Exploiting VHDL and Verilog Interoperability - A Designer's Perspective

Ajoy Aswadhati, Cirrus Logic Inc.
3100, W. Warren Ave., Fremont, CA-94538. Email: ajoy@corp.cirrus.com

Naveena Nanuswamy, Cadence Design Systems, Inc.
270 Billerica Road, Chelmsford, MA 01824. Email: naveena@cadence.com

Abstract

As the majority of VLSI designs migrate to the submicron and deep submicron processes, design complexities continue to increase at a rapid pace. There is no corresponding increase in the design cycle, and the designers are challenged to increase their productivity. HDL (Hardware Description Language) based top-down design methodology can be used effectively to help cut the design cycle. The choice of the HDL, VHDL or Verilog, has a different impact on the design methodology. While they have overlapping scopes, they do have complementing strengths. Today's designers can leverage these complementing characteristics and successfully design next generation products. This paper will enumerate our experiences in the application of both VHDL and Verilog for a successful design methodology.

1. Introduction

Top-down design relies on validating the architecture and the algorithm choices, and leverages the logic synthesis tools for implementation. VHDL, with its ability to describe designs at any level of abstraction is tailor made for architectural and algorithmic analysis. But, the lack of certified technology libraries causes a discontinuity in the design flow. Verilog has been adopted widely by the semiconductor vendors, and meets the timing accuracy requirements for deep sub micron technologies. The ideal solution is to have the expressive power of VHDL, and the implementation accuracy of Verilog libraries.

It is more common than not for the next generation designs to reuse netlists, and pre-designed blocks. These designs could be in-house or from a silicon design partner. Often, these designs are only available in Verilog. In such cases, system-level validation requires VHDL/Verilog interoperability. It is imperative for the designers to be able to leverage the strengths of both VHDL and Verilog, and not to be penalized due to the language choice. EDA vendors have produced technologies which provide VHDL/Verilog interoperability based on different co-simulation approaches. It is essential for the designers to understand the characteristics of these approaches, and make the necessary changes in their design methodology. The co-simulation approaches vary with regards to the performance overhead (for intra-engine traffic), type mapping between Verilog and VHDL, and the user interface. These characteristics can positively or negatively impact the designer's productivity based on the application.

This paper will outline a design methodology that is oriented around the usage of a VHDL/Verilog interoperable environment. We share our experience with this environment and the optimum design flow to get the most out of the two languages. This paper is not meant to be a feature comparison of the languages, but rather a "lessons learned" approach. The paper will also comment on the current status of the languages and the other standards in progress (like VITAL).

2. Why VHDL/Verilog interoperability?

This paper assumes that the audience is quite proficient in VHDL and the inherent advantages it has for describing complex systems. VHDL provides a built in scalability of the language through abstract data types, composite data types, packages, operator and subprogram overloading. These concepts are not available in Verilog. We believe that VHDL is well poised for describing and designing complex systems.

However using VHDL for complex IC (Integrated Circuit) design poses some difficulties. This seeming difficulty in describing IC's using VHDL could suggest a VHDL/Verilog interoperable environment. VHDL could be used as a first step in describing the IC to be designed at a very high level. Using the notion of multiple architectures for each entity, one could evaluate the trade off in choosing different architectures. Once a particular

architecture is determined to be optimal (say with respect to cost and performance), the IC could be implemented in Verilog which is very tuned for IC design. The original high level description of the IC in VHDL could be used for verifying the correctness of the equivalent Verilog implementation which could be just another architecture of the same entity. The same test bench that was developed to test the high level description of the IC, could be used to test the Verilog implementation merely by using the configuration statement which selects the Verilog based architecture of the IC.

In some cases interoperability is almost a necessity. This is when a design partner provides a module in Verilog, while you are designing in VHDL. Not having an interoperable solution will force one to use conversion programs to translate from one language to other, often with mixed results or some hand editing. This also makes it more difficult to debug the converted portion of the design. Another situation where interoperability is required is when a specific ASIC vendor does not have fully qualified VHDL gate level libraries and no path to map extracted delays from layout to a VHDL netlist. Instead the vendor might have a path using Verilog gate level models and delay calculators.

