SystemVerilog 3.1
It's What The DAVEs In Your Company Asked For

Stuart Sutherland, Sutherland HDL, Inc., Portland, Oregon

ABSTRACT

***DAVE**. It's short for all the **D**esign **A**nd **V**erification **E**ngineers at you company. For many years, the behavioral coding features plus, a few extras in the Verilog HDL, satisfied the needs of both hardware design engineers and hardware verification engineers. Designs could be modeled and verified using a single language. The DAVEs in your company were happy.*

As design sizes increased, however, the amount of verification required increased dramatically. The 15+ year old Verilog HDL did not have the constructs needed to effectively verify these extremely large designs. So along came several proprietary Hardware Verification Languages (HVLs) to the rescue. These languages specialized in giving verification engineers powerful constructs to describe stimulus and checking routines in a much more concise manner. These proprietary languages solved a need, but your DAVEs were no longer happy. In fact, in many companies, DAVE had to be severed in two, with one half doing design in one language, and the other half doing verification in an entirely different language. DAVE's life became confusing, difficult to manage, and prone to errors.

*Enter the SystemVerilog 3.1 standard currently being defined by Accellera. It is exactly what all the DAVEs in your company have been asking for—and what they used to have. One language for both design and verification. The SystemVerilog standard comprises a large number of extensions to the Verilog-2001 HDL. Many of the extensions are targeted towards modeling large system-level designs. Other extensions provide the means to verify these very large designs. SystemVerilog is a true **HDVL** (**H**ardware **D**esign AND **V**erification **L**anguage. Your DAVEs are back together, working as one again.*

This paper presents an overview of the features currently planned for the SystemVerilog 3.1 standard, which is expected to be released in June 2003.

Introduction

SystemVerilog is a set of high-level extensions to the IEEE 1364 Verilog-2001[1,2] language. These extensions provide powerful enhancements to Verilog, such as C language data types, structures, packed and unpacked arrays, interfaces, assertions, and much more. SystemVerilog is being defined by Accellera, the combined VHDL International and Open Verilog International organizations. It is expected that the SystemVerilog extensions will become part of the next generation of the IEEE 1364 Verilog standard. The SystemVerilog standard is being released in multiple phases:

- The **SystemVerilog 3.0** standard, which was released in June 2002, added a large number of extensions to the Verilog-2001 HDL. These extensions primarily addressed the needs of hardware modeling for large, system-level designs and IP designs.
- **SystemVerilog 3.1** primarily targets the needs of verification engineers, with another large set of extensions to the Verilog-2001 HDL. This release is planned for June 2003.

SystemVerilog 3.0

SystemVerilog 3.0 began with donations of major portions of the SUPERLOG language by Co-design[3], and assertions work by Verplex (OVL) and Intel (ForSpec). These donations were made in 2001. The Accellera HDL+ subcommittee then standardized these donations as "*SystemVerilog 3.0*". SystemVerilog 3.0 primarily extends the hardware modeling capabilities of Verilog. These extensions bridge a communication gap between high-level system architects working in C, C++ or SystemC and the detailed RTL and gate level hardware design engineer. SystemVerilog enables these engineers to communicate in the same language, and easily share their work without the need for language translators.

The following list summarizes some of the more significant enhancements to Verilog provided by SystemVerilog 3.0:

- Interfaces between major blocks of a design
- Assertions
- Global declarations
- Time units and precision as part of the language
- New data types
  - C language `int` and `char` and `void` types
  - Two-state `bit` type
  - Universal `logic` type
- User-defined types, using the C `typedef`
- Enumerated types
- Structures and unions
- Packed and unpacked arrays
- Type casting
- Increment, decrement and assignment operators (`++`, `--`, `+=`, etc.)
- `do–while` bottom testing loop
- C `break`, `continue` and `return` jump statements
- Specialized procedures `always_comb`, `always_latch` and `always_ff`
- Dynamic processes

The focus of this paper is on the SystemVerilog 3.1 enhancements to the Verilog language. The enhancements in 3.0 are not covered in this paper. The author presented a paper describing SystemVerilog 3.0, titled ***"Verilog, The Next Generation: Accellera's SystemVerilog"***[4] at this conference last year, HDLCon 2002. This paper is available at *www.sutherland-hdl.com/papers/2002-HDLCon-paper_SystemVerilog.pdf*. The SystemVerilog 3.0 Language Reference Manual[3] is a publicly available document, which can be downloaded from the Accellera web site, *www.accellera.org*.

