INTROSPECTION INTO SYSTEMVERILOG WITHOUT TURNING IT INSIDE OUT.

Dave Rich

Mentor Graphics. 510-354-7439 46871 Bayside Parkway Fremont, CA 94538 dave_rich@mentor.com

Abstract

Many times design engineers need to analyze their code beyond looking at simple functional behavior. How many bits of memory are in this design? How many places does the clock signal fan out to? And verification engineers need to dynamically modify their tests based on the result of the same kind of analysis. There are separate tools that provide some of this kind of analysis, and there is a programming language interface to write almost any application in C. but it would be much easier if SystemVerilog provided introspection – the ability to ask questions about your code from within your code. This paper presents a mechanism to add introspection using the existing DPI/VPI in an importable package along with an example of its use.

Introduction

In computer science, introspection in a programming language gives a program the ability to query information about itself. The key is being able to ask these question from within the program itself and act on that information. The motivation behind adding introspection to a language varies, but usually falls into one of these major categories:

- Linting: Error checking and conformity by providing information about poorly written and problematic code.
- Dynamic program modification: Making run-time decisions about a program based on characteristics of that program.
- Statistic reporting: Providing program characteristics for analysis and interfacing with other tools.

Most programming languages have very few of these capabilities natively. For example, most languages can only tell you the filename and line number of the current line of code you are executing. This comes in handy for message reporting while debugging your code. There are also various methods that give you information about the size of an array, as well as the range of indexes. These are all the capabilities SystemVerilog has built in to its language natively.

For anything beyond these simple queries, SystemVerilog provides a Verification Programming Interface providing access to a C program to query most aspects of a SystemVerilog description. But this is outside the native language and must be written in C/C++. The VPI mechanism was originally designed for tool developers integrating other tools into existing Verilog simulators. Over time, the VPI became a modeling interface integrating functionality implemented in C with functionality implemented in Verilog. However, this interface is cumbersome for people who are hardware model developers and not tool developers.

SystemVerilog [1] has a much simpler modeling interface, the Direct Programming Interface, addressing integrating C models with SystemVerilog. This allows code in one language to be called from code in another language without either being aware the code was called from another language. However, the DPI does not provide any of the design query features of the VPI.

Adding introspection to SystemVerilog natively may be a formidable task in this stage of the language's evolution. This paper provides an alternative using existing elements of SystemVerilog that achieve the same goals through a package. By combining the capabilities of the VPI with the DPI, it is possible to make design query routines appear as if they were native introspection routines to SystemVerilog. This has the benefit of eliminating the need for C programming expertise in the development of introspection-like applications.

VPI Object Model

The Verilog language was originally designed to describe hardware and provide stimulus to hardware for simulation. It deliberately left out constructs that were thought of as only useful for software development. However, users quickly found the need to integrate other applications with their simulations, such as debuggers and models written in

other languages. So the Programming Language Interface (PLI) was developed to address this need. It provided a mechanism to add functionality by allowing access to the basic signal database and structure from C. This included the ability to read and write Verilog signals, as well as annotate delays with access routines for every kind of request.

Eventually users demanded access to more elements of the description, like where declarations were made in the source and how they were being accessed within the source. It became cumbersome to extend the existing PLI access routines to handle all of the different types of information. So a whole new mechanism, the Verilog Procedural Interface (VPI)[1] was introduced. The VPI is organized similarly to the way one might create an object oriented system, with virtual methods that return different types of data depending on an opaque handle to a referenced object.
An example of this object model is illustrated with a diagram showing the relationship between a module and the signals (variables or nets) contained within that module.

