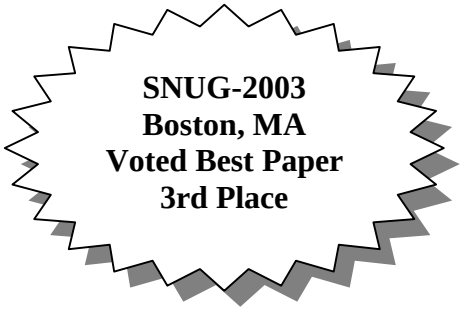
SystemVerilog - Is This The Merging of Verilog & VHDL?

Clifford E. Cummings

Sunburst Design, Inc.

cliffc@sunburst-design.com



ABSTRACT

In his EE Times Industry Gadfly Column, ESNUG moderator, John Cooley, set off a firestorm with his article entitled, "VHDL, the new Latin,[13]" in which he offers a quote from Aart de Geus "that SystemVerilog will be the dominant language."[14]

But comparing VHDL to Latin begs the question: Is VHDL "the new Latin" because it is a dead language or because it contributed to the creation of a new language?

Although SystemVerilog[17][18] is a fully backward compatible superset of Verilog-2001[10], it also incorporates many of the best features of VHDL. Indeed, one of the unspoken goals for SystemVerilog was to incorporate more VHDL-like features to make translation into and co-simulation with SystemVerilog easier. This paper shows how features that used to be unique to VHDL have found their way into SystemVerilog.

1.0 Introduction

I have been very active in Verilog design and standards activities for over a decade, have done both Verilog and VHDL synthesis design and have done detailed examinations of the virtues and pitfalls of Verilog and VHDL. Based on my experience, I believe that SystemVerilog is a revolutionary step forward in the evolution of hardware description and verification languages. In this paper I will discuss the many features of SystemVerilog that were inspired by VHDL.

VHDL is not a dead language. VHDL, along with Verilog, lives in a powerfully enhanced HDL called SystemVerilog.

Should SystemVerilog really be called SystemHDL? Perhaps. But since the core syntax of the language is rooted in the Verilog HDL, the name of the enhanced language bears the Verilog moniker. Any VHDL engineer that claims that SystemVerilog should really be called SystemHDL will receive little argument from me.

2.0 VHDL Features - Not in Verilog - Added to SystemVerilog

2.1 Strong Data Typing

Strong typing was never a feature of Verilog. Because it only supported scalar data types, it was possible for the language to perform (most of the time) the correct type conversions automatically, albeit with a potential loss of accuracy or precision. For example, integer, bit and real values were correctly and automatically converted when needed. Incorrect conversion would occur when large values had to be truncated or when the actual value did not have a numerical interpretation – such as ASCII characters.

Design engineers appreciate the lack of strong typing. They like to deal with values and expressions of various types and bit width without requiring complex conversion operations to satisfy the compiler. This convenience comes at the risk of specifying expressions that may not yield correct results for all possible input values. But these risks can be mitigated by the use of linting or formal checking tools. SystemVerilog maintains the ability to automatically convert between scalar and some aggregate types. Packed *struct* and *union* aggregate types have a well-defined translation to and from scalar bit vectors and can thus be automatically converted.

SystemVerilog does offer strong data typing with the higher-level data types. Unpacked *struct* and *union* as well as *class* cannot be freely mixed amongst themselves or with scalar data types. The *cast_assign* operator allows a user to perform an explicit type conversion. Otherwise, only instances of compatible types can be assigned to variables or specified as actual argument values.

2.2 Time Units

Time values in VHDL were always specified with explicit units. Verilog only allowed the specification of time as a number of units, where units were determined by the last *`timescale*

directive. To avoid compile-order timing problems, Verilog guidelines specified that any module containing timing delays should have a *`timescale* directive.

SystemVerilog has the ability to add a VHDL-like unit to a time delay. There must be no space between the time value and the unit suffix. For example “1ns” is correct whereas “1 ns” is incorrect.

SystemVerilog also retains full backward capability with existing *`timescale* directives. SystemVerilog also adds the new *timeunit* and *timeprecision* keywords, which can be added to a module to replicate the *`timescale* functionality locally to a module.

2.3 Enumerated Types

Without enumerated types, Verilog engineers had to settle with parameters, which required assignment when the parameters were declared or substitution with preprocessor symbols. VHDL could always easily display enumerated names in a waveform display while Verilog engineers had to test the contents of a bus in a case statement and assign string values to a string-variable (a sized reg-variable) and then display the variable with an ASCII radix. This was always awkward and more difficult in Verilog.

SystemVerilog has enumerated types similar to VHDL. Enumerals can be abstract symbolic values or, unlike VHDL, be assigned specified numerical values that will be used in their physical implementation or to automatically convert to and from other scalar values.

By default, SystemVerilog enumerated types are 2-state integers, but SystemVerilog allows an enumerated type to be declared as a 4-state type, which can be very useful for certain types of designs including FSM designs[4].

2.4 Records

SystemVerilog's *struct* is functionally equivalent to VHDL's *record*. Unique to SystemVerilog is the concept of a *packed struct*. A *packed struct* is automatically mapped into an equivalent scalar bit vector value. It allows either traditional field access into the structure or bit-select or a part select of the structure as if it had been declared as a single scalar variable. It also allows automatic conversion when assigning to and from scalar variables or other *packed* structures.

