The Verilog Procedural Interface for the Verilog Hardware Description Language

Charles Dawson, Sathyam K. Pattanam, David Roberts

Cadence Design Systems Inc.

Abstract

The Verilog Procedural Interface is a new C-programming interface for the Verilog Hardware Description Language. Different Verilog HDL based tools such as simulators, synthesizers, timing analyzers, and parsers could support this interface for applications which extend the tool's functionality. VPI is part of the IEEE 1364 Programming Language Interface standard.

VPI is considered to be the third generation procedural interface to Verilog HDL. The first two generations evolved in conjunction with Verilog-XL and the Verilog HDL. This process resulted in interfaces which lacked consistency and functionality for applications. VPI provides a consistent object-oriented access to the complete Verilog HDL language as described in the IEEE 1364 Language Reference Manual. VPI also provides a well defined interface for supporting Verilog-HDL based simulation. It is believed that this interface can be easily extended to meet future needs.

A major portion of the VPI functionality is available in the Verilog-XL 2.2 simulator released in 9502. The complete VPI functionality will be available in the Verilog-XL 2.3 simulator to be released in 9504.

This paper briefly discusses the evolution of the Verilog HDL programming language interfaces, features of the VPI interface, and a set of possible powerful applications.

1 Introduction

It wasn't long after Verilog-XL was first introduced, that it became apparent that an Application Procedural Interface (API) for it would be extremely useful. This API has been known as the Programming Language Interface (PLI). It has been used so much that it has become a defacto standard, along with the Verilog language itself. Verilog HDL based tools such as logic simulators, logic synthesizers, timing analyzers, ...etc. provide the PLI as a mechanism for applications to access the description and to control the behavior of the tool. Possible applications are delay-calculators, delay-annotators, test-vector injectors, debugging environments, translators, and interfaces to software and hardware models.

The PLI has evolved significantly over the years. The first generation of the PLI was the "task-function routines" also called as the "TF interface". These routines are primarily used for operations involving arguments of user-defined system tasks and functions in the Verilog description. User-defined system tasks and functions are constructs in the Verilog-HDL that are prefixed with the '$' (dollar) sign. For each user-defined system task and function the application could register three callback functions:

checktf - called during compilation
calltf - called during simulation when the system task or function is actually executed
misctf - called for miscellaneous reasons such as when an argument changes value

The application's calltf function can also return a value for a user-defined function. Another capability this interface has is the ability to write values to nets and to be able to propagate these values to rest of the description. The main drawbacks of this interface was that it only provided very limited functionality and did not provide any access to the Verilog description.

The second generation interface of the PLI was the "access routines" also called as the "ACC interface". The major improvement which the ACC interface provided was access to the design-hierarchy, connectivity, and structural information. It provides functionality to set and propagate values of regs, and to modify delay values. It also has the ability to register

1085-9403/96 $5.00 © 1996 IEEE

callbacks for value changes of nets, regs, and variables. This callback mechanism is known as the Value Change Link, or VCL. The ACC is similar to the TF inteface in that it suffers from inconsistent and incomplete access to the Verilog description.

The new interface called the "Verilog Procedural Interface" (VPI) is considered the third generation procedural interface. This interface provides a consistent data-model of the complete Verilog HDL and an object-oriented access to the data-model. This interface has been designed to be independent of the TF and ACC interfaces. Therefore, it has all of the functionality provided by the other interfaces.

All the three generations of the Verilog programming interfaces are part of the Verilog IEEE 1364 standard. Cadence has played a significant role in defining these interfaces, and will deliver the first simulator in the industry to have this new interface along with the previous interfaces. Verilog-XL version 2.3 to be released in the last quarter of 1995 will have the new VPI interface.

2. The VPI Interface

The VPI interface was designed to provide a uniform access method to the data within a Verilog design. This methodology has been separated from the data itself which simplifies extensibility. VPI provides access in terms of handles to objects which are recognizable to the Verilog designer. Access to simulation specific information has been designed systematically to provide completeness and accuracy. A new callback mechanism is included which provides more functionality than VCL. Finally, the important functionality of the older PLI utility routines has been provided by utility routines within the VPI.

2.1 Access methodology

All data within a Verilog design is related in some way. The VPI access methodology has been defined in terms of these relationships. As far as a Verilog design is concerned, there are only four types of relationships:

not-related
one-to-one
one-to-many
many-to-one

Conceivably, there could be a fifth relationship, many-to-many, however a need for this not been found. In other words, any given object within a Verilog design is related to another type of object(s) in one of these four ways. As an example, any given primitive instance only lives within one module. The relationship between the primitive instance and it's containing module instance is referred to as one-to-one. In diagram 1, this relationship is illustrated by the single arrow from primitive to module.