SystemVerilog 3.1

SystemVerilog 3.1 is to extend the verification capabilities of Verilog. These extensions are expected to ratified by Accellera in June, 2003. The current draft of the SystemVerilog 3.1 Language Reference Manual[6] is not publicly available at this time, as it is still being defined.

SystemVerilog is based on proven technologies. Instead of re-inventing the wheel, Accellera takes technology donations of propriety languages, and fine tunes the proprietary capability to become an open standard. SystemVerilog 3.0 was based on donations from Co-design, Verplex and Novas. SystemVerilog 3.1 is built on significant new donations from Synopsys and IBM. These donations include:

- The Synopsys VERA-Lite Hardware Verification Language
- The Synopsys VCS DirectC Application Programming Interface (API)
- The Synopsys Assertion Application Programming Interface (API)
- The IBM Property Specification Language (PSL) assertion technology (more commonly referred to as “Sugar”).

Following is summary of the primary features added in SystemVerilog 3.1.

- Classes
  - Object oriented programming, like C++
  - Encapsulation, inheritance, and polymorphism
  - Safe pointers (handles)
- New data types
  - Strings
  - Dynamic arrays
  - Associative arrays
  - Lists
- Pass by reference to tasks and functions
- Dynamic processes and synchronization
  - Enhanced fork—join, with all, any, none, priority exit
  - Semaphore and mailbox constructs
  - Enhanced event data type
- Special program blocks for test programs
- Clocking domains
  - Synchronous interfaces
  - Cycle-based functionality
  - Race-free test bench
- Random values
  - Repeatable random values
  - Constrained random values
- Extended assertions
  - Merge the features of OVA, ForSpec and Sugar into a single standard
  - Provide a common set of assertions for both simulation and formal verification
- Application Procedural Interfaces
  - A Direct Foreign Language Interface (based on DirectC)
  - An assertion API to provide external program interaction with assertions

Note that the information presented in this paper is subject to change. The SystemVerilog 3.1 standard is still being developed. No major changes from what is presented in this paper are anticipated, but the specific keywords and syntax of some constructs may be different in the final, ratified version of the standard. None-the-less, the constructs presented in this paper should serve to illustrate the powerful capabilities that SystemVerilog 3.1 will have to offer.

Overview of SystemVerilog 3.1 Features

Classes

SystemVerilog 3.1 adds Object Oriented classes to the Verilog languages, similar to C++. A class can contain data declarations, and task/function for operating on the data. Data declarations are referred to as "***properties***", and tasks/functions are referred to as "***methods***". The properties and methods together define the contents and capabilities of an "***object***".

Classes allow objects to be dynamically created, deleted and assigned values. Objects can be accessed via handles, which provides a safe form of pointers. Class objects can inherit properties and methods from other classes, as in C++. SystemVerilog automatically handles memory allocation, de-allocation and garbage collection. This prevents the possibility of memory leaks. An example of using classes with SystemVerilog is:

```
class Packet ;
  bit [3:0] command;           // data portion
  bit [40:0] address;
  bit [4:0] master_id;
  integer time_requested;
  integer time_issued;
  integer status;

  function new();              // initialization
    command = IDLE;
    address = 41'b0;
    master_id = 5'bx;
  endfunction

  task clean();
    command = 0; address = 0;
    master_id = 5'bx;
  endtask
                               // public access entry points
  task issue_request( int delay );
    // send request to bus
  endtask

  function integer current_status();
    current_status = status;
  endfunction
endclass
```

String data type

SystemVerilog 3.1 adds a **`string`** data type. This data type contains a variable length array of ASCII characters. Each time a value is assigned to the string, the length of the array is automatically

adjusted. This eliminates the need for the user to define the size of arrays, or to be concerned about strings being truncated due to an array of an incorrect size.