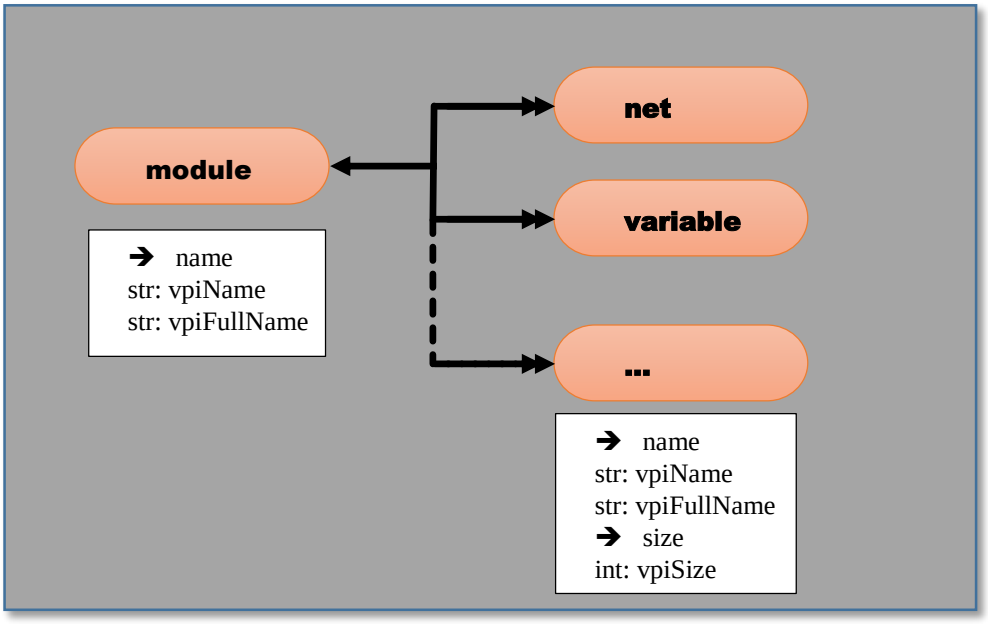

Figure 1 VPI Object Model Diagram

There is a one to many relationship between a module and the many variables it may contain, but there is only one-to-one relationship between a variable and the single module that contains it. The VPI provides a small set of routines for traversing these relationships.

[1] This was later renamed *Verification Procedural Interface* after the introduction of SystemVerilog.

```
// one-to-one
vpiHandle mod_h; // hold a handle to a module instance
vpiHandle net_h; // holds a handle to a net instance
mod_h = vpi_handle(vpiModule, net_h); // given a net, which module instance
does it belong to
// one-to-many
vpiHandle itr;
itr = vpi_iterate(vpiNet, mod_h);
while (net_h = vpi_scan(itr) ) // given a module instance, print all the nets
vpi_printf("\t%s %d\n", vpi_get_str(vpiFullName, net_h), vpi_get(vpiSize, net_h );
```

Figure 2 VPI Application

Note that all of the different objects use a handle with the same type, vpiHandle. You can think of vpiHandle as a base class type with different properties and methods available depending on the type of object it references. You can get the vpiSize property of a net, but not a module.

The most common VPI routines used to traverse and interact with a design can be expressed using the following table.

vpiHandle vpi_handle_by_name(char* **name**, vpiHandle **scope**);	Return a handle by a string name
vpiHandle vpi_handle(int **type**, vpiHandle **ref**)	Return a handle of **type** associated with object **ref**
vpiHandle vpi_iterate(int **type**, vpiHandle **ref**)	Return a itorator of **type** associated with object **ref**
vpiHandle vpi_scan(vpiHandle **itr**)	Return a handle to the next object based on the **type** of **itr**.
int vpi_get(int type, vpiHandle **ref**)	Return an integer property of **type** associated with object **ref**
char* vpi_get_str(int **type**, vpiHandle **ref**)	Return a string property of **type** associated with object **ref**
vpiHandle vpi_get_value(vpiHandle **obj**, p_vpi_value value_p)	Retrieve the value of the referenced object **obj** into value_p
vpiHandle vpi_put_value(vpiHandle obj, p_vpi_value value_p)	Assign the value of the referenced object obj using value_p

Figure 3 Common VPI Routines

Although it seems that C++ might have been a better choice as a programing interface language, the major benefit of defining the VPI as a C library is that C is almost the universal choice for language binding. That means you can write in your favorite language and use these VPI routines as if they were part of your programming language choice.