Fortunately some simulators support VHDL/Verilog interoperability through various co-simulation approaches. Single kernel simulators for both languages are also appearing on the horizon. One has to pay a lot of attention before picking the simulator that supports this interoperability. Performance, ease of use, language compliance, cost and quality of support should be considered before acquiring any simulator.

We will now devote a major portion of this paper describing some of the limitations of VHDL for IC design with special focus on ASIC (Application Specific Integrated Circuit) design. We will also outline some solutions and enhancements to VHDL. Ultimately we believe that enhancing VHDL to fix these limitations will provide maximum flexibility for users and will enable them to exploit the full potential of a powerful language. We will not emphasize the power or features of VHDL as we assume that the audience are fully aware of the language features. Before we discuss these limitations we would like to comment about the current capabilities of commercial synthesis tools as they are an important and integral flow of ASIC design.

3. Current capabilities of commercial Synthesis tools

It is a known fact that most of the commercial synthesis tools produce acceptable results in terms of area and speed of a design only when the input to the synthesis tool is described using a HDL at the RTL (Register Transfer Language) level. Most commercial synthesis tools support both VHDL and Verilog as input HDL to the tool. The current generation synthesis tools support a very limited subset of the above languages to describe the design and to be able to successfully synthesize the input design at the RTL level into a gate level netlist which meets the original design constraints.

So if one were to do a feature by feature comparison of the limited subset of VHDL or Verilog that is acceptable by these synthesis tools, one could pretty much determine that there is not much difference in terms of choosing either one of these languages for an ASIC design. It could be argued that VHDL with its rich set of features like data type abstraction and strict checking of syntax would be a better choice as it would ensure creation of a well structured and comprehensible design.

But from our experience in using both these languages for designing ASICs we have found some limitations/ restrictions in VHDL that make it somewhat less productive toward successful completion of design projects. This is not to say that ASICs cannot be designed in VHDL. There are a very large number of successful design projects that were designed entirely in VHDL. In the next few sections we will illustrate these apparent limitations and suggest some solutions that could lead to increased productivity of using such a rich and powerful language.

4. Limitations of VHDL for ASIC design

The VHDL philosophy in general is to let the designer define all the specific constructs he or she needs: abstract data types, subprograms, generic components, and entities. VHDL can be seen as a kit to build its own hardware description language. As the set of predefined types and operators is very restricted such a possibility becomes a necessity. Any modeling environment should thus include resource packages: general-purpose utility packages as well as application-specific packages [1].

Most of the general-purpose utility packages can be shareable across design groups as well as the entire design community. Examples of general purpose utilities, to mention a few could include:

- Subprograms which overload standard operators for arithmetic functions which can take different types of arguments.
- Conversion functions to convert vectors to integers

and vice versa.

- Miscellaneous procedures/functions to print vectors in various formats like hexadecimal, binary, and octal

In the next few sections we will cover various aspects of the VHDL language as applied for an ASIC design process and point out some deficiencies if fixed would lead to vast increases in productivity. This is not to imply that it is unproductive to design in VHDL! It is meant to enrich an already robust and powerful language. We will show by examples occasionally contrasting with equivalent Verilog implementations to highlight the problem issues.

4.1 Lack of standard arithmetic and conversion functions

One of the most popular general purpose package used by designers and model developers is the '*std_logic_1164*' package developed by IEEE. It is meant for defining a standard in describing interconnection data types used in VHDL modeling. The package specifies a multi-valued logic system which supports both scalar and vector resolved and unresolved types.

Unfortunately there is no corresponding standard package which supports arithmetic and other essential conversion functions of the basic data types of the 'std_logic_1164' package. These functions were left out of the package as they were considered 'orthogonal issues' that were beyond the scope of the package.

But for the real world applications that use VHDL for modeling or chip design these arithmetic and conversion functions are absolutely essential. To fill this important need most synthesis and some simulation vendors provide most commonly used conversion and arithmetic functions. They also provide source code with these packages so that VHDL code that uses procedures from these packages remains portable across simulators.

