Comparison of VHDL, Verilog and SystemVerilog

Stephen Bailey
Technical Marketing Engineer
Model Technology

Model Technology
A MENTOR GRAPHICS COMPANY

Introduction

As the number of enhancements to various Hardware Description Languages (HDLs) has increased over the past year, so too has the complexity of determining which language is best for a particular design. Many designers and organizations are contemplating whether they should switch from one HDL to another.

This paper compares the technical characteristics of three, general-purpose HDLs:

- VHDL (IEEE-Std 1076): A general-purpose digital design language supported by multiple verification and synthesis (implementation) tools.
- Verilog (IEEE-Std 1364): A general-purpose digital design language supported by multiple verification and synthesis tools.
- SystemVerilog: An enhanced version of Verilog. As SystemVerilog is currently being defined by Accellera, there is not yet an IEEE standard.

General Characteristics of the Languages

Each HDL has its own style and heredity. The following descriptions provide an overall "feel" for each language. A table at the end of the paper provides a more detailed, feature-by-feature comparison.

VHDL

VHDL is a strongly and richly typed language. Derived from the Ada programming language, its language requirements make it more verbose than Verilog. The additional verbosity is intended to make designs self-documenting. Also, the strong typing requires additional coding to explicitly convert from one data type to another (integer to bit-vector, for example).

The creators of VHDL emphasized semantics that were unambiguous and designs that were easily portable from one tool to the next. Hence, race conditions, as an artifact of the language and tool implementation, are not a concern for VHDL users.

Several related standards have been developed to increase the utility of the language. Any VHDL design today depends on at least IEEE-Std 1164 (std_logic type), and many also depend on standard Numeric and Math packages as well. The development of related standards is due to another goal of VHDL's authors: namely, to produce a general language and allow development of reusable packages to cover functionality not built into the language.

VHDL does not define any simulation control or monitoring capabilities within the language. These capabilities are tool dependent. Due to this lack of language-defined simulation control commands and also because of VHDL's user-defined type capabilities, the VHDL community usually relies on interactive GUI environments for debugging design problems.

Verilog

Verilog is a weakly and limited typed language. Its heritage can be traced to the C programming language and an older HDL called Hilo.

All data types in Verilog are predefined in the language. Verilog recognizes that all data types have a bit-level representation. The supported data representations (excluding strings) can be mixed freely in Verilog.

Simulation semantics in Verilog are more ambiguous than in VHDL. This ambiguity gives designers more flexibility in applying optimizations, but it can also (and often does) result in race conditions if careful coding guidelines are not followed. It is possible to have a design that generates different results on different vendors' tools or even on different releases of the same vendor's tool.

Unlike the creators of VHDL, Verilog's authors thought that they provided designers everything they would need in the language. The more limited scope of the language combined with the lack of packaging capabilities makes it difficult, if not impossible, to develop reusable functionality not already included in the language.

Verilog defines a set of basic simulation control capabilities (system tasks) within the language. As a result of these predefined system tasks and a lack of complex data types, Verilog users often run batch or command-line simulations and debug design problems by viewing waveforms from a simulation results database.

SystemVerilog

Though the parent of SystemVerilog is clearly Verilog, the language also benefits from a proprietary Verilog extension known as Superlog and tenants of C and C++ programming languages.

SystemVerilog extends Verilog by adding a rich, user-defined type system. It also adds strong-typing capabilities, specifically in the area of user-defined types. However, the strength of the type checking in VHDL still exceeds that in SystemVerilog. And, to retain backward compatibility, SystemVerilog retains weak-typing for the built-in Verilog types.

Since SystemVerilog is a more general-purpose language than Verilog, it provides capabilities for defining and packaging reusable functionality not already included in the language.

SystemVerilog also adds capabilities targeted at testbench development, assertion-based verification, and interface abstraction and packaging.

Pros and Cons of Strong Typing

The benefit of strong typing is finding bugs in a design as early in the verification process as possible. Many problems that strong typing uncover are identified during analysis/compilation of the source code. And with run-time checks enabled, more problems may be found during simulation.