The VPI provides a powerful interfaces to implement many kinds of applications. However there are two spaces it does not handle very well. When trying to integrate models through the VPI, passing data between the language boundaries becomes complex and can suffer a large performance penalty. Much of the performance penalty is consumed by manipulating the internal simulation data structures into a form required by the public VPI, and then pack into the internal data structures. Most models just want to input a set of data; do some computations on that data, and then output the data. There is no need for design access once inside the C model.

Below is an example of a routine that computes the sum of members of a struct. The VPI requires many access routines just to get at the values of two struct members. This example assumes the input to the function is correct and does not use any of the additional routines that normally would be used for compile time error checking.

```
typedef struct { int A,B; } AB_s;
AB_s ab = '{1,2};
int result;
initial result=$sum(ab) // should return 3
```

Figure 4 VPI Call from SystemVerilog

```
int calltfSum(char* user_data) {
   vpiHandle systf_h, itr_h, arg_h,
   s_vpi_value argVal;
   int A, B;
  systf_h = vpi_handle(vpiSysTfCall, NULL);   /* call instance */
  itr_h = vpi_iterate(vpi_Argument, systf_h); /* iterate over arguments */
  arg_h = vpi_scan(itr_h);         /* get handle to 1st argument */
  vpi_free_object(itr_h);          /* housekeeping */
  itr_h = vpi_iterate(vpiMember,arg_h); /* iterate over struct members */
  arg_h = vpi_scan(Itr_h);        /* handle to A member */
  argVal.format = vpiIntVal;
  vpi_get_value(arg_h, &argVal); /* get A value in int format */
  A = argVal.value.integer;
  arg_h = vpi_scan(itr_h);        /* handle to B member */
  vpi_get_value(arg_h, &argVal); /* get B value in int format */
  B = argVal.value.integer;
  argVal.value.integer = A + B;
  vpi_put_value(systf_h, &argVal, NULL, vpiNoDelay); /* function return value */
  vpi_free_object(itr_h);         /* housekeeping */
```

Figure 5 C code for VPI application

DPI

The Direct Programming interface (DPI) was created to serve the modeling space by passing arguments across the language boundary just like any other language binding. There is no need to read or write arguments through access routines. This is aided by the fact that SystemVerilog has a more comprehensive type system that is compatible with C types. [2] Here is the same VPI sum example written using the DPI:

```
typedef struct { int A,B; } AB_s;

AB_s ab = '{1,2};
int result;

import "DPI-C" context
initial result=sum(ab) // should return 3
```

Figure 6 SystemVerilog calling DPI application

```
typedef struct { int A,B; } AB_s;

int sum( const AB_s* arg) {
  return arg->A + arg->B;
}
```

Figure 7 C code for DPI Application

SystemVerilog can import and call this sum() routine and the C code executes without any knowledge of how SystemVerilog data structures are organized.

The VPI as an DPI application

As stated before, the DPI provides all of its access through the arguments of the calling routine. But once across the language boundary into C, one can use the VPI routines. It would be ideal to import the VPI routines directly, but many of those routines have argument types that are not compatible with SystemVerilog, such as all pointer types. So wrappers are needed to handle the type conversions as necessary.

The vpiHandle type is used as an opaque pointer type and readily maps to the DPI's chandle type. vpi_handle could be imported directly. Any other types, like s_vpi_value, that are pointers referencing memory on the C side need to get converted into a SystemVerilog compatible type. So vpi_get_value() must be transformed into a function that returns an integer directly.

```
int32_t VPI_get_int_value(vpiHandle handle) {
  s_vpi_value value_p;
  if (handle) {
    value_p.format = vpiIntVal;
    vpi_get_value(handle,&value_p);
    return value_p.value.integer;
  }
  else
    vpi_printf("VPI_get_value: error null handle\n");
}
```