2.5 Multidimensional Arrays

Multidimensional arrays were added to Verilog-2001.

SystemVerilog also includes the VHDL-like array attribute functions *$left*, *$right*, *$low*, *$high*, *$increment*, *$length* and *$dimensions*. SystemVerilog also adds the concept of multiple packed dimensions for easier manipulation of multiple slices of an array or the entire array itself in a single statement.

2.6 Arrays of Arbitrary Type

VHDL has allowed arrays of arbitrary types, but Verilog did not permit arrays of non-integer types, such as real data types. SystemVerilog permits declarations of arrays of arbitrary types including *real*, object handles and *events*.

2.7 Named Array Subtypes

VHDL has the ability to pre-define "standard" array configurations by creating a subtype for them. The type name can then be reused instead of always requiring the specification of an array type for every variable.

```
subtype A_BYTE is STD_LOGIC_VECTOR(7 downto 0);

variable WORD: STD_LOGIC_VECTOR(15 downto 0);
variable BYTE: A_BYTE;

BYTE := WORD(15 downto 8);
```

The *typedef* statement in SystemVerilog can be used to accomplish the same thing.

```
typedef bit [7:0] byte_t;
bit [15:0] word;
byte_t     a_byte;

initial begin
  a_byte = word[15:8];
  ...
end
```

2.8 Unconstrained Arrays

SystemVerilog includes one-dimensional dynamic arrays whose size can be changed at runtime using the built-in functions new[] and delete(), and whose size can be queried using the built-in function *size()*.

2.9 Unresolved Signals

VHDL has a *std_ulogic* type, an unresolved type that reports an error if more than one driver is attached to the same signal. Verilog only had resolved types in the form of nets, most notably the *wire*, *wand* and *wor* types.

SystemVerilog has new unresolved data types *logic* and *bit*. SystemVerilog permits either a single-driver assignment to any variable or one or more procedural assignments to the same variable. For designs that are built using only unresolved signals, the *logic* or *bit* types can be used for all continuously driven or procedurally assigned signals and variables.

The 4-state *logic* and the 2-state *bit* data types probably would be better understood if they had been named *ulogic* and *ubit*. The *logic* data type is roughly equivalent to the VHDL *std_ulogic* type. Unlike the Verilog *wire* that permits multiple resolved drivers, both *logic* and the 2-state *bit*

SystemVerilog types only allow one driving source. The *wire* type can still be used for automatic resolution of drivers.

The one potential confusion surrounding the *logic* and *bit* keywords is that they represent unresolved logic types in Verilog while *std_logic* and *bit* represent resolved types in VHDL.

I personally would like to see the *logic* and *bit* SystemVerilog keywords changed to *ulogic* and *ubit*, to more closely reflect the VHDL-like behavior of these data types. Unfortunately the Accellera committee, made up mostly of EDA vendors, would prefer to make no changes. If you support changing these keywords to *ulogic* and *ubit*, please send email to the SystemVerilog - Basic Committee, the Accellera subcommittee responsible for this feature, and request that *logic* and *bit* be changed to the more VHDL-like *ulogic* and *ubit*. Requests should be sent to the following email address: sv-bc@eda.org

The Accellera committees pay attention to user requests, but unless users care enough to request this change, *logic* and *bit* will most likely continue to be the new unresolved data type keywords in SystemVerilog.

2.10 Separate Entity & Architecture

VHDL separates the entity (port list declarations) from the architecture (internal functionality of the design). Verilog does not separate the port list from the body of the module.

SystemVerilog has a VHDL-like entity-architecture separation through the use of the *interface* construct. The SystemVerilog *interface* capability goes much further than simply separating the port list declaration from the module (see section 5.6).

2.11 Iterated Instantiations

Verilog-2001 already has a generate for-loop, which is also supported by SystemVerilog.

It should be noted that when instantiating a contiguous range of instances, such as 16 data pads or 32 address pads at the top-level ASIC module, the Verilog-1995 Array of Instance (AOI) is a much more concise and better supported Verilog construct.

Guideline: think Array of Instance first, think Verilog generate statement second.

2.12 Conditional Instantiation

VHDL has an if-generate statement for conditional instantiation. SystemVerilog supports the conditional generation statements added in Verilog-2001: the *generate if-else* statement and the *generate case* statement.

2.13 Configurations

Verilog-2001 added a powerful configuration capability that includes inheritance.

2.14 “others =>” in Literals

VHDL had a fill operation with the use of the *others* => specification. SystemVerilog enhanced the fill operation capabilities of Verilog with the fill operations: '0, '1, 'z and 'x.

Prior to the introduction of the fill operators, Verilog engineers had to explicitly assign a pattern of all 1's (example: reg [63:0] ones = 64'hFFFF_FFFF_FFFF_FFFF;). This was inconvenient for a large number of bits or when the number of bits was unknown.

Unlike VHDL, the fill operators cannot be used with more complex types. They can only be used to fill bits in a scalar variables.

3.0 Notable VHDL Constructs Missing from SystemVerilog

The following features present in VHDL do not (currently) have an equivalent in SystemVerilog. In some cases, the feature was not often used in VHDL models. In other cases, a simple work-around exists. In others, one can hope that they will be included in future versions of the SystemVerilog language.

