO START WRITING USEFUL PLI ROUTINES.

A Verilog programming-language-interface primer

DESIGNERS HAVE EMPLOYED HDLs for more than a decade, using them to replace a schematic-based design methodology and to convey design ideas. Verilog and VHDL are the two most widely used HDLs for electronics design. Verilog has approximately 35,000 active designers who have completed more than 50,000 designs using Cadence's (www.cadence.com) Verilog software suite (www.cadence.com/press_box/releases/verilogXL-desktop.html).

Even with Verilog's success, many seasoned Verilog users still perceive its programming-language interface (PLI) as a "software task" (www.angelfire.com/ca/verilog, www.europa.com/~celiac/pli.html, **Reference 1**). A step-by-step approach helps you "break the ice" when writing PLI functions. By learning the essentials of PLI design without getting bogged down by too many details, you will acquire a basic knowledge of PLI that you can immediately use.

WHY SHOULD YOU USE A PLI?

A PLI gives you an application-program interface (API) to Verilog. Essentially, a PLI is a mechanism that invokes a C function from Verilog code. People usually call the construct that invokes a PLI routine in Verilog a "system task" or "system function" if it is part of a simulator and a "user-defined task" or "user-defined function" if the user writes it. Because the essential mechanism for a PLI remains the same in both cases, this article uses the term "system call" to indicate both constructs. Examples of common system calls that most Verilog simulators include are \$display, \$monitor, and \$finish.

You use a PLI primarily for doing tasks that would otherwise be impossible to do using Verilog syntax. For example, IEEE Standard 1364-1995 Verilog has a

predefined construct for doing a file write, (\$fwrite, which is another built-in system call written using a PLI), but it does not have one for reading a register value directly from a file (**Reference 2**). More common tasks for which PLI is the only way to achieve the desired results include writing functional models, calculating delays, and getting design information. (For example, no Verilog construct gives the instance name of the parent of the current module in the design hierarchy.)

To illustrate the basic steps for creating a PLI routine, consider the problem in **Listing 1**. This problem is much simpler than a real-life problem you solve using PLI, however it shows many of the basic steps you use to build a PLI routine. When you run the Verilog in the **listing**, it should print the value of the register as 10 at time 100 and 3 at time 300. You can think of creating a PLI routine as a two-step process: First, you write the PLI routine in C; then, you compile and link this routine to the simulator's binary code.

WRITING A PLI ROUTINE

The way a PLI routine interfaces with the simulator varies from simulator to simulator, although the main functions remain the same. This article dis-

LISTING 1—VERILOG PROGRAM TO READ A REGISTER'S VALUE

```
module my_module;
  reg [3:0] r1;
  initial
  begin
    r1 = 4'ha;
    #100 $print_reg(r1);
    #100 r1 = 4'h3;
    #100 $print_reg(r1);
    #100 $finish;
  end
endmodule
```

cusses the interfacing mechanisms of the two most popular commercial simulators, Cadence's Verilog-XL (Reference 3) and Synopsys' (www.synopsys.com) VCS (Reference 4). Although other commercial simulators support PLI, their interfacing mechanisms do not differ significantly from these two. Over the years, Verilog PLI has evolved into PLI 1.0 and Verilog Procedural Interfaces (VPI) (see sidebar "A short history of Verilog PLI"). This article covers only PLI 1.0. Despite the differences in interfacing parts and versions, you can break down the creation of a PLI routine into four main steps.

Step 1: Include the header files

By convention, a C program implements a PLI routine in a file *veriuser.c*. Although you can change this name, the *vconfig* tool assumes this default name while generating the compilation script in a Verilog-XL environment. For now, assume that you keep the PLI routine in the file *veriuser.c*.