There would need to be eight different wrappers of vpi_get_value to handle the eight different value formats that it could return. Those eight different formats cover retrieving the values for variables and nets of every SystemVerilog type. Remember that values of aggregates such as arrays and structs are retrieved by iterating over members as shown in Figure 5 C code for VPI application. Once all the VPI DPI-C routines are defined inside a package along with their associated object type constants (i.e. `vpiModule`), that package can be used to write almost any VPI application in SystemVerilog. The same VPI C application can be written in SystemVerilog and would look almost the same.

```
module top;
  typedef struct { int A,B; } AB_s;
  AB_s ab = '{1,2};
   vpiHandle itr_h, arg_h,
   int A, B, result;
  initial begin
    arg_h = vpi_handle_by_name("top.a");   /* get handle to ab variable */
    vpi_free_object(itr_h);          /* housekeeping */
    itr_h = vpi_iterate(vpiMember,arg_h); /* iterate over struct members */
    arg_h = vpi_scan(Itr_h);        /* handle to A member */
    A = vpi_get_int_value(arg_h);  /* get A value in int format */
    arg_h = vpi_scan(itr_h);        /* handle to B member */
    B = vpi_get_int_value(arg_h);  /* get B value in int format */
    B = argVal.value.integer;
    result = A + B;
    vpi_free_object(itr_h);         /* housekeeping */
  end
endmodule
```

Figure 8 VPI application using DPI

Notice that the code above passes the variable 'ab' as a full pathname string. One of the shortcomings of the DPI is that because arguments are effectively always passed by value, there is no way to get a handle to the expression pass to it, so you have to look up the argument by a full path name. There are a number of possible workaround to this problem that need further exploration. A VPI system function could be created that returns a handle to the object passed to it (i.e. `my_handle = $get_hande(my_object);`). The DPI provides a `svGetNameFromScope()` routine that returns a string with the full instance path name from the context of where the DPI routine was called from. This call could be added to a vpi_get_handle_by_name() wrapper that prefixes a path name to the string argument. You can always get a handle to the top of module hierarchy by using `vpi_handle(vpiModule,NULL);`.

Some objects you will not be able to pass as an argument to a VPI routine, (i.e. package or typedef name), so they will have to be looked up by a string pathname, or scanned by name. But once you get a handle to an object of interest, the full power of the VPI is available to you from within your SystemVerilog code.

Summary

By creating a SystemVerilog package of VPI routines, verification engineers now have a quick way of accessing the VPI within the SystemVerilog language. This can aid in rapid prototyping of applications where extensive C knowledge is not available, or familiarity with the integration of C code with the simulator is a barrier. The package shown here in the Appendix has been used in a number of production environments to help print out reports about module parameterizations and network statistics. It has not been used to interact with the event scheduler.The package uses the DPI and VPI and defined in the SystemVerilog standard. However the user should be aware that there may be differences in VPI implementation across tool vendors that need to be verified beforehand.

Bibliography

[1] IEEE, IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language, New York: IEEE 1800-2012, 2012.

[2] R. Edelman, "Using SystemVerilog Now with DPI," in *Design and Verification Conference*, San Jose, 2005.