3.1 Scalar subtypes

VHDL has the ability to define a new scalar subtype that restricts the range of values that can be assigned to a variable of that type. The subtype remains compatible with the base type, so conversion functions are not necessary. Run-time errors are issued if an out-of-range value is assigned.

```
subtype BYTE is INTEGER range 0 to 255;
variable MY_BYTE: BYTE;

MY_BYTE := (...) mod 256;
```

The most common use of scalar subtypes was to help the synthesis tool use the minimum number of bits to implement a register or signal. Declaring a *bit*, *logic* or *reg* variable with the appropriate number of bits achieves the same goal.

```
reg [7:0] MY_BYTE;

MY_BYTE = ...;
```

However, there is no out-of-range value checking on assignment (although linting and formal tools can detect those). Note that for performance reasons many VHDL simulators offered the capability to disable run-time out-of-bound checking for subtypes. Furthermore, it is not possible to define scalar subtypes that do not correspond to whole number of bits, such as the pre-defined *positive* or *natural* integer subtypes.

3.2 New scalar types

VHDL can define new scalar types that are incompatible with the original base type. This can be used to define values that are interpreted differently, such as a scaled fixed-point value. The strong typing can ensure that scalar values with different interpretations are not accidentally mingled.

```
type FIXPNT is INTEGER range 0 to 255;
variable SAMPLE: FIXPNT;

SAMPLE := FIXPNT(0.5 * 256);
```

In my experience, this is a rarely-used feature of VHDL. The same can be accomplished by using one of the strongly-typed types, such as a *struct* or a *class*, to encapsulate the scalar value.

3.3 Enumeral Overloading

VHDL supports the reuse of the same symbols in different enumerated types. For example, the enumeral '1' is used in the predefined *CHARACTER*, *BIT* and *STD_ULOGIC* types, each with a different semantic. If the type of the overloaded enumeral cannot be determine by context, a qualifying expression is necessary to indicate which particular enumeral is being specified.

SystemVerilog enumerated types cannot share the same enumeral symbol if they are in the same scope level. Because they are in different scope, class-level enumerated types can reuse an enumeral symbol used in a different class.

3.4 Subprogram Overloading

In VHDL, different subprograms can be defined with the same name. As long as their signature (i.e. the type or number of their argument and return value) is unique, the compiler will be able to determine which subprogram is called based on the context.

This useful feature is not readily available in SystemVerilog. Non-virtual methods in *classes* can be overloaded with a different signature but they still hide the original declaration in the base class; they do not co-exist at the same scope level.

3.5 Operator Overloading

A consequence of the lack of subprogram overloading is the absence of operator overloading in SystemVerilog. Although useful for creating concise descriptions. they often create confusion if used outside of an arithmetic context.

When operator overloading is used to create a completely new algebra, it is very difficult for someone familiar with the language but unfamiliar with the functionality of the overloaded operators to understand a model. For example, WAVES is a standard for modeling manufacturing test vectors using pre-defined VHDL types and overloaded operators. Anyone not

intimately familiar with the WAVES semantics of common VHDL operators (such as “+”) will be unable to understand even the simplest model, despite being written in a familiar language.

Although operator overloading provides a convenient way to define concise operators to execute user-defined operations, the same functionality can be captured in a Verilog function with no misunderstanding or confusion with the pre-defined semantics of the language. It is my opinion and the opinion of VHDL expert Janick Bergeron[12] that operator overloading is a worm better left in the can.

3.6 Packages

The concept of *packages* as containers for shared declarations does not exist in SystemVerilog. Packages can be emulated by locating variable declarations and subprograms in a port-less module. The module need not be instantiated (if the variables need to be globally unique) and the variables and subprograms it contains can be accessed using a hierarchical reference.

```
module pkg;
   task t;
      ...
   end_task;
endmodule

module model(...);
   initial begin
      ...
      pkg.t(...);
      ...
   end
endmodule;
```

Furthermore, with the availability of an object-oriented programming model, the *package* encapsulation mechanism can be implemented using *classes* to encapsulate types, data and methods. If a single global instance of the data members are required, similar to shared variables or global signals in a VHDL package, *static* data members can be used as well as a singleton pattern[8].

4.0 Features requested for VHDL-200x

There are features that have been requested by VHDL engineers that are readily available in SystemVerilog. Most of these features have reasonable work-arounds in VHDL through the use of a package or more verbose coding tricks, but a concise implementation of these features or a higher-level of abstraction is often the desired solution. Otherwise, I would all still be coding using assembler.

4.1 `ifdef conditional compilation

Conditional compilation is a simple and frequently used feature of Verilog or C to select between different design implementations, different testbench options and to enable selective debug monitors and code.

The VHDL “if-generate” statement does offer limited conditional compilation capabilities but is not as powerful as the *`ifdef* ability to conditionally compile any block of HDL code.

4.2 fork-join

The fork-join statement allows for the spawning of multiple processes and optionally waiting for all of the processes to complete before continuing execution of other processes and code. VHDL cannot dynamically spawn multiple processes.

SystemVerilog adds even more capability to the original Verilog *fork-join* statement by adding two new join options: join_any and join_none.