In a Verilog-XL environment, the file *veriuser.c* must start with the following lines:

```
#include <veriuser.h>
#include <vxl_veriuser.h>
```

In a VCS environment, the file must start with:

```
#include <vcsuser.h>
```

A SHORT HISTORY OF VERILOG PLI

Verilog started as a proprietary product from Gateway Design Automation Inc, a company that Cadence (www.cadence.com) subsequently bought. According to people close to the project at the time, the requirements for a programming-language interface (PLI) in Verilog came up early during designs. One of the problems facing them was that a single workstation could not cope with the simulation load of an entire design. The need for load balancing among multiple workstations soon became a necessity. Designers used a PLI as a tool to solve this problem.

People designed the first generation of PLI routines, called TF or *tf_* routines, to work only with the parameters passed to them. It was soon apparent that you could

LISTING 2—USING CHECKTF IN VERILOG

```
int my_checktf() {
    if (tf_nump() != 1) {
        if_error("Usage:$print_reg(register_name);");
    }
    if (tf_typep(1) != tf_readwrite) {
        if_error("The argument must be a register type\n");
    }
}
```

These header files contain the most basic data structures of the Verilog PLI that the program will use.

Step 2: Declare the function prototypes and variables

A PLI routine consists of several functions. Just as you would for a normal C program, you should place the prototype declarations for the functions before the function definitions. For this case, the function appears as:

```
int my_calltf(), my_checktf();
```

In the above function, the *int* prototype declaration implies that these functions return an integer at the end of their execution. If there is no error, the normal return value is 0. However, if the functions are in separate files, you should declare them as external functions with:

```
extern int my_calltf(), my_checktf();
```

A typical PLI routine, like any other C program, may need a few other house-keeping variables.

Step 3: Set up the essential data structures

You must define a number of data structures in a PLI program. A Verilog

simulator communicates with the C code through these variables. Open Verilog International (OVI) (www.ovi.org/pubs.html), an organization for standardizing Verilog, recommends only one mandatory data structure, *veriusertfs*. However, the exact number and syntax of these data structures vary from simulator to simulator. For example, Verilog-XL requires four such data structures and their functions for any PLI routine to work; VCS needs none of them and instead uses a separate input file.

The main interfacing data structure for Verilog-XL is an array of structures or a table called *veriusertfs*. Verilog-XL uses this table to determine the properties associated with the system calls that correspond to this PLI routine. The simulator does not recognize any names other than *veriusertfs*. Each element of *veriusertfs* has a distinct function, and you need all these functions to achieve the overall objective of correctly writing the PLI routine. The number of rows in *veriusertfs* is the same as the number of user-defined system calls plus one for the last entry, which is a mandatory 0. In this case, the *veriusertfs* array should look like:

```
s_tfcell veriusertfs[] = {
    {usertask, 0, my_checktf, 0, my_calltf,
    0, "$print_reg"},
    {0} /* Final entry must be zero */
}
```

The first entry, *usertask*, indicates that the system call does not return anything. It is equivalent to *procedure* in Pascal or *function returning void* in C.

In the previous data-structure, *my_checktf* and *my_calltf* are the names of the two functions that you use to implement the system call *\$print_reg*. These names are arbitrary, and you can replace them with other names. The function *my_checktf* is generally known as a *checktf* routine. It checks the validity of the passed parameters. Similarly, *my_calltf*, which you usually call *calltf* routine, performs the main task of the system call. The positions of these names in *veriusertfs* are very important. For example, if you want to use *my_checktf* or any other name as a checking function, it must be the third element. The function *veriusertfs* provides options for a few other user-defined functions that you will not use in this routine. A zero replaces

