

# HDVL += (HDL & HVL)

## SystemVerilog 3.1

### The Hardware Description AND Verification Language

Stuart Sutherland  
Sutherland HDL, Inc.  
stuart@sutherland-hdl.com

Don Mills  
LCDM Engineering  
mills@lcdm-eng.com

#### ABSTRACT

What do you get when merge the Verilog HDL (Hardware Description Language) and the VERA HVL (Hardware Verification Language) together? You get SystemVerilog, the first full **HDVL**, or **Hardware Description *and* Verification Language!**

SystemVerilog is an extensive set of enhancements to the IEEE 1364 Verilog-2001 standard. These enhancements provide powerful new capabilities for modeling hardware at the RTL and system level, along with a rich set of new features for verifying model functionality. In his keynote address at the SNUG-Boston conference in September 2002, Synopsys CEO Aart De Geus stated that Synopsys is fully behind SystemVerilog, and will support these significant HDL and HVL language enhancements in VCS, Design Compiler, and other applicable Synopsys tools.

This paper presents an overview of the features in the released SystemVerilog 3.0 standard, and looks ahead at what is expected to be in the SystemVerilog 3.1 standard, which is planned for release in June 2003. The paper also presents Synopsys's plans to support SystemVerilog for simulation and synthesis. The primary objectives of this paper are to show the significant advantages of this novel HDVL approach, and to show that engineers can immediately utilize much of the capabilities of SystemVerilog using Synopsys tools.

## Table of Contents

1.0	Introduction .....	3
2.0	An overview of SystemVerilog 3.0 .....	3
2.1	Assertions .....	4
2.2	Interfaces .....	4
2.3	Global declarations and statements .....	5
2.4	Time unit and precision .....	5
2.5	Data types .....	5
2.6	User defined types .....	7
2.7	Enumerated types .....	7
2.8	Structures and unions .....	7
2.9	Module port connections .....	8
2.10	Literal values .....	8
2.11	Type casting .....	8
2.12	Operators .....	9
2.13	Unique and priority decision statements .....	9
2.14	Bottom testing loop .....	10
2.15	Jump statements .....	10
2.16	Block names and statement labels .....	10
2.17	New procedures .....	11
2.18	Task and function enhancements .....	11
2.19	Continuous assignment enhancements .....	12
3.0	An overview of SystemVerilog 3.1 .....	12
3.1	Classes .....	13
3.2	String data type .....	14
3.3	Arrays .....	14
3.4	Enumerations .....	14
3.5	Tasks and functions .....	14
3.6	Enhanced for loops .....	15
3.7	Enhanced fork—join .....	16
3.8	Inter-process synchronization .....	16
3.9	Random value constraints .....	17
3.10	Test bench program block .....	17
3.11	Clocking domain .....	18
3.12	Assertion extensions .....	19
3.13	Direct foreign language interface .....	19
3.14	Assertion API .....	20
4.0	Synopsys support for SystemVerilog .....	20
5.0	Conclusion .....	20
6.0	References .....	21
7.0	About the authors .....	21

## 1.0 Introduction

For many years, the behavioral coding features of Verilog, plus a few extras such as display statements and file I/O, gave Verilog-based hardware design engineers all they needed to both model hardware and to define a test bench to verify the model. As design sizes have increased, however, the amount of verification required has escalated dramatically. While writing the test bench and verification routines in pure Verilog HDL is still possible, the amount of coding far exceeds what can be accomplished in a reasonable amount of time. So along came proprietary Hardware Verification Languages (HVLs) such as VERA to the rescue. These languages specialized in giving verification engineers powerful constructs to describe stimulus and verify functionality in a much more concise manner. These proprietary languages solve a need, but at the costs of requiring engineers to learn and work with multiple languages, and often at the expense of simulation performance. Having different languages for the hardware modeling and the hardware verification has also become a barrier between those engineers doing the design work and those doing the verification. They don't speak the same language.

