

Verilog: Dialect of VHDL?

John Willis
FTL Systems, Inc.
Rochester, MN
jwillis@ftlsys.com

Gabe Moretti
VeriBest, Inc.
Boulder, Colorado
gmoretti@ingr.com

Paul Menchini
Menchini & Associates
RTP, NC
mench@mench.com

This work was partially supported by the Advanced Research Projects Agency under contract DABT63-96-C-004 with FTL Systems. The views and opinions expressed are those of the authors.

1.0 Abstract

For the next few years, the coexistence of VHDL and Verilog is inevitable within the user community. Existing investments in designers, designs and tools preclude any rapid changes from either language to the other. However, the continued development of separate VHDL and Verilog tools increases the costs to end users.

In order to reduce the cost of supporting two languages and allow reallocation of investment toward greater functionality, this paper examines the implications of analyzing VHDL and Verilog into a common intermediate form based on a superset of VHDL semantics.

We identify the semantic extensions needed to support such a hybrid intermediate format. The most significant changes are to the underlying temporal semantics and to support Verilog's globally visible naming schema. Additions to the type system, programming interface and sequential execution model are modest.

The paper concludes that there is enough overlap between VHDL and Verilog semantics that developers of new tools should seriously consider developing bilingual tools (See Figure 1).

2.0 Introduction

Verilog is a language developed with the specific purpose of modeling hardware devices. This narrow objective allows Verilog to use a much more abstract simulation model than VHDL. The latter, although it

contains the words "Hardware Description" in its name, makes very few assumptions as to the nature of the system being simulated. Therefore, VHDL forces its users to carefully specify the nature of the system being simulated by defining types and dependencies. In addition, VHDL makes no assumptions as to the method by which steady state is achieved in the system being simulated; thus, its scheduling mechanism differs considerably from the one used by Verilog.

This paper takes a top-down approach to Verilog, beginning with a discussion of structural hierarchy. The structural hierarchy expands into low-level modules and inter-module communication mechanisms. Verilog's temporal semantics differ markedly from VHDL's. Verilog's hardware primitives and data types are largely predefined, leading to relatively simple statement and expression constructs. Verilog's compiler control commands are intrinsic to the language, more like Spice than VHDL. System tasks and a programming language interface provide much of Verilog's behavioral extensibility.

There are several operational definitions of Verilog. Initially Verilog was defined by the behavior of Gateway Design's (and later Cadence's) Verilog-XL[™] product. In 1995, an intense IEEE standardization effort resulted in IEEE Std 1364 (Verilog). This paper is based on Std 1364 [1] and the latest revision of VHDL IEEE Std 1076-93 [2].

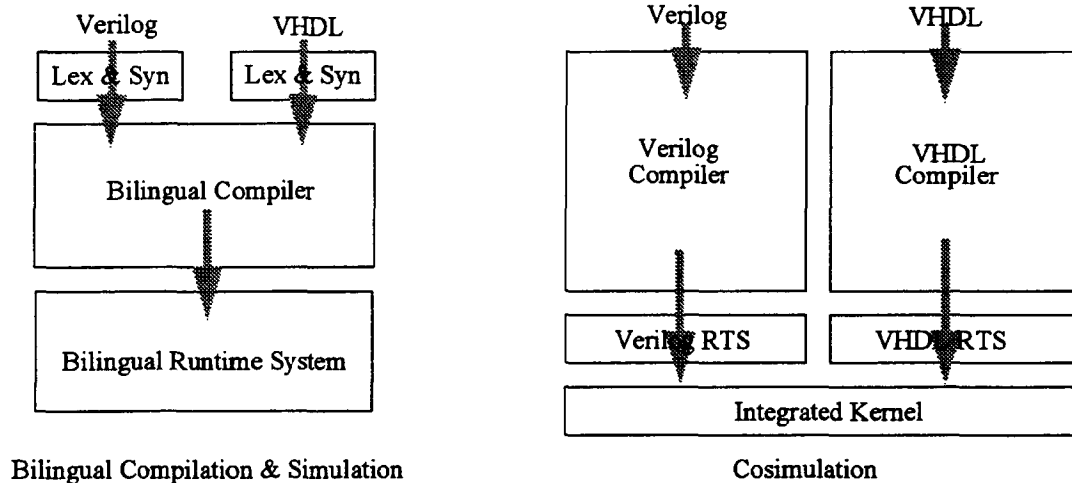


Figure 1. Comparison of bilingual and cosimulation approaches (note degree of sharing)