But the packages may not be portable across synthesis vendors. This is because each synthesis vendor have built in pragmas for arithmetic functions that map them to specific synthesis specific operators. Such vendor specific mapping is widely used in these arithmetic functions provided by synthesis vendors. To get the best results out of synthesis it is important to use the arithmetic packages provided by the respective synthesis vendor. So if one decides to switch to another synthesis vendor considerable work lies ahead in converting the VHDL code to utilize the new packages. All references to conversion functions need to be mapped to the naming specified in the new package.

Solution: A design group could build their own conversion and arithmetic functions and indirectly call the equivalent functions from the synthesis tool vendor's package. So any change to a different synthesis vendor would only require the changes to the packages. This introduces some inefficiency because of the indirection involved in calling the actual conversion function. It would be helpful if VHDL specifies these much required functions as a standard. It would permit each synthesis vendor to add their own compiler directives to direct their synthesis tool. And it would make VHDL portable across synthesis tools.

4.2 Limited Text Input/Output support

Modeling of a digital system and or its components often need support in the form of adequate file input/output handling on the host system. While file I/O might seem irrelevant for hardware design and modeling, it is absolutely essential for test verification and debug of the design. The requirements for file I/O handling could be needed for the following reasons:

- Reading test vectors from a file and applying the vectors to the design under test.
- Capturing vectors from the input/output pads of the design under test. These vectors can be used for creating 'golden' files that can subsequently be used for regression testing after any modification to the design. Or these vectors could be used for production testing of the part after manufacturing.
- Reading macro commands from a file and generating appropriate stimulus for the device under test. Commonly used in bus models like PCI (Peripheral Component Interconnect), VL-Bus (VESA Local Bus) etc. from different vendors.
- Writing simple messages about the current states of internal Finite State Machines for diagnostic or debug purposes.

The above list is definitely a small subset of tasks that are actually needed in a typical ASIC design process involving text/file input output. Writing even simple messages to the terminal or to a file is a big chore in VHDL. Consider the steps involved in printing a message with the following format:

```
<time> "Address : " <address_val> ","
"Data : " <data_val>
```

Where the items enclosed in "<>" are either variables or signals. In the case of <time> we want the current simu-

lation time to be prefixed before the message. A straight forward way to print the above message using VHDL would be:

```
--  package to write hexadecimal numbers
use  work.hex_utils.all;
variable buff_line : line;
variable address : std_logic_vector(31
downto 0);
variable data : std_logic_vector(15 downto
0);
....
write(buff_line, now); --get sim time
write(buff_line, string'("Address : " ));
hexwrite(buff_line, address);
write(buff_line, string'(", Data : "));
hexwrite(buff_line, data);
writeline(OUTPUT, buff_line);
```

A few points are to be noted in the above VHDL code fragment. In the second invocation of the procedure 'write' there is an explicit specification of the string to be printed by using a qualified expression. i.e string'("Address : "). This is because the textio package overloads the write procedure for types STRING and BIT_VECTOR. To resolve any ambiguity we need to qualify the string "Address : " as a string!

Since we wanted the address and data variables to be printed as hexadecimal numbers we need to have appropriate procedures that take a std_logic_vector quantity and write it as a hexadecimal string. This is accomplished by using 'hexwrite' which was probably written by the user or supplied by a vendor as an utility package. Also note that it is important to include the package 'hex_utils' which has the body of the procedure 'hexwrite'.

On the other hand if we are to print the same message in Verilog, it could be accomplished in one line as follows:

```
reg [31:0] address;
reg [15:0] data;
$display($stime, "Address : %h, Data :
%h", address, data);
```

Not only is the description in Verilog concise, it is also very readable (Because of it's 'C' language like syntax). Whereas in VHDL, this seemingly trivial task has gotten mired into the syntax and semantics of the language. The main task of printing this message can be lost and one has to worry about calling the correct procedures, in the correct order and include the correct packages. Now imagine trying to print more complicated diagnostic messages! This discussion is to illustrate the fact that hardware design and verification does not happen purely at a hardware description level. There are many occasions where one needs to do some form of text I/O.