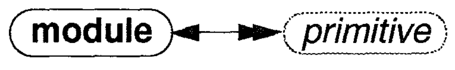

Diagram 1

The routine which is used to make this kind of traversal is *vpi_handle()*.

Any module instance could have many (zero or more) primitive instances. Therefore, the relationship between the module instance and the primitive instances contained within it is referred to as one-to-many. In diagram 1, this relationship is illustrated by the double arrow from module to primitive. The routines used to make this kind of traversal are *vpi_iterate()* and *vpi_scan()*. *vpi_iterate()* returns an object known as an iterator. An iterator holds the place in the list of objects. *vpi_scan()* moves the place holder to the next object in the list.

These relationships define the basic methodology for traversing the data structures which represent the Verilog design.

Associated with any given object within the design is a set of properties. These properties are almost always expressed as either an integer or a string. Since most properties have these two types, two routines which will return a property value have been created (conceivably, there could be routines for other basic C variable types, such as floats). More complicated properties (such as values and delays) are handled individually. Diagram 2 illustrates one of each of these properties.

To access a property which is of type str, use the routine *vpi_get_str()*. To access a property of type int, use the routine *vpi_get()*. More complex properties have the

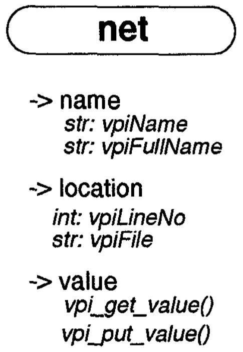

Diagram 2

routine for accessing them directly in the diagram describing the object. For example, to get the value of a net, use *vpi_get_value()*.

2.2 Classes and Methods

Sometimes it is convenient to group a set of objects together. A group of objects are referred to as a class. Diagram 3 illustrates a class definition.

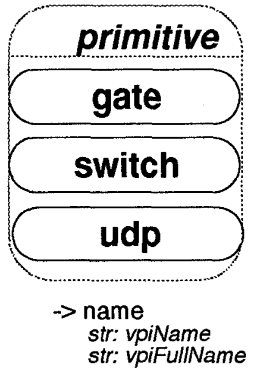

Diagram 3

The properties vpiName and vpiFullName apply to all of the objects within the class primitive. It is possible to access a class from some given object using a method. Methods are illustrated by tags such as vpiRhs and vpiLhs in diagram 4.

Methods and classes never correspond to actual object

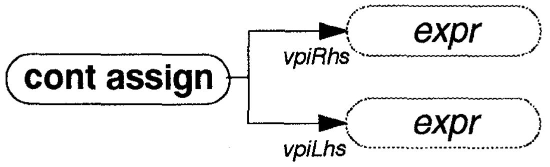

Diagram 4

types, since they represent groups of objects. In other words, the type of a gate object will be vpiGate, not the class vpiPrimitive. Another example would be using the method vpiRhs. Using vpiRhs from a cont assign will not return an object whose type is vpiRhs. The object will have a type of one of the objects within the class vpiExpr.

2.3 Data Availability

The data which comprises a Verilog design and simulation can be broken into three categories. The first category of data is information which is contained within any single module description. The second category of data is connectivity and hierarchy information created during linking (also known as elaboration). The last category of data would be simulation specific information created as the simulation progresses. The first two categories contain static information. The VPI as defined in IEEE 1364 allows access to all of the information within these two categories. In other words, it is assumed that an implementation of VPI will not attempt to allow applications to access data until after elaboration. However, it is possible to define a subset of the access provided by the VPI to be available after compilation, but prior to elaboration.

2.4 Static data Model

Most of the information to which VPI provides access is static in nature. It originates in the HDL itself, and remains available throughout the simulation. This static data model is encapsulated in the data model diagrams described in sections 2.1 and 2.2. Often, this information is not needed during the actual simulation of the design. Most VPI applications will need at least some of this information during their processing. This can create a "conflict of interest" between the simulator which is trying to simulate the design as efficiently as possible, and the VPI application. Sometimes the VPI implementation will need to create objects which the simulator does not keep. A routine, *vpi_free_object()* has been created which allows the VPI application to inform the interface that the object is no longer needed.

2.5 Simulation Specific Data

Access is provided through the VPI to simulation specific information once simulation has begun. Simulation specific information includes:

the values of non-constant objects such as nets or regs
delay values, which can be modified by VPI applications such as an SDF annotator
the current simulation time
access to time slices in the time queue

This information is also known as dynamic information, since it will change as the simulation progresses.

2.6 Callbacks

Process control is handled through a mechanism known as callbacks. A VPI application tells the simulator to call a specified routine when some kind of event occurs. These events are separated into two categories:

Callbacks for event related to a specific HDL object
Callbacks for simulator events

The first category includes events such as when a net changes value or before a behavioral statement is executed. The second category include events such as the end of compilation, or before a save occurs.