This paper also focuses exclusively on the semantics of Verilog and VHDL. Lexically and syntactically, the two languages have different antecedents (respectively, C and Ada) and differ greatly. However, when considering a modern, optimizing compiler, the lexical and syntactic analysis phases represent but a few percent of the total development cost. A much greater portion of the development cost relates to optimizing transformations, optimizing code generation, parallelization and runtime support. In all four areas, the degree of semantic commonality is of much greater significance in reducing development cost than is the lexical or syntactic differences between Verilog and VHDL.

3.0 Structural Hierarchy

Verilog defines structural hierarchy in terms of *modules*. Higher-level modules can create instances of lower-level modules, much as a VHDL entity-architecture pair (or *design entity*) can instantiate (via component instances and bindings) other design entities to form a design hierarchy.

Conceptually, Verilog modules can be viewed as a subset of VHDL's design entity construct. Each module has but a single implementation (rather than, as in VHDL, allowing multiple architectures).

Within a module, instances of other modules or predefined hardware components may be created with less flexibility than is allowed by VHDL's binding mechanism. Verilog's module binding is immediate (during

compile time), rather than deferred until elaboration time as it is in VHDL.

The association of module interfaces is also more restrictive than what VHDL permits. Modules use simple positional or named association; there is no provision for formal subelement association.

Module interfaces contain only ports. *Defparam* statements can be used as a substitute for VHDL's generics, but reach across hierarchy boundaries, using a path, to alter the value of parameters within an instantiated module. This ability to reach across module boundaries during expansion of a design hierarchy (analogous to elaboration in VHDL terms) has no conceptual equivalent in VHDL.

Verilog's module instantiation statement allows the creation of arrays of instances, providing a subset of the functionality of VHDL's generate statement.

Within a module, Verilog's procedural statements are analogous to VHDL's process statement. An *always* statement begins execution at the onset of simulation and executes repetitively. In contrast, the *initial* statement executes once at the beginning of simulation, corresponding to a VHDL process ending in a single wait statement containing no resumption clauses.

Verilog's parallel block has no semantic equivalent in VHDL. It resembles a concurrent block in that multiple threads may concurrently execute within the block; however, unlike VHDL, control flows in and out of the parallel block. The Verilog parallel block may be

embedded as a statement within other sequential or parallel blocks. The full semantics of Verilog parallel blocks may require dynamic creation and deletion of threads, a capability which is not required in a VHDL runtime system.

Verilog *tasks* and *functions* are loosely analogous to VHDL *procedures* and *functions* (respectively). In Verilog, all concurrently enabled instances of a task share a common set of local variables; in VHDL, each call to a procedure or function dynamically elaborates the sub-program's variables. Hence, a Verilog implementation may use statically allocated storage for task or function locals, whereas a VHDL implementation must use a mechanism similar to stack frames. The Verilog storage paradigm results in simpler and more efficient storage allocation at the expense of greatly increased presence of side effects (simulation artifacts). For example, several instances of the same task may communicate via their common local variables, resulting in many different execution results depending on how task execution is interleaved.

4.0 Inter-Module Communication

Verilog modules communicate via ports connected by *nets* and *registers*, or expressions thereof. Expression and name semantics permitted in this context closely resemble VHDL's bit slice, subscripting names and concatenation operators. (Since Verilog has no records, record field selection does not occur in Verilog.)

The "value" of a Verilog net at any point of simulated time is determined by the value of all active drivers to the net, and, in the case of a net having a *trireg* type, also the drive history of the net. Verilog nets most closely resemble a VHDL signal, although details of the drivers and net differ enough to require distinct treatment during optimization, code generation, and simulation.

Unlike VHDL signals, a Verilog net can, in sequence, assume more than one value at the same simulated time (and, in VHDL terms, during the same simulation cycle). This characteristic permits non-deterministic simulation results. Non-determinism greatly complicates compile-time analysis, however the additional flexibility can accelerate uniprocessor execution.