Enter the SystemVerilog standard currently being defined by Accellera. To provide Verilog designers with greater capability, at a faster pace, Accellera—the combined VHDL International and Open Verilog International organizations—has defined a set of high-level extensions to the IEEE 1364 Verilog-2001<sup>1,2</sup> language, known as **SystemVerilog**. These extensions provide powerful enhancements to Verilog, such as structures, interfaces, assertions, and much more.

Accellera plans to donate the SystemVerilog standard to the IEEE 1364 Verilog Standards Group. 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, 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 3.2** will continue to extend Verilog modeling and verification capabilities, based on the evolving needs of design and verification engineers.

The combination of the Verilog-2001 standard and the SystemVerilog extensions creates a new breed of hardware language—an **HDVL**, or **Hardware Description and Verification Language**. As the “+=” operation in the title of this paper suggests, combining HDL and HVL capabilities into one language results in even greater capabilities than just the merging of the individual languages.

## 2.0 An overview of SystemVerilog 3.0

This section of the paper presents an overview of the language features that make up SystemVerilog 3.0, which were ratified by Accellera in June, 2002. The SystemVerilog 3.0 Language Reference Manual<sup>3</sup> is publicly available from the Accellera web site ([www.accellera.org](http://www.accellera.org)).

As noted in the introduction, SystemVerilog 3.0 primarily extends the hardware modeling capabilities of Verilog. These extensions bridge a communication gap between high-level system architect engineers working in C, C++ or SystemC and hardware implementation engineers working at the detailed RTL and gate level of hardware. SystemVerilog enables both groups of engineers to communicate in the same language, and easily share their work without the need for language translators.

SystemVerilog 3.0 began with donations of major portions of the SUPERLOG language by Co-design<sup>5</sup>, and assertions work by Verplex and Intel (ForSpec). These donations were made in 2001. The Accellera HDL+ subcommittee then met an average of twice monthly to standardize these donations. A major part of this standardization effort has been to ensure that SystemVerilog is fully compatible with the IEEE 1364-2001 Verilog standard. Members of the Accellera HDL+ technical subcommittee include experts in simulation engines, synthesis compilers, verification methodologies, members of the IEEE Verilog Standards Group, and senior design and verification engineers.

The following subsections provide a brief explanation of many of the enhancements to Verilog provided with the SystemVerilog 3.0 standard.

## 2.1 Assertions

SystemVerilog 3.0 provides special language constructs to verify design behavior. An assertion is a statement that a specific condition, or sequence of conditions, in a design is true. If the condition or sequence is not true, the assertion statement will generate an error message. The message can be a default message, or a user-defined message.

A SystemVerilog assertion can be “immediate” or “temporal”. Immediate assertions test for a condition at current simulation time. Temporal assertions test for a condition, or sequence of conditions, at a future time. The future time can be at a specific time, or within some range of time. The following assertion example checks that in the `FETCH` state, `request` must be true immediately, and `grant` must become true within the next three clock cycles. Should either condition not occur, a default error message will be generated.

```
always @(State)
  if (State == FETCH)
    assert (request;[0:3];grant) @(posedge clk);
```

SystemVerilog 3.0 provides a basic set of assertion capabilities. SystemVerilog 3.1 will add several additional capabilities, which are presented later in this paper.

## 2.2 Interfaces

Verilog connects one module to another through module ports. This requires a detailed knowledge of the intended hardware design, in order to define the specific ports of each module that makes up the design. Several modules often have many of the same ports, requiring redundant port definitions for each module. Every module connected to a PCI bus, for example, must have the same ports defined.

SystemVerilog provides a new, high level of abstraction for module connections, called

*interfaces*. An interface is defined independent from modules, between the keywords `interface` and `endinterface`. Modules can use an interface the same as if it were a single port. In its simplest form, an interface can be considered a bundle of wires. All the signals that make up a PCI bus, for example, can be bundled together as an interface. Interfaces go beyond just representing bundles of interconnecting signals. An interface can also include functionality that is common to each module that uses the interface. In addition, an interface can include built-in protocol checking. Common functionality and protocol checking is possible because SystemVerilog interfaces can include parameters, constants, variables, structures, functions, tasks, initial blocks, always blocks, and continuous assignments.