The fork-join_any combination spawns multiple processes but only waits for the first process to complete before continuing execution of following sequential code. The fork-join_none combination spawns multiple processes but does not wait for any of the processes to complete before continuing execution of the following sequential code.

4.3 Hierarchical Referencing

Verilog hierarchical referencing (also referred to as cross-module-referencing or XMR or CMR), is a feature that is extensively used in Verilog testbenches. This feature allows simple probing into or monitoring of buried signals without requiring that the signals be routed to the top of design for observation. No declaration of global signals in a package is required to take advantage of this feature nor any modification of the original monitored code is required.

4.4 Bitwise Reduction Operators

Verilog has the concise C-like unary reduction operators built into the language. VHDL engineers use a package like the Synopsys-copyrighted *std_logic_misc* to introduce the equivalent functionality via the pre-defined functions AND_REDUCE, NAND_REDUCE, OR_REDUCE, NOR_REDUCE, XOR_REDUCE and XNOR_REDUCE.

4.5 Left-Hand Concatenation

The ability to do left-hand-side (LHS) concatenation is a feature that facilitates the modeling of several functions. A classic example of opportune LHS concatenation is in the RTL modeling of a barrel shifter. After making the appropriate declarations, a very simulation and synthesis efficient parameterized barrel shifter model can be coded using one line of code, as shown in Example 1.

```
module barrel_shifter
  #(parameter shiftSIZE = 8, cntSIZE = 3)
  (output [(shiftSIZE-1):0] y,
   input  [(  cntSIZE-1):0] rotate_cnt,
   input  [(shiftSIZE-1):0] a);

   wire   [(shiftSIZE-1):0] tmp; // discarded unused-bits

  // The shifted MSBs will be truncated upon assignment to y
  assign {y,tmp} = {a,a} << rotate_cnt;
endmodule
```

Example 1 - Verilog-2001 barrel shifter model

In addition to the above barrel shifter model, Verilog arithmetic operations are often efficiently coded using LHS concatenation.

4.6 Multiple Concatenation or Replication {{}}

Another operator that would be a nice addition to VHDL is the Verilog replication operator. Verilog has the ability to replicate the contents of a bit or range of multiple bits using the replication operator. The operation `{8{inv}}` is equivalent to the multiple concatenation of `{inv,inv,inv,inv,inv,inv,inv,inv};` The `inv` identifier could be a single bit replicated into an equivalent 8-bit vector, or it could be multiple bits, such as 4-bit vector replicated into a 32-bit vector

A simple bus inversion can be implemented with either of the following assignments:

```
assign busout = inv ? ~busin : busin; // using ternary if-else operator
assign busout = {8{inv}} ^ busin;     // using replication operator
```

The `{{}}` operator is often used to do manual sign extension for an RTL model.

```
assign value16[15:0] = { {8{value[7]}, value[7:0] };
```

4.7 Repeat loops

Iterating for a constant number of iterations is a common operation. Although it can be easily modeled using a *for-loop* construct, the intent is more easily conveyed (and the function correctly coded) using a *repeat* loop.

4.8 Object Oriented Programming

SystemVerilog's *class* provides an object-oriented programming model. It supports virtual methods and classes, single inheritance, data and method overloading, static data members and constructors.

4.9 Constraint Solving

The apparition of Hardware Verification Languages such as *e*, Vera and the SCV library demonstrated the need for the ability to randomly generate coherent and interesting input stimuli. "Randomly generate" is the easy part, which can be done using $random in Verilog or by using one of the pre-defined random generation packages in VHDL. "Coherent and interesting" is the hard part. It requires that the randomly generated values be subjected to constraints to ensure that the set of random values creates valid stimulus. It also requires the ability to cross-constrain multiple instances of those same variable sets to create interesting scenarios. Furthermore, it is also necessary to be able to modify or add to those constraints to create corner cases or inject errors[12].

SystemVerilog has a powerful constraint specification and control mechanism. Built on top of the object-oriented framework, constraints are declarative and can be overloaded in user or test-specific class extensions. Constraint blocks can also be defined out-of-file or turned off. All of these offer much better control than the simple semantic of soft constraints and are more flexible because no constraint is ever truly hard. Virtual methods can also be overloaded to insert directed procedural data generation statements before or after the randomization process.

The constraint solving mechanism or technology is not specified in the language. It is left to the implementers of the language. But directives, such as *solve before*, can be used to help the solver provide better distribution of solutions for a given variable.

4.10 Standard C Interface

For as long as there have been models, engineers have wanted the ability to compile their C-algorithms directly with their HDL simulations.

VCS has had the ability to compile C-code with Verilog for years, but users have been reluctant to use the feature, fearing that any code written to take advantage of this feature would not be portable to other simulators, if needed.

SystemVerilog has standardized the C-interface to the SystemVerilog language making it possible for Verilog code to call C-functions and C-code to call SystemVerilog functions. With the C-interface standardized to SystemVerilog, users can now compile their C-code and SystemVerilog code together to take advantage of the high-level algorithmic and architectural capabilities of C programs and the efficient HDL styles of a Verilog-VHDL-like language.