A variety of Verilog net types are predefined, including wire, several versions of wired and tristate logic, and "constant" supply signals. However, no user-defined nets may be defined. All of these variants are readily

handled by compilers and runtime systems capable of dealing with VHDL's semantics.

Verilog's capacitive networks, comprising one or more *trireg* nets, are probably the most complex Verilog net construct to introduce into a VHDL environment. Such nets require analysis beyond that typically applied to VHDL bidirectional ports and buses.

Verilog's register construct, abbreviated *reg*, most closely resembles a VHDL shared variable adhering to the IEEE Std 1076-1993 shared variable semantics. (Such shared variables are a source of non-algorithmic semantics.) Once assigned a value, registers retain the assigned value until the next assignment.

The resolution behavior of two or more values driven onto a net at the same time is predefined in Verilog, whereas VHDL leaves the specification of the resolution method up to the VHDL modeler. Verilog's intrinsic resolution functionality can readily be handled as a subset of VHDL's existing mechanisms. Existing implementations of the IEEE Std 1164 standard logic package[3] or Vital[4] suggest some approaches, although details of Verilog's logic system are slightly different than the logic system used in either of these VHDL extensions.

In summary, the greatest complexity associated with interpreting Verilog interconnects as a VHDL dialect results from the non-deterministic assignments to Verilog nets and registers. Analysis and code generation techniques required to handle these cases closely resembles techniques needed to handle VHDL-93's shared variables.

5.0 Temporal Semantics

The IEEE standardization of Verilog did much to formalize Verilog's temporal semantics. IEEE Std 1364 describes Verilog's temporal semantics in terms of connected threads of execution (or processes). These threads can be evaluated in response to events visible at their inputs, potentially updating internal state and generating outputs events.

Algorithms for queueing pending (future) events are different in VHDL and Verilog. Verilog "stratifies" future events into five different categories based on when a given event becomes active. Once a Verilog event is queued, it eventually occurs (unless the entire simulation is terminated prior to the event's maturation). VHDL uses several algorithms, modeling inertial and

transport delay, to queue pending events. Future additions to a VHDL driver's pending event queue may remove or modify events already on the queue. Details of the Verilog and VHDL pending event mechanisms require distinct treatment all the way through optimization, transformations, code generation and runtime.

Verilog allows communication of events between processes such that different behaviors may result from the same initial conditions. Such differences result from the order in which events are removed from the active event queue and the way in which concurrent evaluations are interleaved. This flexibility can lead to non-determinism and race conditions within the model. VHDL-93's shared variables lead to similar timing complications, however proposed changes to VHDL's shared variables under IEEE PAR 1076a [6] are expected to reduce opportunities for non-determinism and race conditions in VHDL. Today, Verilog and VHDL implementations must deal with very comparable non-determinism and race condition issues.

A single integer describes simulation time in Verilog, in contrast to VHDL's delta cycles and physical time units (beginning with FS). Bilingual compilation is made simpler if Verilog's stratification of events at the current time maps into VHDL delta cycles and Verilog's explicit time maps into VHDL's physical type TIME.

Verilog defines a <minimum / typical / maximum> tuple to represent temporal delay, in contrast to the simpler scalar subtype provided by VHDL (DELAY_LENGTH, a predefined subtype of the physical type TIME). This tuple forms an argument to Verilog gate delays (semantically resembling VHDL's wait statement), nets (semantically resembling a VHDL resolution with intrinsic delay), and continuous assignments (semantically resembling the after clause within a VHDL signal assignment).

Verilog provides powerful capabilities by which a *disable* statement executing in one thread may terminate the activity of other tasks or blocks, often asynchronously with respect to the terminated statement sequence. In its simplest form, disable is analogous to VHDL's *next* and *exit* statements. Whereas next and exit are executed synchronously with the statement sequence being disabled, Verilog's disable has much more powerful capability for handling exception situations such as hardware interrupts and resets. The asynchronous potential of a Verilog disable statement greatly complicates global, compile-time optimization and parallelization.