Solution: It is important for design groups or companies to invest in building a large set of utilities that makes it easier for printing basic VHDL data types in different output formats. Third party vendors are also providing such utility kits. It is worth investigating and acquiring such utility packages. It would be very helpful to engineers if VHDL were extended to have similar capabilities as Verilog system tasks for text I/O processing. It would be sufficient to support the basic VHDL data types and the ones included in 'std_logic_1164'.

4.3 No direct support for monitoring internal signals

In VHDL there is no support for observing internal signals of modules in a hierarchy. One example where one needs to monitor internal signals, could be for printing assertion messages whenever a state machine inside the IC goes into an error state. If these assertion messages are placed inside the IC design description, they will be discarded during the synthesis process. The final gate level netlist would not have these assertion messages. In a sense, the debug capability available in behavioral code is lost after synthesis. Instead if we had a way to probe internal signals in the hierarchy, we could place these assertion messages at the test bench level surrounding the IC under development. This will ensure that there is no discontinuity in diagnostic messages displayed during simulation, immaterial of whether the IC is simulated at the behavioral level or at the gate level.

The only way that one can monitor internal signals at the top level, is to bring the required signals as ports traversing the hierarchy of the design so as to be visible at the top level. We will soon illustrate the need for monitoring internal signals. Before that we would like to mention some ways of accomplishing it in VHDL and the corresponding difficulties.

Bringing internal signals to the top level for observing is not practical, because it is not always known up-front the list of signals that are required to be brought to the top level. Besides there is another problem.

- It is a tedious task to bring several signals buried in the hierarchy as ports to the top level, as it involves updates of entity ports, the component declarations, the component instantiation through the entire hierarchy.

Now we will provide a practical example highlighting the need for monitoring internal signals.

In ASIC design methodology vectors are usually captured during functional simulation and subsequently processed to convert them into the format that is acceptable by commercial test equipment. So basic test vector generation is often built into the test bench designed for the device under test (DUT). The test vectors that are generated are also useful for building suites of vectors for regression testing.

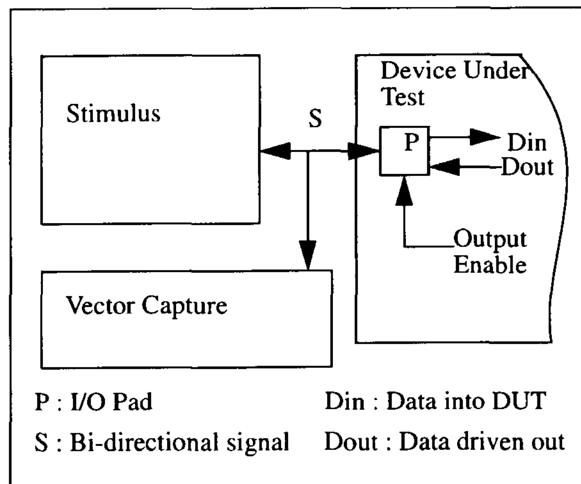

Figure 1. Typical test vector capture

Consider Figure 1, where a typical test bench is shown for the DUT. For simplicity we have shown only 1 bi-directional pad in the DUT, connected to a stimulus driver which could be a sophisticated bus model, purchased from an outside vendor. The designer also drops in a module, which is a monitor for capturing vectors from the signals that are connected to the DUT, during simulation.

These vectors along with the gate level netlist, would eventually be passed on to the ASIC or gate array vendor for 'sign-off' simulation. Since the signal labeled 'S' in the figure is bi-directional, there is presumably an inherent protocol (not shown in the figure) when the 'Stimulus' drives 'S' at some predefined points in time and likewise the DUT drives signal 'S' at some other point in time. At any given time either of the stimulus or the DUT is driving and the other module using the driven value. Even though the Vector Capture module is faithfully recording all transitions on 'S' there is no information for the outside vendor on when to drive the pad 'P' with the specified value or when to strobe for the expected value driven by the DUT. A few solutions to this problem will be presented now.