### 2.3 Global declarations and statements

SystemVerilog adds an implicit top-level hierarchy, called `$root`. Any declarations or statements outside of a module boundary are in the `$root` space. All modules, anywhere in the design hierarchy, can refer to names declared in `$root`. This allows global variables, functions and other information to be declared, that are shared by all levels of hierarchy in the design.

### 2.4 Time unit and precision

In Verilog, time values are specified as a number, without any time unit. The Verilog standard does not specify a default unit of time or time precision (where precision is the maximum number of decimal places used in time values). The time units and precision are a property of each module, set by the compiler directive ``timescale`. There is an inherent danger with compiler directives, however, because they are dependent on source code order. If there are multiple ``timescale` directives in the source code, the last directive encountered before a module is compiled determines the time units of the module. If a module is not preceded by a ``timescale` directive, the time units and precision of that module become dependent on the order the source code is compiled. This can potentially cause different simulation runs to have different results.

SystemVerilog adds two significant enhancements to control the time units of time values. First, time values can have an explicit unit specified. The unit is one of `s`, `ms`, `us`, `ns`, `ps` or `fs`, for seconds down to femtoseconds. The time unit is a suffix to the time value, and cannot be preceded by a white space. For example:

```
    forever #5ns clock = ~clock;
```

Second, SystemVerilog allows the time units and time precision to be specified with new keywords, `timeunit` and `timeprecision`. These declarations can be specified within any module, or globally, in the `$root` space.

### 2.5 Data types

Verilog provides hardware-centric net, reg and variable data types. These types represent 4-state logic values, and are used to model and verify hardware behavior at a detailed level. The net data types also have multiple strength levels and resolution functions for zero or multiple drivers of the net. SystemVerilog adds several new data types to Verilog, which allow hardware to be modeled at more abstract levels, using data types more intuitive to C programmers.

- **char** — a 2-state signed variable, that is the same as the `char` data type in C. Note that the `char` data type is typically 8-bits wide, but the C standard does not guarantee this size.
- **byte** — a 2-state signed variable, that is defined to be exactly 8 bits.
- **shortint** — a 2-state signed variable, that is defined to be exactly 16 bits.
- **int** — a 2-state signed variable, that is similar to the `int` data type in C, but is defined to be exactly 32 bits.
- **longint** — a 2-state signed variable, that is defined to be exactly 64 bits, similar to the `longlong` type in C.
- **bit** — a 2-state unsigned data type of any vector width, that can be used in place of the Verilog `net` or `reg` data type, but with some restrictions.
- **logic** — a 4-state unsigned data type of any vector width, that can be used in place of either a Verilog `net` or `reg` data type, but with some restrictions.
- **shortreal** — a 2-state single-precision floating-point variable, that is the same as the `float` type in C.
- **void** — represents no value, and can be specified as the return value of a function.

The SystemVerilog `bit`, `byte`, `int`, `shortint`, and `longint` data types allow modeling designs using 2-state logic, which can be more efficient for simulation performance. Since the Verilog language does not have any 2-state data types, many simulators have provided the capability as an option to the simulator. These options do not work the same on every simulator, however, resulting in portability issues between different simulators. In addition, 2-state simulation modes often have the side effect of forcing 2-state logic in regions of a design where 3-state or 4-state logic is needed. The SystemVerilog 2-state data types can greatly improve simulation performance, while still allowing 3-state or 4-state logic in the regions of a design where needed. By using a data type with defined behavior instead of proprietary simulator options, models with 2-state logic are portable to all SystemVerilog simulators.