6.0 Hardware Primitives

Verilog includes both a large number of predefined structural primitives and a mechanism by which users may define their own simple primitives (user-defined primitives). Many of these primitives closely resemble VHDL's Vital[4] library (in fact, many aspects of Vital were drawn from Verilog).

Verilog's predefined gate and switch primitives include:

- logical gates (and, nand, nor, or, xnor, xor),
- buffer gates (buffer and inverter),
- tristate gates (buffer and inverter),
- pullup and pulldown primitives,
- MOS switches (CMOS, NMOS, PMOS, etc.),
- bidirectional switches (transmission gates, etc.).

Of these, only bidirectional switches require semantic extensions to a VHDL simulator.

Verilog also supports user-defined primitives (UDPs) and programmable logic array (PLA) models. The constructs supporting Verilog UDP and PLAs are a subset of those available within a VHDL process. With respect to Verilog UDP and PLAs, Verilog is truly a subset of VHDL.

7.0 Data Types

Verilog is based on a much simpler type system than VHDL. All of Verilog's base types are predefined (although users may define the extent of an array). Only bidirectional net types, connected to forms of transmission gates, present new complexity relative to existing VHDL functionality.

8.0 Sequential Statements

Verilog's sequential statements are largely a subset of those provided by VHDL. There are no significant differences between Verilog and VHDL *if* statements.

Verilog *case* statements have several complexities not found in VHDL. In Verilog the case alternatives need not be exclusive; each case alternative must be considered in the order given. The Verilog 'x' and 'z' values lead to 2 *case* variants, *casez* and *casex*, in which some elements in a case selector expression are not considered when choosing a case alternative. There is no direct equivalent in VHDL. The optimization and code generation impacts of this additional functionality are modest.

In Verilog's looping construct, the iterator control flow follows a much more flexible form than that provided by VHDL. The initial assignment, loop termination condition and step assignment can be more flexible than VHDL's discrete range. Verilog's loop iterator is also declared external to the loop; in VHDL, the loop iterator is local to a declarative region formed by the loop statement. Again, the additional Verilog complexity does not significantly complicate optimization or code generation relative to that necessary for VHDL.

Verilog allows three different constructs controlling the execution of procedural statements: delay control, event control and wait statements. All three can generally be expanded into the semantics of VHDL wait statements; except that one must consider Verilog's broader visibility into events occurring elsewhere in the model. In VHDL, only those events occurring on signals local to the design entity containing the process or on global signals can affect the execution of the process. Intra-assignment timing closely resembles VHDL's *after* clauses within a waveform element.

9.0 Expressions

Expressions in Verilog generally resemble those in VHDL. Significant differences are the intrinsic value system and Verilog's reduction operators. Verilog's intrinsic operators utilize a 4-valued system (0, 1, x and z), whereas VHDL's bit data type is 2-valued (0 and 1). Use of 4-valued logic system leads to the additional of reduction operators in Verilog. Verilog also includes a conditional operator, resembling C's conditional operator.

None of these differences add substantial complexity to a VHDL optimizer, transformations, code generation or runtime. VHDL implementations already require operations on arbitrary enumerated types. An internal construct such as the conditional operator is already strongly suggested by VHDL's rules for concatenating null arrays.

10.0 System Tasks & Functions

Both information about and control of the simulation environment is manifest in Verilog; whereas, in VHDL, such information and control is essentially external to the language.

Verilog's *\$stop* and *\$finish* system tasks terminate execution with varying degrees of information and control.

In VHDL, assertions with FAILURE severity often accomplish the same objective; however, such behavior is not prescribed within the VHDL language specification.

Numerous timing checks, such as *\$setup*, *\$skew*, *\$recovery* and *\$nochange* are predefined in Verilog as system tasks, whereas such functionality must be explicitly coded in VHDL (by the user or as a predefined library such as the Vital library [4]).

In like fashion, there is no inherent complexity associated with implementing Verilog system tasks related to PLAs, formatted I/O or simulation time within a VHDL framework. (Indeed, there are several commercial and freely available VHDL packages providing at least a subset of such capabilities.) Verilog's random number functionality suggests some form of shared variable to retain a common seed (already addressed in VHDL-93 and by the VHDL mathematics packages [7]). Verilog's access to wall clock time (*\$realtime*) can be implemented in VHDL using a foreign subprogram that calls the appropriate routines in the underlying operating system.