LISTING 3—IMPLEMENTING \$PRINT_REG IN VERILOG-XL

```
int my_calltf(), my_checktf();
char *veriuser_version_str;

int (*endofcompile_routines[])() = {0};

bool err_intercept(level, facility, code)
int level; char * facility; char *code;
{ return (true); }

s_tfccl veriusertfs[] = {
{usertask, 0, my_checktf, 0, my_calltf, 0, "$print_reg"},
{0} /* Final entry must be zero */
};

int my_checktf() {
    if (tf_nump() != 1) {
        tf_error("Usage: $print_reg(register_name);\n");
    }
    if (tf_typep(1) != tf_readwrite){
        tf_error("The argument must be a register type\n");
    }
}

int my_calltf(){
    io_printf("$print_reg: Value of the reg=%h at time=%d\n",
        tf_getp(1), tf_gettime());
}
```

any function that you do not use. Therefore, the second, fourth, and sixth elements in the entry are zeroes. **Table 1** summarizes the objectives of each entry in a row of *veriusertfs*. If there are additional system calls, you need to define a separate entry in *veriusertfs* for each one.

Verilog-XL also needs the following variables or functions:

```
char *veriuser_version_str;
int (*endofcompile_routines[])();
bool err_intercept();
```

The first variable, *veriuser_version_str*, is a string indicating the application's user-defined-version information. A *bool* (Boolean) variable is an integer subtype with permitted values 0 and 1. In most cases, you can use Cadence-supplied default values for these variables or functions.

In VCS, instead of a table, you use the equivalent information in a separate file, which you usually call *pli.tab*. This name is also user-defined. In the current example using \$print_reg, the contents of this file are:

```
$print_reg check=my_checktf
call=my_calltf
```

Step 4: The constituent functions

With the prototype declarations in place, you are ready to write the two functions, *my_checktf()* and *my_calltf()*, which constitute the main body of the PLI application.

As previously discussed, a *checktf* routine checks the validity of passed parameters. It is a good practice to check whether the total number of parameters is the same as you expect, and whether each parameter is required. For example, in this case, you expect the program to pass only one parameter of type register. You do this task in the function *my_checktf()* (**Listing 2**).

The functions starting with *tf_* are the library functions and are commonly known as utility routines. The library functions in the previous function and their usage are *tf_nump()*, which determines how many parameters you are passing, and *tf_typep()*, which determines the parameter type by its position in the system call in the Verilog code. In this case, the system call is *\$print_reg*.

Thus, *tf_typep(1)* gives the type of the first parameter, *tf_typep(2)* gives the type of the second parameter, and so on. If the parameter does not exist, *tf_typep()* returns an error. (In this case, *tf_typep(2)* does not exist.) In the current example, you expect the program to pass a register value as a parameter. Therefore, the type should be *tf_readwrite*. If a wire is the expected parameter, the type should be *tf_readonly*. To facilitate error-condition checking, it is a good idea to first check the number of parameters and then to check their types. The *tf_error()* function prints an error message and signals the simulator to increment its error count. These library functions and constants are parts of the header files that you have included at the top of the file in Step 1.

A *calltf* function is the heart of the PLI routine. It usually contains the main body of the PLI routine. In this case, it should read the value of the register and then print this value. The following code shows how you can accomplish this job:

```
int my_calltf(){
    io_printf("$print_reg: Value of the
reg=%x at time=%d\n",
        tf_getp(1), tf_gettime());
}
```

In the above code, *io_printf()* does the same job as *printf()* in C, printing the value in standard output. Additionally, the function prints the same information

LISTING 4 - COMPILED VERILOG-XL RUN WITH A PLI ROUTINE

```
Compiling source file "test.v"
Highest level modules:
my_module
$print_reg: Value of the reg=10
$print_reg: Value of the reg=3
L9 "test.v": $finish at simulation time 400
15 simulation events
```