The downside of strong typing is performance cost. Compilation tends to be slower as tools must perform checks on the source code. Simulation, when run-time checks are enabled, is also slower due to the checking overhead. Furthermore, designer productivity can be lower initially as the designer must write type conversion functions and insert type casts or explicitly declared conversion functions when writing code.

The $1,000,000 question is this: do the benefits of strong typing outweigh the costs?

There isn't one right answer to the question. In general, the VHDL language designers wanted a safe language that would catch as many errors as possible early in the process. The Verilog language designers wanted a language that designers could use to write models quickly. The designers of SystemVerilog are attempting to provide the best of both worlds by offering strong typing in areas of enhancement while not significantly impacting code writing and modeling productivity.

Language Feature Comparison

The following table presents a feature-by-feature comparison of the three HDLs. Note that the purple font color differentiates Verilog 2001 features from Verilog 1995 features.

	VHDL	Verilog (2001)	SystemVerilog
Strong typing	**Yes**	**No** • Bit • bit-vector • wire • reg) • unsigned • **signed** • integer • real • String in certain contexts only	**Partial** Not strongly typed in areas backward compatible with Verilog **Yes** Enhanced type system is strongly typed (but not as strong as VHDL)
User-defined types	**Yes**	**No**	**Yes**
Dynamic memory allocation (pointer types)	**Yes**	**No**	**Partial** Class objects can be dynamically created/destroyed, but via handles ("safe pointers")
Physical types	**Yes**	**No**	**No**
Named events	**No**	**Yes**	**Yes**
Enumerated types (FSM modeling)	**Yes**	**No**	**Yes**
Records/structs	**Yes**	**No**	**Yes**
Variant/unions	**No**	**No**	**Yes**
Associative/sparse arrays	**Partial** (But can be modeled using access types)	**No**	**Yes**
Class/inheritance	**No**	**No**	**Yes** (single inheritance)
Data packing	**No**	**No**	**Yes**
Bit (vector) / integer equivalence	**Partial** Not built-in but standard package supports	**Yes**	**Yes**
User defined signal/net resolution	**Yes**	**No**	**No**
Subprograms (procedural)	**Yes** Function & procedure always automatic	**Yes** Static and **automatic** functions and tasks	**Yes** Same as Verilog plus void functions (procedures)
Subprograms (concurrent) aka tasks	**Yes** Concurrent procedure calls	**Yes** Static tasks	**Yes** Static tasks
Methods	**No**	**No**	**Yes** (goes hand-in-hand with classes)
Separate packaging	**Yes** Packages	**Yes** Include files	**Yes** Include files

continues on pg 4

continued from pg 3

	VHDL	Verilog (2001)	SystemVerilog
Other hierarchy	**Yes** Separate entity / architecture (Interface / implementation)	**No**	**Yes** Programs, Clocking domains, Interfaces
All-read sensitivity	**No**	**Yes** **@(*)**	**Yes** Same as Verilog. Plus: always_comb
Reactive region processes	**Yes** Postponed processes	**No**	**Yes** Programs, Clocking domains, Final blocks
Dynamic process creation/deletion	**No**	**Yes** Fork/join. Block/task disable.	**Yes** Same as Verilog.
Conditional statements	**Yes** • If-then-else/elsif (priority) • Case (mux) • Selected assign (mux) • Conditional assign (priority) • No "don't care" matching capability	**Yes** • if-else (priority) • case (mux) • casex (mux) • ?: (conditional used in concurrent assignments)	**Yes** Same as Verilog. Adds priority and unique keywords to infer priority encoding/mux implementation
Iteration	**Yes** • Loop • while-loop • for-loop • exit • next Can name the loop to exit or continue with next	**Yes** • repeat • for • while	**Yes** Same as Verilog, Plus: • do-while • break • continue Only closest enclosing loop can be break or continue
Operators & expressions	**Yes** All expected: • arithmetic • logical • bit-wise • shift • concatenation Overloadable (polymorphism). No unary reduction. No logical scalar/vector.	**Yes** All expected: • arithmetic • logical • bit-wise • shift • concatenation • unary reduction • logical scalar/vector • case (in)equality. • conditional (?:) No rotate left/right	**Yes** Same as Verilog. Plus: • wild (in)equality • increment • decrement • assignment (+=, -=, |=, etc.) No rotate left/right
Gate level modeling	**Yes** VITAL. Very good FPGA library support.	**Yes** Builtin primitives. UDPs. Better availability of ASIC library support	**Yes** Same as Verilog. Except, library support yet to be qualified as vendors won't assume Verilog sign-off = SystemVerilog sign-off
Interface abstraction	**Partial** Component abstracts interface from specific module. Two layer binding allows flexibility in generic/port mapping.	**No**	**Yes** Interfaces are a separate construct in language. Supports multiple abstraction level and eases interface reuse. Can reduce coding.