String operations can be performed using standard Verilog operators: `=`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `{,}`, `{{}}`. In addition, a number of string class methods are defined, using SystemVerilog's Object Oriented classes: `len()`, `putc()`, `getc()`, `toupper()`, `tolower()`, `compare()`, `icompare()`, `substr()`, `atoi()`, `atohex()`, `atooct()`, `atobin()`, `atoreal()`, `itoa()`, `hextoa()`, `octtoa()`, `bintoa()` and `realtoa()`.

Arrays

SystemVerilog 3.0 enhances Verilog arrays in several significant ways. SystemVerilog 3.1 adds two additional important enhancements: *dynamic arrays* and *associative arrays*.

Dynamic arrays are one-dimensional arrays where the size of the array can be changed dynamically. Objected Oriented class methods provide a means to set and change the size of dynamic arrays during run-time.

Associative arrays are one-dimensional arrays that can be indexed using any data type, including enumerated type names (static and dynamic arrays can only be indexed by integer expressions). Associative arrays are sparse arrays, meaning that storage for each element in the array is not actually allocated until the address is accessed. In addition to being able to use an index to access an array element, SystemVerilog 3.1 provides several class methods for working with associative arrays: `exists()`, `first()`, `last()`, `next()`, `prev()` and `delete()`.

Enumerations

SystemVerilog 3.0 adds enumerated types to the Verilog language. Enumerated types are strongly typed, and can only be assigned a value from its type set. Operations on enumerated types are limited. For example, it is illegal to increment an enumerated type, so a state machine model with an enumerated type called `State` cannot advance to the next state using `State++`. SystemVerilog 3.1 will extend enumerated types by providing a set of class methods to perform operations such as incrementing to the next value in the enumerated type set. In addition, SystemVerilog 3.1 adds a dynamic cast system function, called $cast. Dynamic casting can perform checking, and verify that, for example, `State = $cast(State + 1)` will result in a legal value in the `State` enumerated type set.

Task and Functions

SystemVerilog 3.0 added several enhancements to Verilog tasks and functions. SystemVerilog 3.1 adds additional enhancements. Verilog requires that a task or function call pass values in the order of the task/function argument declaration. SystemVerilog 3.1 allows values to be passed in any order, using the task/function argument names. The syntax is the same as named module port connections. For example:

```
task t1 (input int a, b, output int c);
   ...
endtask

check (.c(out), .a(in1), .b(in2));
```

With SystemVerilog 3.1, task and function input arguments can be assigned a default value as part of the declaration. This allows the task or function to be called without passing a value to each argument.

```
task t2 (input int a = 1, b = 1);

t2();          //a is 1, b is 1
t2( 3 );       //a is 3, b is 1
```

```
t2( , 4 );     //a is 1, b is 4
t2( 5, 8 );    //a is 5, b is 8
```

SystemVerilog allows task or function arguments to be passed by reference, instead of copying the value in or out of the task or function. Passing by reference allows the task or function to work directly with the value in the calling scope, instead of a local copy of the value. To use pass by reference, the task or function argument must be explicitly declared as a **`var`**, instead of an `input`, `output` or `inout`.

```
int my_array [1023:0]; //an array of integers

task t3 (var int arg1 [1023:0]); //arg1 is a reference to calling scope

t3 (my_array); // task will use my_array directly, instead of copying it
```

SystemVerilog also allows the return value of a function can be discarded by assigning the return to `void`. For example:

```
void = f1(...);
```

Enhanced for loops

Verilog `for` loops can have a single initial assignment statement, and a single step assignment statement. SystemVerilog 3.1 enhances `for` loops to allow multiple initial and step assignments.

```
for (int count=0, done=0, shortint i=1; i*count < 125; i++, count+=3)
   ...
```

Enhanced fork—join

In the Verilog `fork–join` statement block, each statement is a separate thread, that executes in parallel with other threads within the block. The block itself does not complete until every parallel thread has completed.