5.0 Additional SystemVerilog Features Not Found in VHDL

5.1 Logic-Specific Processes

VHDL has the *process* statement, Verilog has the equivalent *always* block, and SystemVerilog adds a few nice logic-specific variations of the *always* block that will permit better linting and checking of desired functionality by simulators, synthesis tools and formal tools.

The *always_comb* block conveys the designer's intent to model combinational logic without the need to expand the required combinational sensitivity list. In the following example a simulation or synthesis tool may warn: “Combinational logic requested but latch was inferred.”

```
module ao1b (
  output bit_t q,
  input  bit_t en, d);

  always_comb
    if (en) q <= d;
endmodule
```

Example 2 - Erroneous combinational logic example modeled using always_comb and detected by the SystemVerilog compiler

The *always_latch* block conveys the designer's intent to model latched logic, again without the need to expand the required latch sensitivity list. In the following example a simulation or synthesis tool may warn: “Latch requested but combinational logic feedback loop inferred.”

```
module lat1b (
  output bit_t q,
  input  bit_t en, d);

  always_latch
    if (en) q <= d;
    else    q <= q;
endmodule
```

Example 3 - Erroneous latch-logic example modeled using always_latch and detected by the SystemVerilog compiler

The *always_ff* block conveys the designer's intent to model registered logic, but a sensitivity list is required to show if sets and resets are synchronous or asynchronous and to also name the clock and clock polarity. In the following example a simulation or synthesis tool may warn: “Incorrect sensitivity list, flip-flop not inferred.”

```
module dff1b (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always_ff @(clk, rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 4 - Erroneous sequential logic example modeled using always_ff and detected by the SystemVerilog compiler

One potentially exciting enhancement related to the *always_ff* block is the future possibility to do RTL modeling of Dual-Data Rate (DDR) flip-flops. Consider the following example, which would be illegal with current synthesis tools (because the clk signal does not have a required *posedge* or *negedge* qualifier):

```
module ddrff (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always_ff @(clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 5 - DDR flip-flop modeled using the new SystemVerilog always_ff construct

Future synthesis tools could read this example, determine the design intent (*always_ff* == registered logic), recognize that the clk signal is not part of any "if" or "if-else" tests and must therefore be a clock that triggers on both edges of the clock and the resulting logic should therefore be a DDR flip-flop. Naturally, the synthesis tool would only permit one signal in an *always_ff* block to be listed without a corresponding *posedge* or *negedge* qualifier. If this were a typical always block, the synthesis tool would not know if combinational or sequential logic was intended and would simply report an error.

The *always_comb*, *always_latch* and *always_ff* logic-specific processes are small but very useful enhancements for RTL coders.

5.2 Implicit Port Connections

Verilog and VHDL have both had the ability to instantiate modules using either positional or named port connections. Positional ports are subject to mis-ordered incorrect connections, which is why most experienced companies have internal guidelines requiring the use of named port connections. Unfortunately the use of named port connections in a top-level ASIC or FPGA design is typically a very verbose and redundant set of connections that requires multiple pages of coding to describe. Often, most of the top-level module port names match the equivalent net or bus connections.

Whenever a design review is conducted using a verbose top-level model, the reviewing engineers always ask the same question, "did you simulate it?" The instantiations are so tedious and verbose that nobody intends to read and verify every connection in the HDL design.

SystemVerilog addresses the top-level verbosity issue with two new implicit port connection enhancements that have no equal in VHDL: .*name* and .* connection.

5.2.1 The .*name* implicit port connection enhancement

Whenever the port name and size matches the connecting net or bus name and size, the port name can be listed just once with a leading period as shown below.

```
cpu u1 (.data(data), .addr(addr), .dval(dval),
        .aval(aval), .clk(clk), .rst_n(rst_n)); // Verilog-2001 style

cpu u1 (.data, .addr, .dval, .aval, .clk, .rst_n); // SystemVerilog style
```

5.2.2 The .* implicit port connection enhancement

Just like the *.name* implicit port connection enhancement, whenever port names and sizes match the connecting net or bus names and sizes, all like-named ports can be replaced with .* as shown below.

```
cpu u1 (.data(data), .addr(addr), .dval(dval),
        .aval(aval), .clk(clk), .rst_n(rst_n)); // Verilog-2001 style

cpu u1 (.*); // SystemVerilog .* style
```

5.2.3 Important implicit port connection rules

There are six important rules related to the implicit port connection enhancements. They are:

(1) *.name* and .* implicit ports are not allowed to be mixed in the same instantiation. Instantiating one module with *.name* implicit ports and another module with .* implicit ports is permitted.
(2) *.name* or .* implicit ports are not allowed to be mixed in the same instantiation with positional port connections.
(3) A named port connection is required if the port size does not match the size of the connecting net or bus. For example: a 16-bit **`data`** bus connected to an 8-bit **`data`** port requires a named port connection to show which of the 16 bits are connected to the 8-bit **`data`** port.
(4) A named port connection is required if the port name does not match the connecting net or bus name. For example the 32-bit pad address named **`paddr`** connecting to a 32-bit **`addr`** port would require a named port connections **`(`**... **`.addr(paddr),`** ...**`);`**
(5) A named port connection is required if the port is unconnected. For example. if the above instantiations have an unconnected bus error (**`berr`**) port, the unconnected port must be listed as a named empty port **`(`**... **`.berr(),`** ...**`);`**
(6) All nets or variables connected to the implicit ports must be declared in the instantiating module, either as explicit net or variable declarations or as explicit port declarations.