continues on pg 5

continued from pg 4

	VHDL	Verilog (2001)	SystemVerilog
Configuration & Binding	**Yes** Control of instance or component binding to entity. Incremental (re)binding of generics and ports.	**Partial** **Control of module to instance binding.**	**Partial** Same as Verilog.
Conditional & iterative generation	**Yes** • If (conditional) • For (iterative)	**Yes** • **If** • **if-else (mutually exclusive)** • **case** • **for**	**Yes** Same as Verilog.
Attributes	**Yes** Attributes are typed. Attribute values can be specified. Attribute values can be referenced. Anything labeled with a name can be attributed. Groups allow attributes to relate two or more named entities in the design.	**Partial** **Not-typed.** **Can be placed virtually anywhere.** **What is attributed is determined by lexical proximity.** **Attribute values cannot be referenced.**	**Partial** Same as Verilog.
Verification targeted capabilities	**Partial** • Access types • Recursive subprograms • Extensive File I/O • Postponed processes • Standard package for random number generation	Limited • File I/O • Random number generation • Recursive subprograms • Fork/join	**Yes** Same as Verilog. Plus: • Random and constrained random value generation • Programs • Clocking domains • Associative arrays • Semaphores • Mailboxes • Classes
Assertions	**Partial** • Combinatorial (Boolean) assertions • User-defined severity and message control	**No**	**Yes** • Combinatorial and sequential (concurrent) assertions. • Sequence (temporal) expression. • Sequence-local variables. • User-defined severity and message control. • API extensions for assertions and coverage information for assertions
Foreign interfaces	Limited • Standard 'Foreign attribute • VhPI defined, but not yet standardized	**Yes** Standard C API (tf, acc, vpi)	**Yes** Same as Verilog. Plus: • Extensions to API for assertions and coverage • Direct C language interface

Summary

With all of the recent publicity surrounding languages and standards, many people are wondering where to go next. The answer to this question will vary greatly by designer and organization. In addition to the language feature comparison above, here are some final points to consider:

- SystemVerilog is an emerging standard that is still evolving. With a compelling set of features, SystemVerilog is the likely migration path for current Verilog users. However, widespread tool support won't be available until the specification stabilizes.
- For VHDL users, many of the SystemVerilog and Verilog 2001 enhancements are already available in the VHDL language. There is also a new VHDL enhancement effort underway that will add testbench and expanded assertions capabilities to the language (the two areas where SystemVerilog will provide value over VHDL 2002). Considering the cost in changing processes and tools and the investment required in training, moving away from VHDL would have to be very carefully considered.

For more information, call us or visit: www.model.com

Model Technology
A MENTOR GRAPHICS COMPANY

Corporate Headquarters
Model Technology
8005 S.W. Boeckman Road
Wilsonville, Oregon 97070 USA
Phone: 503-685-0824

Mentor Graphics

Corporate Headquarters
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, Oregon 97070 USA
Phone: 503-685-7000