Pads which drive bi-directional signals have an output enable port which if asserted will drive the value on the outgoing signals. These are 'Output Enable' and 'Out' as shown in Figure 1. If the vector capture module had access to the signal 'Output Enable' it would be possible to determine when to drive the pad 'P' or strobe for a expected value. If 'Output Enable' is not asserted then the pad 'P' is always driven with the value captured during simulation, and if true then strobed for the expected value observed during simulation. If the vector capture module is dropped into the DUT, then it could have access to the 'Output Enable' signal. This might work for RTL level pre-synthesis simulation. But synthesis will discard the vector capture module and it will be dropped in the gate level netlist of the DUT. It is important to capture vectors even at the gate level simulation and compare the vectors with the ones obtained with RTL level simulation. This will help determine if the device is working as per the specification.

Another alternative is for the stimulus to pass on control information to the vector capture module that is as described above. But, if as we mentioned earlier the stimulus was purchased as a generic model from an outside vendor, such information is not usually provided.

Finally a third alternative is to dump all the boundary signals of the DUT, including the output enables of the bi-directional pads into a file and do post-processing of the dump file. Most VHDL simulators support some form of signal changes to be dumped into a file. But there is a problem with this approach. Since the format of the dump file is different for each simulator, it is not possible to write a generic post processing program for vector generation. This solution is not portable across different VHDL simulators.

Clearly the first solution would be the best as it offers a portable solution besides other intrinsic advantages. In Verilog one can observe signals that are buried in the hierarchy at any level. So test vector capture becomes an intrinsic part of the test environment for Verilog users.

Solution: We have shown a work around like capturing all necessary signal traces into a file with the dump feature provided by most simulators. For maximum productivity, VHDL needs to be enhanced to support internal monitoring of signals, as it increases the power of the language for ASIC design flow.

4.4 Accessability of procedures in other entities

VHDL does not permit calling of procedures that are defined in another entity. The procedures should be

either declared and defined locally in an entity or should reside in a package and appropriately made visible through a 'use' clause.

Verilog permits a task (equivalent of VHDL procedure) that is defined in one module, to be used in another module. In Verilog a task can be considered a named object and every named object can be referenced uniquely in its full form by concatenating the names of the modules, tasks, functions, or blocks that contain it [2]. The period character is used to separate each of the names of the hierarchy. We will illustrate an application that makes use of this feature effectively.

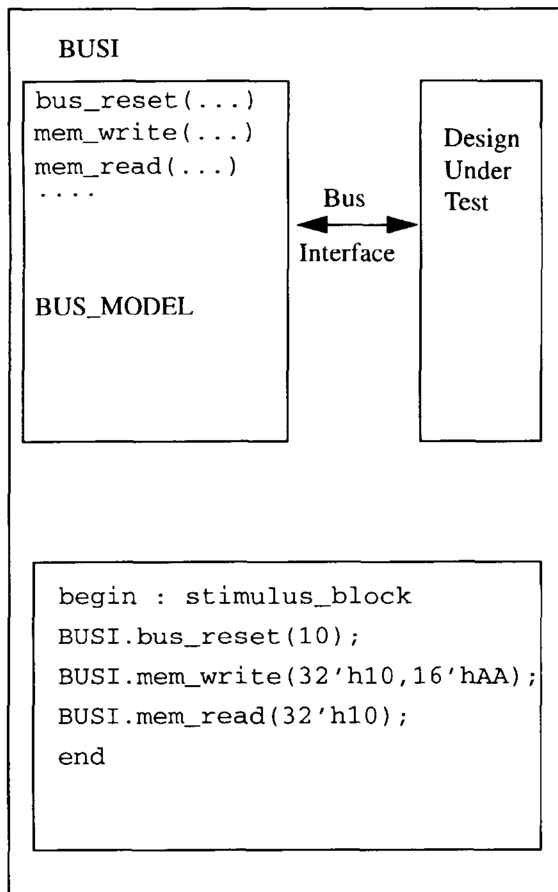

Figure 2. A Verilog test bench that calls procedures from another module

In Figure 2 a Verilog test bench shows a design under test that connects to a model (BUS_MODEL) that emulates a specific system bus. The test bench also includes a block of code that invokes tasks (procedures) which are defined in the module 'BUS_MODEL'. It follows the Verilog convention of specifying the full path name to the tasks that reside inside the BUS_MODEL. i.e. BUSI.bus_reset(10); BUSI is the instance name of BUS_MODEL.