SystemVerilog 3.1 adds significant new capabilities to the Verilog `fork–join` statement block, using the modifiers `all`, `any` and `none`, where:

- **`all`** — statements which follow the `fork–join` are blocked from execution until *all* threads have completed execution. This is the default behavior of a `fork–join` statement block, if no modifier is specified.
- **`any`** — statements which follow the `fork–join` are blocked from execution until *any* thread has completed execution. That is, the join is blocked until the first thread completes.
- **`none`** — statements which following the `fork–join` are not blocked from execution while the parallel threads are executing. That is, each parallel thread is an independent, dynamic process.

SystemVerilog 3.1 also includes special system tasks to control the execution of parallel threads.

- **`$terminate`** causes all child processes that have been spawned by the calling process to immediately exit execution.
- **`$wait_child`** causes the calling process to block its execution flow until all child processes that have been spawned have completed.
- **`$suspend_thread`** causes the calling process to momentarily suspend execution. It is similar to a #0 delay, except that the process is guaranteed to suspend until all pending non-blocking assignments have been updated (#0 does not guarantee that behavior).

Inter-process synchronization

SystemVerilog 3.1 provides three powerful ways for synchronizing parallel activities within a design and/or a test bench: ***semaphores***, ***mailboxes***, and enhanced ***event*** types.

A *semaphore* is a built-in Object-Oriented class. Semaphores serve as a bucket with a fixed number of "*keys*". Processes using semaphores must procure one or more keys from the bucket before they can continue execution. When the process completes, it returns its keys to the bucket. If no keys are available, the process must wait until a sufficient number of keys have been returned to the bucket by other processes. The semaphore class provides several built-in methods for working with semaphores: `new()`, `get()`, `put()` and `try_get()`.

A *mailbox* is another built-in class. Mailboxes allow messages to be exchanged between processes. A message can be added to the mailbox at anytime by one process, and retrieved anytime later by another process. If there is no message in the mailbox when a process tries to retrieve one, the process can either suspend execution and wait for a message, or continue and check again at a later time. Mailboxes behave like FIFOs (First-In, First-Out). When the mailbox is created, in can be defined to have a bounded (limited) size, or an unbounded size. If a process tries to place a message into a bounded mailbox that is full, it will be suspended until there is enough room. The mailbox class also provides several built-in methods: `new()`, `put()`, `tryput()`, `get()`, `peek()`, `try_get()` and `try_peek()`.

SystemVerilog 3.1 adds several enhancements to the Verilog `event` data type. In Verilog, the `event` type is a momentary semaphore that has no logic value and no duration. The `event` type can be triggered, and other processes can be watching for the trigger. If a process is not watching at the exact instance the event is triggered, the event will not be detected. SystemVerilog enhances the `event` data type by allowing events to have persistence, so that the event can be checked after the event is triggered. Special system tasks are provided to work with persistent events: `$wait_all()`, `$wait_any()`, `$wait_order()` and `$wait_var`.

Random value constraints

The Verilog standard includes a very basic random number function, called `$random`. This function, however, gives very little control over the random sequence and no control over the range of the random numbers. SystemVerilog adds two random number classes, **`rand`** and **`randc`**, using SystemVerilog's Object Oriented class system. These classes provide methods to set seed values and to specify various constraints on the random values that are generated.

The following example creates a user-defined class called Bus, that can generate a random address and data value, with limits on the value sizes. A constraint on the address ensures that the lower two bits of random address value will always be zero. The class is then used to generate 50 random address/data value pairs. Using the `rand` and `randc` classes and methods that are provided, Much more elaborate random number generation is possible than what is shown in this simple example.