VPI also defines a mechanism which allows the VPI application to register new user defined system tasks and functions with the simulator. Associated with the system tasks are a set of routines which will be called when events related to the task or function occur (i.e. when the task or function is compiled, a routine can be called to allow the VPI application to check that the task or function is being used properly). These routines are similar to the checktf, calltf, and misctf routines in the TF interface. A new system task or function can be register through the VPI at any time, which eliminates a restriction of the previous interfaces.

Callbacks can also be removed. Routines which create callbacks return a handle to an object whose type is vpiCallback. This handle can be passed to another routine which will remove the callback.

2.7 Utility routines

VPI provides a set of utility routines which complement the rest of the interface.

Utility routines provide the ability to write to files. There is a routine which allows an application to write to multiple files simultaneously. This mechanism is known as Multi-Channel Descriptors, or MCDs. Descriptors are reserved for the stderr, stdout, and the simulator log file. There is a mechanism available to open and close files as well. Another routine will provide the name of a file given a channel descriptor.

VPI has a well defined error mechanism. This error mechanism is passive. In other words, the VPI application can choose to ignore errors. There is a callback reason provided to allow a specified routine to be called whenever an error occurs. Possible errors that can be handled through this mechanism include errors caused by the VPI application itself, and errors which are simulation related, such as an error during compilation.

3 Applications for the VPI

Any application which used the ACC and TF interfaces can be written using the VPI interface. One of the major benefits of using VPI is that it was designed to provide access to the entire Verilog HDL. The ACC interface was defined to handle the structural aspects of a design, but largely ignored behavioral HDL objects. Some examples of applications that can only be accurately written with VPI are:

Verilog HDL Decompiler/Translator
Design Rule Checker
Driver Resolution

3.1 Verilog HDL Decompiler/Translator

An application can be written which will traverse the entire design and decompile it in the exact syntactic format of the original text. A very similar application would be a translator (i.e. Verilog to VHDL). One weakness of the VPI is that it does not handle the definition and use of compiler macros. This weakness can affect a decompiler/translator.

With the ability to access all parts of the language, applications can now deal with issues in a more natural way. If an ACC application has to deal with an expression, it would have to get the expression as a string, then parse the string to perform the required operations. A VPI application can get a handle to each leaf element in the expression, eliminating the need for an expression parser in the application.

3.2 Design Rule Checker

A design rule checker can be written to verify both structural and behavioral code. Unlike design rule checkers written as ACC applications, a VPI design rule checker can now enforce design restrictions. There are HDL constructs that the ACC interface cannot detect, causing an ACC design rule checker to be unable to accurately control a design style.

Some design methodologies could actually remove restrictions, which were required because of deficiencies in the ACC and TF interfaces.

Since VPI provides the ability to setup callbacks without having a user defined system task or function within the HDL, VPI applications can now be built into the simulator and executed automatically. A design rule checker can be installed which would not require the designer to call a system task or function in order to invoke it. This application can persist throughout the course of the simulation, without the designer aware of it (providing he does not violate any design rules).

3.3 Driver Resolution

Using VPI, an application can now find all of the contributors to a net's value, not just the structural drivers. The application is able to find continuous assignments, primitives, forces, behavioral ports, ...etc. The VPI application can get the values and strength of all these drivers. Except for finding future scheduled values on a driver, an application can be developed that produces the same output as the system task $dumpvars.

An application can accurately detect two or more contributors which are driving the same value on a net at the same time. While this is allowed by the Verilog HDL, this tends to be considered by designers to be a run time error.

Another by-product of being able to find all the drivers of a net is that delay calculators can now be enhanced to deal with behavioral models. Since the ACC interface cannot handle behavioral code, a design can attach a behavioral model to a net and the delay calculator will never know it is there.

4. Weaknesses of the VPI

There are several weaknesses with the present VPI definition. Fortunately, because of the care which was taken in it's design, the VPI can easily be extended to rectify these deficiencies. The weaknesses can be broken into six categories:

Text biased applications
No concept of an object's reference
Cannot create objects
Data model restricted to language concepts
Simulator control
Memory consumption

4.1 Text Biased Applications

The VPI interface was designed to provide access to a compiled design. Because of this, a text biased application cannot find and work with many of the different compiler directives, including:

```
`define macro `macro
`if `else `endif
```

Properties needed for textual manipulation are also missing. An example would be the column location of an object within a file. Textual biased applications such as GUIs and debuggers will need further extensions to the VPI.