In summary, the semantics of Verilog system tasks and functions are readily handled by VHDL.

11.0 Value Change Dump File

Verilog includes the specification of a *value change dump (VCD) file* containing a record of changes to selected variables. During simulation, execution of value-change system dump tasks appends information to these dump files.

Although the standard definition of VHDL has no such dump file, WAVES[5] files are sometimes provided by VHDL simulators. Although both are ASCII files, the data structures and paradigms are entirely different. There are no inherent reasons why VCD files could not be written using VHDL-like processes calling VHDL's TEXTIO package.

12.0 PLI

An extensive programming language interface provides much of Verilog's personality. A large library of Verilog internal access routines (ACC) provide information and control of models and simulation.

In general, the Verilog PLI maps into the VHDL foreign subprogram interface (FLI). However the ACC routines

require access to the compiler and simulation internals, not normally available to the FLI. The ability of Verilog names to cut across hierarchical boundaries using the PLI complicates the optimization and parallelization of Verilog.

13.0 Conclusions

In many ways, Verilog can be viewed semantically as a dialect of VHDL. A great deal of the optimization, transformation, code generation and runtime functionality required by a VHDL simulator can be leveraged to create a Verilog simulator.

The critical semantic differences between Verilog and VHDL relate to visibility and temporal semantics. Aspects of Verilog's visibility rules and temporal semantics both complicate optimization (in isolation) and require substantial new functionality when Verilog capability is added to an existing VHDL compiler.

Analysis of the benefits to a hardware designer's productivity of these Verilog capabilities is beyond the scope of this paper. Nevertheless, the cost of these features or lack thereof should be studied.

Verilog allows references to nets and registers that cross hierarchy boundaries in almost arbitrary ways. When this flexibility is actually used, a compiler is forced to perform global analysis, where, without such use, local analysis would have sufficed. Forced transition to global analysis generally requires more compiler memory, more compilation time and, in practice, results in less optimal results.

Verilog's more abstract timing model results in additional degrees of execution freedom. This freedom often complicates compile-time analysis but can yield high execution performance on a uniprocessor. If parallel implementations are not constrained to mimic the incidental reference interleaving of a uniprocessor, parallel performance can also improve.

Verilog is not strictly a semantic dialect of VHDL; however, the exceptions are small enough that development of a bilingual tool is well worth considering, in order to maximize the return on scarce development resources.

We are left with a few key questions: Is Verilog the best way to describe a specific modeling style? Does Verilog suggest a way in which VHDL models could be written to increase designer productivity? The complements to these questions are also of interest.

14.0 Disclaimer

This paper is intended to convey generally useful technical ideas of interest to the VHDL and Verilog community. Whereas both VeriBest, Inc. and FTL Systems, Inc. have both VHDL and Verilog capability, this paper has no direct relationship to specific planned or current commercial products.

15.0 Bibliography

- [1] Verilog Hardware Description Language Reference Manual, IEEE Std 1364 (Draft of April 1995), The Institute of Electrical and Electronics Engineers, New York, NY, 1995.
- [2] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076, The Institute of Electrical and Electronics Engineers, New York, NY, 1994.
- [3] IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164), IEEE Std 1164, The Institute of Electrical and Electronics Engineers, New York, NY, 1995.
- [4] IEEE Standard VITAL ASIC Modeling Specification, IEEE Std 1076.4, The Institute of Electrical and Electronics Engineers, New York, NY, 1995.
- [5] IEEE Standard for Waveform and Vector Exchange (WAVES), IEEE Std 1029.1, The Institute of Electrical and Electronics Engineers, New York, NY, 1991.
- [6] VHDL Shared Variable Language Change Specification (PAR 1076a), Version 4.1, The Institute of Electrical and Electronics Engineers, New York, NY, June, 1995.
- [7] Standard VHDL Mathematics Packages, IEEE Std 1076.2, The Institute of Electrical and Electronics Engineers, New York, NY, 1996.