Rule #6 requires that 1-bit nets be declared if the net is to be implicitly connected to a port of the instantiated module. Similarly, multi-bit buses must still be declared. Implicit port connection does not support automatic 1-bit net declaration.

My experience so far has shown that typically only a few dozen additional wire declarations are required to take advantage of the *.name* and .* implicit port connection enhancements. The .* enhancement can reduce 10 pages of top-level ASIC or FPGA instantiation code down to three pages of equivalent code while highlighting the differences in the port connections.

5.2.4 Stronger port connection-typing

An interesting side-effect of the implicit port connection enhancements is that not only are the coding styles more concise and less error prone, but the coding style actually imposes some VHDL-like stronger typing on the port connections that did not previously exist in Verilog.

Verilog allows connections of unequal sizes and then issues a port-size mismatch warning when the design is elaborated. The *.name* and .* implicit instantiation enhancements require that all sizes be matched; hence, reducing port-size instantiation errors.

Verilog allows unconnected ports to be omitted from the instantiation port list. The *.name* and .* implicit instantiation enhancements require that all unconnected ports be listed; hence, reducing instantiation errors related to accidental omission of ports.

Verilog does not require declaration of 1-bit nets and declaring 1-bit nets does not increase the name checking of 1-bit nets. The *.name* and .* implicit instantiation enhancements require that connections be made to declared nets and variables in the instantiating module. This means that declarations will be required and tested in the instantiating module without the onerous use of the Verilog-2001 `default_nettype none directive (which also requires the keyword wire to be added to all net-ports).

The SystemVerilog designer will now get stronger size and declaration checking with an enhancement that reduces top-level RTL coding by as much as 70%. A very nice trade-off!

The .* implicit port instantiation enhancement not only offers better port checking, it also makes the code more concise and highlights net-connections that are exceptions to like-named connections. Reviewers will more easily focus on the important parts of an upper-level netlist as opposed to pages of redundant and error-prone verbose connections.

5.2.5 Potential implicit port connection problems

There is a new type of potential error associated with implicit port connections: what if a port name accidentally and unintentionally matches a net name in the instantiating module? The .* implicit connection enhancement will erroneously connect the same-named port and net together and it will have to be debugged (a bug which may not be easy to find). This problem is similar to the potential misconnection caused by scripts that automatically generate named port lists. In both cases, the wrong port may be connected to a same-name net

This enhancement was actually added at the request of Intel engineers that had a very similar capability with Intel's internal IHDL language. The SystemVerilog committee took the opportunity to ask Intel engineers if they had encountered significant difficulties debugging the problem described above. Intel engineers responded that they had seen the above problems but that they were rare and relatively easy to find and correct.

5.2.6 Use it right! Don't blame the EDA tools!

Editorial note from Cliff Cummings - Some EDA tool developers are worried about this enhancement because they are concerned that engineers will use it wrong, blame the EDA tools when errors are reported and tie up EDA support resources to debug engineering mistakes. This is a valid concern - stupid engineers doing stupid things and blaming the EDA tools.

To all engineers that intend to use this extremely powerful enhancement - when EDA tools report compile errors, please examine your code carefully before pointing the finger at the EDA vendor. I do not want to give EDA tool development engineers reason to reject future powerful enhancements due to a few stupid engineers!

I believe that the stronger port-typing will actually remove more support problems than will be introduced by stupid engineers using the .* enhancement in a stupid way.

5.3 The SystemVerilog 3.1 Event Scheduler

SystemVerilog extends the Verilog event scheduler and adds event regions targeted at enhancing verification and further standardizing PLI access into the event queue. These regions enable functionality similar to that provided by VHDL *postponed* processes. Existing recommendations and guidelines for RTL coding to avoid Verilog race conditions will remain the same when coding with SystemVerilog[3][5][11].

The content and complexity of the new event scheduler is beyond the scope of this paper, but it should be noted that the new event scheduler is 100% backward compatible with the IEEE Verilog event queues and offers new event regions that will further reduce any potential race conditions that could have occurred between assertions, testbench code and RTL models without any degradation of simulation performance.

5.4 Unions

SystemVerilog has the concept of a *union*. Like C's *union*, they provide a mechanism to store and manipulate different data representations in a single storage area. *Unions* can be *packed* to enable automatic conversion to and from scalar variables or access the content of the *union* using bit slicing[1].

5.5 Sparse Arrays

SystemVerilog has associative arrays that are ideal for modeling sparse arrays, such as very large memory spaces. This capability previous was handled in some Verilog simulators using a pragma or by using the DAMEM PLI code to model memories as dynamics sparse arrays[2]. Sparse arrays had to be similarly modeled using linked lists of records in VHDL.

5.6 Interfaces

SystemVerilog introduces the interface, a construct that, at it's lowest level, is a record-like bundle of signals encapsulated in a common interface block. It can be extended in advanced interfaces to include legal testing tasks, and powerful interface assertion checking.

One significant advantage related to the use of an interface is that now, “somebody owns the bus!”

Using interfaces means that instead of multiple engineers defining port connections at opposite ends of a bus, one engineer will encapsulate the bus connectivity and functionality into an interface and allow other team members to connect the interface to their piece of the design.