4.2 No Concept of Object References

Any object which can be termed "declared" in VPI has a name. For example, any net has a name. Declared objects can be referred to in many places. The VPI has no concept of a reference to an object. Often, when a VPI application is traversing a design and it encounters an object reference the application can lose it's context.

Access to expressions is a typical scenario for this type of problem. As an example, given the following expression:

```
a + b
```

Where a and b are declared as:

```
wire a, b;
```

A handle to this expression would have a type of vpiOperation. You can get the line number of this expression because it is complex. You can also get the operands of the plus operator. The operands for the expression are references to nets. When a VPI application traverses to the operands, it will get objects of type vpiNet. An attempt to get the line number for these objects will return the line number for their declaration, not the expression.

Further, once an application goes to the operands, there is no way to traverse back up the expression tree to the operator.

4.3 Data Model Restricted to Language Concepts

The VPI was designed using terms which are familiar to Verilog designers. Unfortunately, some situations call for new types of objects which don't necessarily fit this criteria. As an example, take the following continuous assign:

```
assign {a,b} = foo[1:0];
```

If an application iterates on drivers for the scalar signal "a" it will get the handle to the continuous assignment. The expression on the left hand side can be obtained and then scanned to find where the object of interest exists in the expression. Determining which bit of the continuous assignment is driving the signal could be simplified with the creation of the concept of a continuous assignment terminal and continuous assignment terminal bits.

4.4 Cannot Create Objects

Another powerful concept which could be added to the VPI interface would be the ability to create objects from within the VPI application. At present VPI cannot create objects, they must already exist within the HDL.

Many applications would like the ability to create a driver on a net. VPI is able to place values directly into a net, but this is of little use on a net with any drivers. Placing a value on a net will override the values of the drivers, until one of them changes value again causing the net to be re-evaluated. Using VPI to set the value of a net is known a "soft forcing" the net. If an application could create a persistent driver of a net, in which it has control over both the value and drive strength, it could influence the net in a more natural way. This driver would be able to influence the net just as any other driver would.

4.5 Simulator Control

Access to individual events was determined to be too implementation specific to add to the IEEE 1364 specification. The problem is that there is no consensus on the definition an event. Certainly, an event in a cycle simulator may not be defined the same as an event in an event driven simulator. Events mean different things in a compiled simulator versus a interpreted simulator, ...etc.

Given the variety of Verilog simulators today, we could not provide a general solution that would allow VPI applications to access events, even though this could be a very useful feature.

Other simulation control functions are missing, including:

The equivalent to the system tasks $stop and $finish

A graceful way to get the simulator to exit and control the exit status

A signal handler which could call a VPI application when it receives an interrupt.

4.6 Memory consumption

At present once a handle is given to an application it must remain valid until the end of the simulation or until the application calls *vpi_free_object()*. Only when an application calls *vpi_free_object()* does the simulator know that the application is not retaining the handle for some reason. There are situations when an application will access an object that will force the VPI interface to allocate memory for the handle in order to return it to the application. Currently there is no way to notify the VPI application that this is occurring. Therefore, in order to prevent memory leaks, the application will need to call *vpi_free_object()* for every handle it encounters.

Since a typical application will not retain copies of most of the handles it encounters, this mechanism is too awkward. A mechanism which would better fit the most common use model would be to have the application inform VPI that it is retaining the handle. This would allow the VPI to free allocated storage in a more timely manner without overburdening the application.

5. Extensions to the VPI and the Future

Clearly there is a lot of work left before there will be an API for the Verilog language which completely satisfies the needs of most applications. It was obvious from the prior attempts to create an API for Verilog that any attempt at a new API should be general and easily extensible. In this regard, the VPI has been very successful. Cadence is in the process of extending the VPI to meet the needs of many of VPI's customers.

5.1 Text Biased Applications

We have already begun extensions to the VPI to handle text biased applications. Specifically, the new UI for the INCA technology will be using VPI heavily. These extensions include the ability to get handles to every textual reference of a defined object (i.e. a net) in a module, and to get the column properties needed for selection of an object in a file.

Eventually Cadence hopes to propose the extensions to the IEEE for inclusion in the next version of the IEEE 1364 document.

5.2 VPI and VHDL

There currently is no industry standard API for the VHDL language. Each simulator has it's own API. This causes a great deal of problems for customers who are trying to write applications for more than one simulator.

Cadence is currently applying the knowledge gained through the development of VPI on a new API for VHDL. Essentially, this new API, known as VHPI, will use the same or very similar access mechanism, but it will have it's own data model. VHPI may also use the utility routines and callback mechanism which are part of the VPI. Other parts of the VPI, such as access to values and delays, may be too Verilog specific for VHPI. However, expect VHPI to provide similar functionality in a similar way.

Eventually, Cadence hopes to propose the VHPI to the IEEE for inclusion in the VHDL standard.