Complete Example

```
/* $Id: top.sv,v 1.7 2014/08/01 20:39:37 drich Exp $ */
   /* dave_rich@mentor.com  */

   module automatic top;
   import VPI_pkg::*;

   string pathname = "top.";
   integer A=2;

   vpiHandle my_handle;

   initial begin
        my_handle = vpi_handle_by_name({pathname,"A"},null);

        $display("top.A is %d",vpi_get_value(my_handle));

        vpi_put_value(my_handle, 3);

        $display("top.A is %d",vpi_get_value(my_handle));
        end

      DUT DUT();

      vpiHandle listOfNetworks[string][$];

      initial begin
         doAllModules(null);

         foreach(listOfNetworks[simNet]) begin
         vpiHandle networks[$];
         networks  = listOfNetworks[simNet];
         $display("Network %s",simNet);
         foreach(networks[nHandle]) begin
            $display(vpi_get_str(vpiFullName,networks[nHandle]));
         end
         end
      end

      function void doAllModules(vpiHandle scope);
         vpiHandle moduleIterator,moduleHandle;

         moduleIterator        = vpi_iterate(vpiModule,scope);

         while((moduleHandle = vpi_scan(moduleIterator)) != null)
       begin
          doAllModules(moduleHandle);
          print_timescale(moduleHandle);
          print_parameters(moduleHandle);
          collect_simulated_nets(moduleHandle);
       end
      endfunction : doAllModules
```

```
   function void print_timescale(vpiHandle moduleHandle);
      int TimeUnit  = vpi_get(vpiTimeUnit, moduleHandle);
      int TimePrec  = vpi_get(vpiTimePrecision, moduleHandle);
      $display("Found Module %s Timescale:  %0d/%0d",
            vpi_get_str(vpiFullName, moduleHandle),
            TimeUnit, TimePrec
            );
   endfunction : print_timescale

   function void print_parameters(vpiHandle moduleHandle);
      vpiHandle paramIterator,paramHandle;
      paramIterator  = vpi_iterate(vpiParameter,moduleHandle);
      while((paramHandle = vpi_scan(paramIterator)) != null)
     $display("parameter %s=%d",
              vpi_get_str(vpiFullName,paramHandle),
              vpi_get_value(paramHandle)
              );
   endfunction : print_parameters

   function void collect_simulated_nets(vpiHandle moduleHandle);
      vpiHandle netIterator,netHandle;
      netIterator  = vpi_iterate(vpiNet,moduleHandle);
      while((netHandle = vpi_scan(netIterator)) != null)
     listOfNetworks[
                   vpi_get_str(vpiFullName,vpi_handle(vpiSimNet,netHandle))
                   ].push_back(netHandle);

   endfunction : collect_simulated_nets

endmodule

module DUT;
   timeunit 1ms;
   timeprecision 10ns;

   wire w1,w2,w3;

   sub #(1) u1(w1,w2);
   sub #(2) u2(w1,w2);
   dub u3(w1);
endmodule : DUT

`timescale 10ns/1ps

module dub(inout wire p3);
   wire w6;
   sub #(3) u5(w6,p3);
endmodule

`timescale 1ns/10ps
module sub(inout wire p1,p2);
   parameter P      = 0;
   wire p3;
endmodule
```

Appendix

SystemVerilog DPI package

```
//------------------------------------------------------------------------
//   Copyright 2005-2007 Mentor Graphics Corporation
//
//   Licensed under the Apache License, Version 2.0 (the
//   "License"); you may not use this file except in
//   compliance with the License.  You may obtain a copy of
//   the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in
//   writing, software distributed under the License is
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
//   CONDITIONS OF ANY KIND, either express or implied.  See
//   the License for the specific language governing
//   permissions and limitations under the License.
//------------------------------------------------------------------------
/* $Id: dpi_vpi.sv,v 1.6 2013/03/22 23:39:42 drich Exp $ */
/* dave_rich@mentor.com  */

package VPI_pkg;
   typedef chandle vpiHandle;
   // context is needed for calling VPI from DPI imported routines

   import "DPI-C" context
     VPI_handle_by_name = function vpiHandle vpi_handle_by_name(string name, vpiHandle scope);
   import "DPI-C" context
     VPI_iterate           = function vpiHandle vpi_iterate (int _type, vpiHandle _ref);
   import "DPI-C" context
     VPI_scan              = function vpiHandle vpi_scan (vpiHandle itr);
   import "DPI-C" context
     VPI_get_str           = function void _vpi_get_str (int prop, vpiHandle obj, output string
name);
      function automatic string vpi_get_str(int prop, vpiHandle obj);
      string s;
      _vpi_get_str(prop,obj,s);
      return s;
      endfunction

   import "DPI-C" context
     VPI_handle            = function vpiHandle vpi_handle(int prop, vpiHandle obj);
   import "DPI-C" context
     VPI_get               = function int vpi_get(int prop, vpiHandle handle);
   import "DPI-C" context
     VPI_get_int_value            = function int vpi_get_int_value(vpiHandle handle);
   import "DPI-C" context
     VPI_put_int_value            =  function  void  vpi_put_int_value(vpiHandle  handle,  int
value);
```

```
/******************************** OBJECT TYPES ********************************/

    parameter vpiAlways           = 1;   /* always construct */
    parameter vpiAssignStmt       = 2;   /* quasi-continuous assignment */
    parameter vpiAssignment       = 3;   /* procedural assignment */
    parameter vpiBegin            = 4;   /* block statement */
    parameter vpiCase             = 5;   /* case statement */
    parameter vpiCaseItem         = 6;   /* case statement item */
    parameter vpiConstant         = 7;   /* numerical constant or string literal */
    parameter vpiContAssign       = 8;   /* continuous assignment */
    parameter vpiDeassign         = 9;   /* deassignment statement */
    parameter vpiDefParam         = 10;   /* defparam */

[ The complete set of constants appears in Annex K and M od the IEEE 1800-2012 LRM]
endpackage : VPI_pkg
```