The SystemVerilog `logic` and `bit` data types are more versatile than the Verilog `net` and `reg` data types, which makes it easier to model hardware at any level of abstraction. The `logic` and `bit` types can receive a value any one of the following ways:

- Assigned values by any number of procedural assignment statements, replacing the `reg` type
- Assigned a value by a single continuous assignment statement, a restricted replacement for the `wire` type
- Connected to the output of a single primitive or module, a restricted replacement for the `wire` type

Since the `logic` and `bit` types can be used in place of either a Verilog `reg` or a `wire` (with restrictions), they allow writing models at a high level of abstraction, and adding details to the model as the design progresses, without having to change data type declarations. The `logic` and `bit` data types do not represent strength levels and do not have resolution functions for wired logic, which makes these types more efficient to simulate than the Verilog `wire` type.

## 2.6 User defined types

SystemVerilog provides a method to define new data types using `typedef`, similar to C. The user-defined type can then be used in declarations the same as with any data type.

```
typedef unsigned int uint;
uint a, b;
```

## 2.7 Enumerated types

SystemVerilog allows the creation of enumerated types, using a C-like syntax. An enumerated type has one of a set of named values. By default, the values increment from an initial value of 0, but the initial value can also be explicitly specified. The enumerated type will have the same vector size as the initial value.

```
enum {red, green, blue} RGB;
enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, DONE=3'b100} states;
```

An enumerated type can be defined as a user-defined type, using `typedef`.

```
typedef enum {FALSE=1'b0, TRUE} boolean;
boolean ready;
boolean test_complete;
```

## 2.8 Structures and unions

SystemVerilog adds structures and unions, which allow multiple signals, of various data types, to be bundled together and referenced by a single name.

```
struct {
    bit [15:0] opcode;
    logic [23:0] addr;
} IR;

union {
    int i;
    shortreal r;
} N;
```

User-defined types can be created from structures and unions, using `typedef`.

```
typedef struct {
    bit [15:0] opcode;
    logic [23:0] addr;
} instruction_t;

instruction_t IR;
```

Fields within a structure or union are referenced using a period between the structure or union name and the field name, as in C.

```
IR.opcode = 1; //set the opcode field in IR
N.r = 0.0; //set N as floating point value
```

A structure can be assigned as a whole, using a concatenation of values.

```
IR = {5, 200};
```

## 2.9 Module port connections

Verilog restricts the data types that may be connected to module ports to net types, and the variable types `reg`, `integer` and `time`. SystemVerilog removes all restrictions on connections to module ports. Any data type can be passed through ports, including reals, arrays and structures. When different data types are connected across module ports, standard rules define how values are coerced from one type to another.

SystemVerilog allows any variable type to be passed through module output ports. To prevent inadvertent wired-logic errors, SystemVerilog does not allow multiple outputs to be wired together using variables. Verilog net types, which have wired logic resolution functions, must be used to wire multiple outputs together.

## 2.10 Literal values

SystemVerilog adds the following enhancements to how literal values can be specified.

- All bits of a literal value can be filled with the same value using `'0`, `'1`, `'z` or `'x`. This allows a vector of any size to be filled, without having to explicitly specify the vector size of the literal value.

```
bit [63:0] data;  
data = '1; //set all bits of data to 1
```

- A string literal can be assigned to an array of characters. A null termination is added as in C. If the size differs, it is left justified, as in C.

```
char foo [0:12] = "hello world\n";
```

- Several special string characters have been added:

```
\v for vertical tab  
\f for form feed  
\a for bell  
\x02 for a hex number representing an ASCII character
```

- Arrays can be assigned literal values, using a syntax similar to C initializers, except that the replicate operator is also allowed. The number of nested braces must exactly match the number of dimensions (unlike C).

```
int n[1:2][1:3] = { {0,1,2}, {3{4}} };
```

## 2.11 Type casting

SystemVerilog adds the ability to change the type of a value using a cast operation, represented by `<type>'`. The cast can be to any type, including user-defined types.

```
int'(2.0 * 3.0) //cast result to int  
mytype'(foo) //cast foo to the type of mytype
```

A value can also be cast to a different vector size by specifying a decimal number before the cast operation.

```
17'( x - 2) //cast the operation to 17 bits
```

The signedness of a value can also be cast.

```
signed'(x) //cast x to a signed value
```

## 2.12 Operators

SystemVerilog adds several new operators:

- ++ and -- increment and decrement operators
- +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, <<<= and >>>= assignment operators

It is important to note that all these operators execute as Verilog blocking assignments. Care must be taken to avoid potential simulation race conditions, as with any blocking assignment. A general guideline is to avoid using these constructs in models representing sequential logic.