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;
   constraint word_align { addr[1:0] == 2'b0; }
endclass

//Generate 50 data and quad-aligned addresses, with data
Bus bus = new;
repeat(50)
   begin
      integer result = bus.randomize();
   end
```

Test bench program block

In Verilog the test bench for a design must be modelled using a Verilog module. Modules are used to model hardware at various levels of abstraction. They have no special semantics to indicate how test values should be applied to the hardware. SystemVerilog 3.1 adds a special type of module, declared between the keywords **`program`** and **`endprogram`**. The program block has special semantics and syntax restrictions to meet the needs of modeling a test bench. A program block:

- Has a single implicit initial block
- Executes simulation events in a special "verification phase", which is synchronized to hardware simulation events so as to prevent race conditions between the test bench and hardware models
- A special **`$exit`** system task that exits simulation after all program blocks are completed (unlike $finish, which exits simulation immediately, even if there are pending events)

A program block has ports, just like modules. One or more program blocks can be instantiated in a top-level netlist, and connected to the design under test.

Clocking domain

SystemVerilog adds a special clocking block, using the keywords **`clocking`** and **`endclocking`**. The clocking block identifies a "*clocking domain*", containing a clock signal and the timing and synchronization requirements of the blocks in which the clock are used. A test-bench may contain one or more clocking domains, each containing its own clock plus an arbitrary number of signals. Clocking domains allow the test bench to be defined using a cycle-based methodology of cycles and transactions, rather than the traditional event-based methodology of defining specific transitions times for each test signal.

Clocking domains greatly simplify defining a test bench that does not have race conditions with the design being tested. One common source of race conditions is when the test bench reads design values on the same clock edge in which the design may be changing values. Conversely, applying new stimulus on the same clock edge in which the design reads those values can result in race conditions. Using standard Verilog, the following example could result in unpredictable and erroneous simulation results.

```
initial //Race conditions with Verilog; no races with SystemVerilog
  begin
    @(posedge clk) input_vector = ...      //drive stimulus onto design
    @(posedge clk) $display(chip_output);  //sample result
                   input_vector = ...      //drive stimulus onto design
    @(posedge clk) $display(chip_output);  //sample result
                   input_vector = ...      //drive stimulus onto design
  end
```

To avoid this race in standard Verilog, it is necessary to calculate a time shortly before the clock edge in which to sample values. Since it is not possible to go backwards in time, it is not possible simply to do the sampling using, for example `@(posedge clock) - 2`. Instead, a delay must be added after each clock edge before driving a new test value. For example:

```
initial //Work around to avoid races with Verilog
  begin
    @(posedge clk) #1 input_vector = ...      //drive stimulus onto design
    #(clk_period - 2) $display(chip_output); //sample result
    @(posedge clk) #1 input_vector = ...      //drive stimulus onto design
    #(clk_period - 2) $display(chip_output); //sample result
    @(posedge clk) #1 input_vector = ...      //drive stimulus onto design
  end
```

A clocking domain can define detailed skew information for the relationship between a clock and one or more signals. *Input skews* specify the amount of time *before* a clock edge that input signals should be sampled by a test bench. *Output skews* specify how many time units *after* a clock edge that outputs should be driven. Note that "input" and "output" are relative to the test bench—that is, an output of the design under test is an input to the test bench. For example:

```
clocking bus @(posedge clk);
   default input #2ns output #1ns;   //default I/O skew
   input  enable, full;
   inout  data;
   output empty;
   output #6ns reset;                //reset skew is different than default
endclocking
```

Using clocking domain skews, the test bench itself can simply reference a clock edge to sample a value or drive stimulus, as is shown in the first example in this subsection. The appropriate skew will automatically be applied, thus avoiding race conditions with the design.

Assertion extensions

The PSL ("Sugar") assertion donation adds significant capabilities to the existing SystemVerilog 3.0 (OVA-based) assertions. PSL allows the same assertions to be used by a variety of EDA tools including simulation and formal verification. The Accellera SystemVerilog subcommittee for assertions has worked to closely integrate the features of Sugar, OVA and the existing SystemVerilog assertions. The result is a single assertion language, instead of several competing languages, that provides a convergence of all the best features of each of these assertion methodologies. And the best-of-the-best convergence is tightly integrated in the SystemVerilog language. Since SystemVerilog is an extension to Verilog-2001, all of this assertion capability is fully compatible with existing Verilog models. SystemVerilog 3.1 assertions provide consistent results between simulation and formal verification.

One important capability that SystemVerilog 3.1 adds to SystemVerilog 3.0 is the ability to define assertions outside of Verilog modules, and then bind them to a specific module or module instance. This allows test engineers to add assertions to existing Verilog models, without having to change the model in any way.

At the time of this writing, the semantics and capabilities for SystemVerilog 3.1 assertions had been defined, but the specific syntax was still being specified.

Direct Foreign Language Interface

SystemVerilog 3.1 will provide a means to SystemVerilog code to directly call functions written in another language (primarily C and C++), without having to use the Verilog Programming Language Interface (PLI). Verilog and SystemVerilog values can be passed directly to the foreign language function, and values can be received from the function. The foreign language function will also be able to call Verilog tasks and functions, which gives the foreign language functions access to simulation events and simulation time. This significantly improves the "bridge" that SystemVerilog provides between high-level system design and lower-level RTL and gate hardware design.

The SystemVerilog Direct Foreign Language Interface is based on the "DirectC" donation from Synopsys. The SystemVerilog standard subcommittee over this donation has carefully reviewed and modified this donation to provide an efficient and versatile interface between Verilog and C/C++.

Conclusion

SystemVerilog provides a major set of extensions to the Verilog-2001 standard. These extensions allow modeling and verifying very large designs more easily and with less coding. By taking a proactive role in extending the Verilog language, Accellera is accelerating the IEEE standardization process. It is expected that the IEEE Verilog standards group will adopt the SystemVerilog extensions as part of the next generation of the IEEE 1364 Verilog standard.

SystemVerilog 3.0 extends the modeling aspects of Verilog. It bridges the gap between hardware design engineers and system design engineers, allowing both teams to work in a common language. SystemVerilog 3.0 is a released Accellera standard, and is supported by the Synopsys/Co-design SystemSim simulator.

SystemVerilog 3.1 extends the verification aspects of Verilog, and will be released as a standard in June 2003. SystemVerilog 3.1 incorporates the capabilities of VERA-light and enhanced assertions. This release also incorporates a DirectC-like interface, allowing C, C++, SystemC and Verilog code to work together without the overhead of the Verilog PLI. This capability complete the bridge between high-level system design and low-level hardware design.

The SystemVerilog 3.0 and 3.1 extensions to Verilog create a whole new type of engineering language, an **HDVL**, or **H**ardware **D**escription *and* **V**erification **L**anguage. This unified language will greatly increase the ability to model huge designs, and verify that these designs are functionally correct.

References

[1] *"IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey, 2001.

[2] *"Verilog 2001: A Guide to the new Verilog Standard"*, by Stuart Sutherland, Kluwer Academic Publishers, Boston, Massachusetts, 2001.

[3] *"SUPERLOG® Extended Synthesizable Subset Language Definition"*, Draft 3, May 29, 2001, © 1998-2001 Co-Design Automation Inc.

[4] Paper "*Verilog, The Next Generation: Accellera's SystemVerilog*". A paper on SystemVerilog 3.0 by Stuart Sutherland, presented at HDLCon-2002, San Jose, CA. Available at www.sutherland-hdl.com/papers.

[5] *"SystemVerilog 3.0: Accellera's Extensions to Verilog"*, Accellera, Napa, California, 2002.

[6] *"SystemVerilog 3.1, draft 2: Accellera's Extensions to Verilog"*, Accellera, Napa, California, 2003.

About the author

Mr. Stuart Sutherland is a member of the Accellera SystemVerilog committee that is defining SystemVerilog, and is the technical editor of the SystemVerilog Reference Manual. He is also a member of the IEEE 1364 Verilog Standards Group, where he serves as co-chair of the PLI task force. Mr. Sutherland is an independent Verilog consultant, specializing in providing expert training on Verilog, SystemVerilog and the Verilog PLI. Mr. Sutherland can be contacted by e-mail at *stuart@sutherland-hdl.com*.