VPI C wrapper

```
//------------------------------------------------------------------------
////   Copyright 2005-2012 Mentor Graphics Corporation
////
////   Licensed under the Apache License, Version 2.0 (the
////   "License"); you may not use this file except in
////   compliance with the License.  You may obtain a copy of
////   the License at
////
////       http://www.apache.org/licenses/LICENSE-2.0
////
////   Unless required by applicable law or agreed to in
////   writing, software distributed under the License is
////   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
////   CONDITIONS OF ANY KIND, either express or implied.  See
////   the License for the specific language governing
////   permissions and limitations under the License.
////------------------------------------------------------------------------
/* $Id: dpi_vpi.c,v 1.5 2013/03/22 22:46:21 drich Exp $ */
/* dave_rich@mentor.com  */

#include "vpi_user.h"
#include "dpi_vpi.h"

vpiHandle VPI_handle_by_name(const char* name, vpiHandle scope) {
  vpiHandle handle;
  if(handle = vpi_handle_by_name((PLI_BYTE8*)name,scope))
    return handle;
  else {
      vpi_printf("VPI_handle_by_name: Can't find name %s\n", name);
      return 0;
  }
}
int32_t VPI_get_value(vpiHandle handle) {
  s_vpi_value value_p;
  if (handle) {
```

```
    value_p.format = vpiIntVal;
    vpi_get_value(handle,&value_p);
    return value_p.value.integer;
  }
  else
    vpi_printf("VPI_get_value: error null handle\n");
}
int32_t VPI_get(int32_t prop, vpiHandle obj) {
  if (obj) {
    return vpi_get(prop, obj);
  }
  else
    vpi_printf("VPI_get: error null handle\n");
}
vpiHandle VPI_iterate(int32_t type, vpiHandle ref) {
  return vpi_iterate(type,ref);
}
vpiHandle VPI_scan(vpiHandle itr){
  return vpi_scan(itr);
}
vpiHandle VPI_handle(int32_t type, vpiHandle ref) {
  return vpi_handle(type,ref);
}
void VPI_get_str(int32_t prop, vpiHandle obj, const char** name) {
  if (obj) {
    name[0] = vpi_get_str(prop, obj);
    return;
  }
  else
    vpi_printf("VPI_get_str: error null handle\n");
}
void VPI_put_value(vpiHandle handle, int value) {
  s_vpi_value value_p;
  if (handle) {
    value_p.format = vpiIntVal;
    value_p.value.integer = value;
    vpi_put_value(handle,&value_p,0,vpiNoDelay);
  }
  else
    vpi_printf("VPI_put_value: error null handle\n");
}
```