The concept and use of an interface by designers will take time to absorb and fully exploit.

5.6.1 Interface Data Types

Since an interface generally describes a variable that is driven from one module and read by another, in general, interface signals should be declared using variable types and not nets. Declaring wires in an interface is almost always an error. The exceptions are bi-directional buses or multiply driven signals as used in specific logic designs like one-hot multiplexers or bus crossbars.

5.6.2 Modports

Once a bundle of signals has been added to an interface, the keyword *modport* can be used to add port direction information to those signals that will be connected to *modport*-specified modules. The *modport* defines the direction of the signals with respect to the module that instantiates the interface into the module port header.

Typically, there will be two (or more) *modports* in each interface. In the case of standard bus interfaces, there will typically be one *modport* to describe the initiator device and another *modport* to describe the target device. Both use the same signals but the direction of the signals are typically reversed between the two devices.

Standard interfaces will frequently have sender-*modport* signals and receiver-*modport* signals. There may also be unusual bus interfaces that describe a three-way connection using three different subsets of a collection of signals, each with a unique *modport* description. It probably will be rare to build an interface with only one *modport*.

In the absence of *modport* declarations, net types default to *inout* ports and variables default to a rather unusual new SystemVerilog port type called a *ref*-port. *Ref* ports are not described in this paper but their behavior is somewhat non-intuitive, which is why I give the following guideline:

Guideline: Declare and use modports for all interface signals.

Declaring *modports* will assign commonly understood port directions to all signals in an interface.

5.6.3 Generic Interfaces

SystemVerilog also introduces the concept of a generic interface. Instead of instantiating a specific interface, a module header can generically instantiate an interface using the keyword *interface*, followed by an instance name. When the module is instantiated within another module,

the interface that is connected to the instantiated module becomes the inherited interface within the lower-level module.

Using generic interfaces requires careful planning to make sure that all interface port signals and task names that are referenced from the module, exist in the interface that will be generically connected.

5.6.4 Interface Testing Tasks and Functions

Adding low-level, pre-tested, read and write tasks and other useful verification tasks and functions to an interface can make block-level verification and interface-connected designs easier to test. Instead of writing the read and write routines within each block-level test, the block test can be allowed to call the read and write tasks that are pre-coded and known to work with an interface.

5.6.5 Interface Assertions

One very nice feature of interfaces is that assertions can be added to the interface. This means that any design engineer or IP developer can add signal dependency and sequencing checks to an interface to report a violation when the interface is being used incorrectly.

This can help an IP user or a verification engineer to quickly spot problems in how IP is being used or in how a design is being tested. This reduces frustration on the part of the user or verification engineer while simultaneously reducing the number of support calls or interruptions related to simple and easily correctable mistakes.

5.7 Vera-Like Testbench Features

SystemVerilog adds Vera-like capabilities with a Verilog-like syntax. Powerful testbench features include program blocks, clocking domains for well-timed signal sampling and stimulus driving, semaphores, mailboxes, event sequencing and more.

5.8 Assertion Capabilities

SystemVerilog adds PSL-compatible assertion capabilities with a Verilog-like syntax. Assertion capabilities include immediate and concurrent assertions, sequences, properties, multi-clock support and binding properties to scopes or instances.

Useful assertion severity system tasks, assertion control system tasks and built-in assertion functions are all part of SystemVerilog.

Accellera has announced its intention to merge SystemVerilog assertion and PSL assertion capabilities and syntax in the next year.

6.0 EDA Tool Support

One argument often posted against SystemVerilog is that it took years for vendors to support VHDL-1993 and no vendor currently supports all of the Verilog-2001 features (by the time this paper is published, one or more vendors may have full Verilog-2001 support). SystemVerilog includes multiple complex new enhancements. Certainly it will take years for vendors to fully support these features!

I believe this argument is flawed. Both VHDL-1993 and Verilog-2001 included new features that had never been tried within their respective languages. The Standards bodies for both languages proposed enhancements that they wanted without knowing exactly how the features would be implemented. Similarly, vendors experienced some difficulty implementing new features with no reference model.

The complex SystemVerilog enhancements were based on donations of working and successful technologies. SystemVerilog 3.0 was largely based on the donation of the Superlog syntax and SystemVerilog 3.1 was largely based on multiple donations from multiple sources, including the Open Vera Assertions (OVA), Open Verification Library (OVL) assertions, additional Superlog features and even Intel HDL (IHDL) syntax and capabilities. The fact that so many SystemVerilog proposed features have already been implemented by one tool or another, has contributed to unprecedented implementation of new features into a standard language product.

I fully expect vendors to implement SystemVerilog enhancements in record time and both myself and noted VHDL expert, Janick Bergeron[12], are looking forward to using the new features on near-future design and verification projects.

6.1 Linting Tools

SystemVerilog's scalar types are still not strongly typed or strongly sized. For many design teams, a good linting tool and accompanying methodology will continue to make sense when using SystemVerilog.

6.2 Legacy VHDL Code

I hope I have been successful at showing VHDL users that SystemVerilog possesses all of the necessary features and capabilities to write models and testbenches without significant changes in methodology or approaches. By following a few well-known modeling guidelines, it is possible to write SystemVerilog models that are as reliable and as high-level as any VHDL model. Companies and individuals who decide to adopt SystemVerilog for new projects should have little difficulty in making the transition.