LISTING 5 - IMPLEMENTING \$INVERT IN VCS

Listing 5 - Implementing \$invert in VCS

```
#include "vcsuser.h"

int invert_calltf(), my_checktf();
void move();

int my_checktf() {
    if (tf_nump() != 1) {
        tf_error("Usage: $invert(register_name);\n");
    }
    if (tf_typep(1) != tf_readwrite) {
        tf_error("Argument must be a register type\n");
    }
}

int invert_calltf() {
    char *val_string;
    /* Temporarily holds the value of the reg */

    /* Step 1 of the algorithm */

    val_string = (char *) malloc(tf_sizep(1)+1);
    /* +1 to accommodate the null char at the end */
    strcpy(val_string, tf_strgetp(1, 'b'));

    /* Step 2 of the algorithm */

    move('1', '2', val_string);
    move('0', '1', val_string);
    move('z', 'x', val_string);
    move('2', '0', val_string);

    /* Step 3 of the algorithm */

    io_printf("$invert: %s --> %s at time %d\n",
        tf_strgetp(1, 'b'), val_string, tf_gettime());
    tf_strdelputp(1, tf_sizep(1), 'b', val_string, 0, 0);
}

void move(from, to, in_str)
char from, to, *in_str;
{
    int i=0;
    while(*(in_str+i)) {
        if (*(in_str+i) == from)
            *(in_str+i) = to;
        i++;
    }
}
```

in the Verilog log file. The function *tf_getp()* gets the register's integer value. The function *tf_gettime()* returns the current simulation time and needs no input parameter. *Calltf* is the most complicated function in a PLI routine. It often runs for several hundred lines.

Now that you have created the two main functions, you need to put them into one place. **Listing 3** shows how to implement \$print_reg in a Verilog-XL environment.

The next task is making the Verilog

simulator understand the existence of your new system call. To accomplish this task, you must compile the PLI routine and then link it to the simulator's binary. Although you can manually integrate the PLI code with the Verilog binary by running a C compiler and merging the object files, it is more convenient to use a script. In Verilog-XL, a program called *vconfig* generates this script. The default name for this script is *cr_vlog*. While generating this script, the program *vconfig* asks for the name of the compiled Verilog that you prefer. It also asks whether to in-

clude model libraries, which are PLI code, from standard vendors. For most of these questions, the default answer, which you input if you press return without entering anything, is good enough unless you have a customized environment. At the end, *vconfig* asks for the path for your *veriusers.c* files. Once you generate the script *cr_vlog*, you need only to run the script to generate a customized Verilog simulator that can execute the new system call.

A sample run of the compiled Verilog using this PLI routine produces the out-

put with a Verilog-XL simulator (**Listing 4**).

MODIFYING THE VALUE OF A REGISTER

You use a PLI is to read design information and to modify this information in the design database. The following example shows how you can do this modification. Create a system call, \$invert, to print the value of a register (as in the previous example), bitwise invert the content, and print this updated value. Many ways exist to invert a value of a binary number. By using a straightforward al-

TABLE 1—PLI DATA STRUCTURES FOR VERILOG-XL

Element number	Name	Type	What it does
1	Ustask/userfunction	Integer	Indicates whether the system call returns a value.
2	Data	Integer	Reuses the same function for multiple system calls. It is an argument passed to the other functions described below.
3	Checktf	Function returning integer	Checks the vailidity of the arguments passed to the system call.
4	Sizetf	Function returning integer	Returns the size (in bits) of the returned value for a user-defined function.
5	Calltf	Function returning integer	Is the main body of a user application.
6	Misc tf	Function returning integer	Triggers a task when a predefined event occurs; mainly used for "call-backs."
7	System call name	String	Denotes the system call name. It must start with a \$.

gorithm to do the inversion, as the following list shows, helps you gain more insight into the PLI mechanism and library functions:

- read the content of the register as a string;
- convert all ones in the string to twos, convert all zeros to ones, convert all twos to zeros, convert all Z's to X's, and leave X's intact; and
- put the modified string back into the register.

The second step converts all ones to zeros and zeros to ones. Note that the *checktf* function in this case does not differ from the earlier one, because in both cases the number and type of input parameter are the same.

CREATING THE PLI ROUTINE

Listing 5 shows the implementation of *\$invert* routine in a VCS environment. This program uses the following library functions:

- *tf_strgetp()* returns the content of the register as a string. Just like *tf_getp()*, it reads the register's con-

tent as a decimal. In **Listing 5**, *tf_strgetp(1, b)* reads the content of the first parameter in binary format. (An h or o, in place of a b, read it in hexadecimal or octal format, respectively). The routine then copies the content to a string *val_string*.