For convenience we will refer to the Verilog 'task' as a procedure. If the procedure 'bus_reset' is called the BUS_MODEL drives the 'reset' signal (which is part of the Bus Interface) for the specified duration which is an argument to the procedure. This 'reset' signal is visible to the 'bus_reset' task which is a port of the module BUS_MODEL. But notice that the Bus Interface signal 'reset' is not specified as an argument in the procedure invocation 'BUSI.bus_reset'. A similar observation is true for the other tasks that are invoked in the 'stimulus_block'. They just deal with the data that is required as an argument to the procedures. They do not specify any signals that need to be driven or observed.

Not having to specify the signals that need to be driven or observed as parameters to the procedure, is a great convenience while writing stimulus patterns. It becomes even more important if the bus model was purchased from an external vendor. Users can take a 'data centric' approach in writing test stimulus. For example if a memory write command needs to be generated by the model to the DUT, one has to only specify the address of the location to write to and the data that needs to be written. Internally the model could use a lot of internal as well as the bus interface signals to accomplish this task based on the bus protocol. Following the 'data centric' approach reduces the learning curve of using the models which are complex for some industry standard busses. Besides one doesn't have to remember the numerous signals that might have to be passed if the procedure expects them as arguments.

Coming back to VHDL, is it possible to emulate the same concept? As it turns out one cannot accomplish this directly. One might mistakenly conclude that all the procedures in the module can be rolled into a package and include this package in the stimulus test bench. This is best explained in the VHDL Language Reference Manual [3]. The text is reproduced as is:

"If a given procedure is declared by a declarative item that is not contained within a process statement, and a signal assignment statement appears in that procedure, then the target of the assignment statement must be a formal parameter of the given procedure or of a parent of that procedure, or an aggregate of such formal parameters"

In other words the only way to avoid passing formal parameters is to declare the procedures locally to a process where the signals in the 'Bus Interface' are in the scope of that process. But this would still not permit encapsulation of the procedures in the bus model.

Solution: One way to reduce the burden of passing a number of signals as arguments to procedures is to group them into a record and pass the record as an argument. With increasing popularity of industry standard bus models, a mechanism that permits invoking procedures using a hierarchical naming mechanism is required in VHDL. This would reduce the burden in creation and usage of bus models and other complex systems.

4.5 No standard Programming Language Interface

Verilog provides a standard Programming Language Interface (PLI) [4]. The PLI mechanism works with utility routines, access routines and data structures that allow user-supplied routines to interact dynamically with the Verilog simulation process and the simulator data structures. The user routines are written in the 'C' programming language. We will describe two useful applications that use the PLI mechanism in Verilog and explore what can be accomplished in VHDL.

Verilog simulators that support PLI allow applications to monitor the value changes of selected objects dynamically during simulation. The access mechanism is well defined and a standard. This implies that an application that uses this access mechanism will run on any Verilog simulator without any change. We will describe how this helps third party tool vendors to develop powerful debugging aids.

Graphical display of simulation results in the form of waveforms remains the most popular debug tool that is used by designers. Most simulators provide a waveform display tool that either show waveforms of simulation results after reading a simulation trace or interactively during simulation. However most of the waveform display tools that come with the simulators do not have good debug capabilities. To fill this need for powerful waveform display and debug tools there are several companies that have such tools, which work in the Verilog environment through the use of PLI access routines, to extract simulation data of objects, from the simulator. They are much more sophisticated and powerful than the wavetool that comes with the simulator.