It is clear that the major difficulty will be in dealing with legacy VHDL models. SystemVerilog's backward compatibility with Verilog does not present similar difficulties for legacy Verilog designs. To that effect, two solutions are immediately apparent: translation and co-simulation.

The VHDL-inspired features that are available in SystemVerilog will make translation of VHDL models and verification suites into SystemVerilog easier to implement than past efforts to translate VHDL into Verilog. However, the effort required to translate and debug working VHDL designs may not be worth the investment. Given the current challenges in translating RTL models – which have a simple and well-defined semantics in both language, it is unlikely that an automated process will reliably translate arbitrary VHDL code, even with the availability of similar language constructs.

It is more likely that users will require (and tool vendors will provide) SystemVerilog-VHDL co-simulation environments to support simulation of new SystemVerilog code with existing VHDL designs and testbenches. It will be important for such co-simulation environments to be able to support VHDL designs embedded in SystemVerilog designs or testbenches, as well as SystemVerilog designs embedded in VHDL designs or testbenches.

7.0 Conclusions

Engineers with VHDL experience may have last compared VHDL to Verilog-1995[9] syntax and capabilities, unaware that Verilog-2001 added many VHDL-like features to the Verilog HDL. VHDL engineers that want to run an updated comparison would do well to compare VHDL to SystemVerilog 3.1. Not all SystemVerilog 3.1 features have been fully implemented yet, but should be mostly available in by Q1 2004 from Synopsys[14].

VHDL is not dead. For better or for worse, U.S. military contracts still require contractors to submit RTL-based designs using VHDL. But even better, the best features from VHDL have been combined with the best features and syntax of Verilog to form a near-superset of both languages. Engineers with a Verilog background will unknowingly benefit from VHDL features that have been added to SystemVerilog and engineers with a VHDL background may do well to call this new SystemVerilog language, SystemVHDL!

In addition to the best-of-class features from both Verilog and VHDL, SystemVerilog includes powerful enhancements for even more powerful RTL coding, improved assertion based dynamic and formal verification and an easy interface to C-coded architectural, algorithmic and verification algorithms.

8.0 Acknowledgements

My sincere thanks to Paul Stein and Brian Kane for reviewing, correcting and offering recommendations to improve the quality of this paper.

A special thanks to respected Verilog, VHDL and verification expert, Janick Bergeron, who contributed significant information and answered important VHDL and Object Oriented questions to help me compile this paper.

References

[1] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language, Second Edition," Prentice Hall, Upper Saddle River, NJ, 1988.

[2] Clifford E. Cummings, "Efficient Verilog Memory Modeling Using DAMEM," International Cadence Users Group Conference 1995.

[3] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," SNUG (Synopsys Users Group) 2000 User Papers, section-MC1 (1st paper), March 2000. Also available at www.sunburst-design.com/papers

[4] Clifford E. Cummings, "Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements," SNUG (Synopsys Users Group San Jose, CA 2003) Proceedings, March 2003. Also available at www.sunburst-design.com/papers

[5] Clifford E. Cummings, "Verilog Nonblocking Assignments with Delays, Myths & Mysteries," SNUG (Synopsys Users Group Boston, MA 2002) Proceedings, September 2002. Also available at www.sunburst-design.com/papers

[6] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG (Synopsys Users Group) 1999 Proceedings*, section-TA2 (2nd paper), March 1999. Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[7] Douglas J. Smith, "HDL Chip Design," Doone Publications, Madison, AL., January 1997.

[8] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 0-20163361-2

[9] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-1995

[10] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001

[11] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, 2nd edition, Kluwer Academic Publishers, 2003

[12] Janick Bergeron, personal communication

[13] John Cooley, "VHDL, The New Latin," EE Design, April 7, 2003, www.eedesign.com/columns/industry_gadfly/OEG20030407S0056 & www.deepchip.com/gadfly/gad040703.html

[14] Michael Santarini, "DVCon: SystemVerilog key to new design paradigm," EE Design, February 24, 2003, www.eedesign.com/printableArticle?doc_id=OEG20030224S0068

[15] Stephen Bailey, "Comparison of VHDL, Verilog and SystemVerilog," Available for download from www.model.com

[16] Stuart Sutherland, "Interfacing C-Language Models to Verilog Simulations Using the Verilog PLI," IHDL (International HDL Conference) 2000, Tutorial 7, March 2000.

[17] *SystemVerilog 3.0 Accellera's Extensions to Verilog*, Accellera, 2002, freely downloadable from: www.systemverilog.org

[18] *SystemVerilog 3.1 Accellera's Extensions to Verilog*, Accellera, 2003 freely downloadable from: www.systemverilog.org

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 21 years of ASIC, FPGA and system design experience and 11 years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, is the only Verilog and SystemVerilog trainer to co-develop and co-author the IEEE 1364-1995 & IEEE 1364-2001 Verilog Standards, the IEEE 1364.1-2002 Verilog RTL Synthesis Standard and the Accellera SystemVerilog 3.0 & 3.1 Standards.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers Verilog, Verilog Synthesis and SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of September 8th, 2003)