- *tf_strdelputp()* writes a value to a parameter from a string-value specification. This function takes a number of arguments, including the parameter index number (the relative position of the parameters in the system call, starting with 1 for the parameter on the far left); the size of the string whose value the parameter contains (in this case, it should be same size as the register); the encoding radix; the actual string; and two other delay-related arguments, which you ignore by passing zeros for them. Although not shown in the current example, a simpler decimal counterpart of *tf_strdelputp()* is *tf_putp()*. It is important to remember that a

LISTING 6—VERILOG PROGRAM USING \$INVERT CALL

```
module mymodule;
reg [7:0] r1;
initial begin
  r1 = 8'hf;
#100 $invert(r1); // Invert the content of r1
#100 $finish;
end
```

program can change or overwrite only the value of certain objects from a PLI routine. Any attempt to change a variable that you cannot put on the left of a procedural assignment in Verilog, such as a wire, results in an error. In general, you can modify the contents of a variable of type *tf_readwrite*. The *checktf* function *my_checktf()* checks this feature.

One additional user-defined function that **Listing 5** uses is *move()*, which has three parameters. In the third parameter, a string, the second parameter replaces every occurrence of the first parameter. In the current case, a series of calls to *move()* changes the zeros to ones and ones to zeros. However, once a program converts zeros to ones, you must distinguish the converted ones in the string

from the original ones. To make this distinction, the program first changes all initial ones to twos. A two is an invalid value for binary logic, but you can use it in an intermediate step for this example. At the end, the program converts all twos back to zeros. The program completes its task by putting the inverted value back into the register using the `tf_strdelputp()` function.

Listing 6 shows a sample Verilog program containing the call `$invert`. In a VCS environment, a file lists the functions associated with a PLI routine. Although this file can have any name, you usually call it `pli.tab`. This file is equivalent to the `veriusertfs[]` structure in Verilog-XL. The content of this file in the current example is:

```
$invert call=invert_calltf
check=my_checktf
```

By saving this program in the file `test.v`, you generate an executable binary `pli_invert` with the command:

```
$ vcs -o pli_invert -P pli.tab test.v
```

`veriusertfs`

Executing `pli_invert`, you get:

```
$invert: Modifying the content from
00001111 to 11110000 at time
100
$finish at simulation time      200
```

You now know the basic structure of a PLI routine and the mechanism of linking it to build a custom version of Verilog. You can also read, convert, and modify the values of the elementary components of a design database in Verilog through a PLI routine and read the simulation time for the routine. These tasks are the basic and often most necessary ones for a PLI routine to carry out. Considering all that PLI offers, this information is just the tip of the iceberg. You can also use a PLI to access design information other than register contents. (For more information about Verilog and C modeling, see **references 5** and **6**.) □

REFERENCES

1. Mitra, Swapnajit, *Principles of Verilog PLI*, Kluwer Academic Publisher, 1999, ISBN 0-7923-8477-6, www.wkap.nl.
2. IEEE Standard 1364-1995, IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language, IEEE Press, ISBN 1-55937-727-5, www.ieee.org.
3. *Verilog-XL Reference Manual*, Cadence Design Systems.
4. *VCS/VCSi Users Guide*, Synopsys.
5. Verilog FAQ, www.angelfire.com/in/verilogfaq.
6. Balph, Tom, and Pat O'Malley, "C modeling accelerates HDL-system growth," *EDN*, Oct 22, 1998, pg 139.

AUTHOR'S BIOGRAPHY

Swapnajit Mitra is a core-design engineer doing logic design and verification for SGI Computer Systems (Mountain View, CA). He has worked on various I/O and networking systems and has an MS degree from the Indian Institute of Technology (Kharagpur, India).