As the primary focus of EDA companies specializing in simulator tools is in performance of the simulator, there is less emphasis on the features of the wave form tool developed by them. But CAD departments in IC design houses can pick the best Verilog simulator and the best waveform display tool that interfaces with Verilog through PLI and provide a much better solution. This also fits into the model of picking the best 'point tool' for designing complex IC's.

Another example where PLI would be extremely useful is to build powerful and sophisticated parsers for accepting stimulus commands from a file. The parser could be written in 'C' and linked into the simulator using PLI. By allowing this the HDL does not have to support the capabilities that are so popular in high level programming languages like 'C'. And again because the PLI definition and interfacing is a standard in Verilog, the 'C' application that is written by the user is guaranteed to be portable across Verilog simulators provided by many vendors. In the next section we will describe the 'Foreign Language Interface' that is supported in VHDL93 and how it compares with the Verilog PLI.

4.6 Foreign Language Interface in VHDL

The foreign language interface has been standardized in VHDL93. It allows the designer to replace an architecture with code written in languages other than VHDL.

Using this mechanism one can accomplish the same tasks as were mentioned in the previous section. But VHDL does not propose any standards beyond specifying that the architecture of a body is "foreign". For instance there is no defined standard procedure names to access the simulator data structures. While most VHDL simulator vendors provide functions to access and interact with the simulator, they all enforce their own conventions. So any utility routines that someone writes using the foreign language interface, will need considerable re-work when they move to another simulator during a different stage in the design process. But why is this portability important?

There can be several reasons why one would like their design and the surrounding environment to be portable. One VHDL simulator vendor might provide the fastest simulator at the behavioral or RTL level. Another vendor might specialize in accelerating gate level designs in VHDL either through special algorithms or through utilizing hardware accelerators. This would cut down verification time during gate level simulation. Another scenario could be that a design house might have purchased one VHDL simulator, but for ASIC or gate array sign off verification they might have to use a different simulator that is supported by the gate array vendor

(Often referred to as the 'Golden Simulator'). It becomes crucial at this stage to preserve and utilize the investment that went into building the test environment, whether it be in VHDL or through the use of the 'Foreign Language Interface'.

Solution: VHDL committee should extend the 'Foreign Language Interface' by further defining the interface to access and interact with the simulator.

4.7 Limited ASIC library support

One of the most chronic problems that VHDL designers faced was the lack of certified ASIC libraries. This was due to the lack of a standard way to back annotate timing information into a VHDL simulation. Another reason was the speed of VHDL based gate level simulation was not as fast as other gate level simulations.

Both of these issues are being addressed through an industry wide consortium that is backing 'VITAL' - which stands for VHDL Initiative Toward Asic Libraries. Currently revision 3.0 of the VITAL specification is bound for balloting in september, 1995. There are quite a few vendors who have announced ASIC libraries conforming to VITAL standards.

Conclusion

In this paper we have outlined the need for VHDL/Verilog interoperability. We have demonstrated this need by showing examples where VHDL is lacking in some aspects of IC design. We have also suggested solutions in the form of enhancements that are required for VHDL in fulfilling a role of a powerful language that could describe complex systems as well as complex IC's. We have not specified the possible syntax and semantics required for extending VHDL to overcome these limitations. Nor have we deliberated on the feasibility of such extensions within the current definition and scope of the VHDL language. These tasks are beyond the scope of this paper. Where applicable we have suggested work arounds to handle these limitations. All the problems that we have listed are ones that we ran into during the design of commercial ASICs that have been designed entirely in VHDL.

Acknowledgments

We would like to thank Arun Vaidyanathan of Future Integrated Systems and Jagannath Raghavendran of Zycad Corporation for providing feedback about this paper. We would also like to thank Krishna Kumar of Cadence Design Systems for his suggestions.

References

1. "VHDL Designer's Reference"
   Jean-Michel Berge, et. al
   Kluwer Academic Publishers, 1992.
2. "Verilog-XL Reference"
   Cadence Documentation
3. "IEEE Standard VHDL Language Reference Manual"
   IEEE Std 1076-1987, pp 8-5, March, 1988
4. "Programming Language Interface Reference"
   Cadence Documentation.