## 2.13 Unique and priority decision statements

The Verilog `if-else` and `case` statements can be a source of mismatches between RTL simulation and how synthesis interprets an RTL model, if strict coding styles are not followed. The synthesis `full_case` and `parallel_case` pragmas can lead to further mismatches if improperly used.

SystemVerilog adds the ability to explicitly specify when each branch of a decision statement is unique or requires priority evaluation. The keywords `unique` and `priority` can be specified before the `if` or `case` keyword. These keywords can be used to instruct simulators, synthesis compilers, and other tools the specific type of hardware intended. Tools can use this information to check that the `if` or `case` statement properly models the desired logic. For example, if a decision statement is qualified as `unique`, simulators can issue a warning message if an unexpected case value is found.

```
bit [2:0] a;

unique case(a)
  0, 1: y = in1;
  2: y = in2;
  4: y = in3;
endcase //values 3,5,6,7 will cause a warning

priority casez(a)
  3'b00?: y = in1; // a is 0 or 1
  3'b0???: y = in2; //a is 2 or 3;
  default: y = in3; //a is any other value
endcase
```

## 2.14 Bottom testing loop

Verilog has the `for`, `while` and `repeat` loops, all of which test to execute the loop at the beginning of the loop. SystemVerilog adds a `do-while` loop, which tests the loop condition at the end of executing code in the loop.

## 2.15 Jump statements

Verilog provides the ability to jump to the end of a statement group using the `disable` statement. Using `disable` to carry out the functionality of `break` and `continue` requires adding block names, and can create code that is non intuitive. SystemVerilog adds the C `break` and `continue` keywords, which do not require the use of block names, and a `return` keyword, which can be used to exit a task or function at any point.

- **break** — exits a loop, as in C
- **continue** — skips to the end of a loop, as in C
- **return** expression — exits a function
- **return** — exits a task or void function

SystemVerilog does not include the C `goto` statement.

## 2.16 Block names and statement labels

Verilog allows a `begin-end` or `fork-join` statement block to be named, by specifying the name *after* the `begin` or `fork` keyword. The name represents the entire statement block. SystemVerilog allows a matching block name to be specified after the block `end` or `join` keyword. This can help document which `end` or `join` is associated with which `begin` or `fork` when there are long blocks or nested blocks. The name at the end of the block is optional, and must match the name at the beginning of the block.

```
begin: foo //block name is after the begin
...
  fork: bar //nested block with a name
  ...
  join: bar //name must be the same
  ...
end: foo //name must be same as block name
```

SystemVerilog also allows individual statements to be labeled, as in C. A statement label is placed *before* the statement, and is used to identify just that statement.

```
initial begin
  test1: read_enable = 0;
  ...
  test2: for (i=0; i<=255; i++)
    ...
end
```

## 2.17 New procedures

Verilog uses the `always` procedure as a general purpose construct to represent RTL models of sequential logic, combinational logic and latched logic. The general purpose `always` procedure is also used in test benches and code that is not intended to be synthesized. Synthesis and other software tools must infer the intent of the `always` procedure from the context of the `@` event control at the beginning of the procedure (the “sensitivity list”) and the statements within the procedure. This inference can lead to mismatches in simulation and synthesis results.

SystemVerilog adds three new procedures to explicitly indicate the intent of the logic:

- `always_ff` — the procedure should represent sequential logic
- `always_comb` — the procedure should represent combinational logic
- `always_latch` — the procedure should represent latched logic

For example:

```
always_comb begin
    if (sel) y = a;
    else y = b;
end
```

Software tools can examine the event control sensitivity list and procedure contents to ensure that the functionality matches the type of procedure. For example, a tool can infer that an `always_comb` procedure must be sensitive to all external values read within the procedure. The tool can also check that the procedure makes assignments to the same variables for every branch of logic, and that the branches cover every possible condition. If either of these conditions are not true, then a software tool can report that the procedure does not properly model the engineers intent of combinational logic.

## 2.18 Task and function enhancements

SystemVerilog adds several enhancements to the Verilog task and function constructs.

**Static and automatic storage:** By default all storage within a Verilog task or function is static. Verilog-2001 allows tasks and functions to be declared as `automatic`, making all storage within the task or function automatic. With SystemVerilog:

- Specific data within a static task or function can be explicitly declared as `automatic`. Data declared as `automatic` has the lifetime of the call, and is initialized on each entry to the task or function call.
- Specific data within an automatic task or function can be explicitly declared as `static`. Data declared to be `static` in an automatic task or function has a static lifetime but a scope local to the task or function.

**Return from any point:** Verilog returns from a task or function when the execution reaches the `endtask` or `endfunction` keyword. The return value of a function is the last value assigned to the name of the function. SystemVerilog adds a `return` keyword, which allows a task or function to exit at any point.

**Multiple statements:** Verilog requires that a task or function have a single statement or statement block. Multiple statements must be grouped into a single `begin-end` or `fork-join` block. SystemVerilog removes the restriction of a single statement or block. Therefore, multiple statements can be listed in a task or function without using `begin-end` or `fork-join`. Statements that are not grouped will execute sequentially, as if within a `begin-end`. SystemVerilog also allows a task or function definition with no statements.

**Void functions:** The Verilog language requires that a function have a return value, and that function calls receive the return value. SystemVerilog adds a `void` data type, which can be specified as the return type of a function. Void functions can be called the same as a Verilog task, without receiving a return value. The difference between a void function and a task is that Verilog functions have several semantic restrictions, such as no time controls.

**Function inputs and outputs:** The Verilog standard requires that a function have at least one input, and that functions can only have inputs. SystemVerilog removes these restrictions. Functions can have any number of inputs, outputs and inout, including none.

## 2.19 Continuous assignment enhancements

In Verilog, the left-hand side of a continuous assignment can only be a net data type, such as `wire`. The continuous assignment is considered a driver of the net. Nets can have any number of drivers. SystemVerilog allows any variable data type to be used on the left hand side of a continuous assignment. Unlike nets, however, all variable data types are restricted to being driven by a single continuous assignment. It is illegal to mix continuous assignments and procedural assignments (including initial assignments) for the same variable.

## 3.0 An overview of SystemVerilog 3.1

The focus of SystemVerilog 3.1 is to extend the verification capabilities of Verilog (SystemVerilog 3.0 focussed on extending hardware modeling constructs). SystemVerilog 3.1 is expected to be ratified by Accellera in June, 2003. The current draft of the SystemVerilog 3.1 Language Reference Manual<sup>6</sup> is not publicly available at this time, as it is still being defined.

**Note that all information in this section is subject to change.** The SystemVerilog 3.1 standard is still being developed. The constructs presented in this paper serve to illustrate the powerful capabilities that SystemVerilog 3.1 will provide, but the final syntax of some constructs may be different in the final, ratified version of SystemVerilog 3.1.

Accellera is defining SystemVerilog from 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 primarily on donations from Co-design, Verplex and Novas. SystemVerilog 3.1 is based primarily on the following donations:

- 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 (“Sugar”).

These donations, along with some internal proposals, make up the core of the Verilog language enhancements that will be in SystemVerilog 3.1. The following subsections provide a more detailed overview of these features.

### 3.1 Classes

SystemVerilog 3.1 adds Object Oriented classes to the Verilog language, similar to C++. A class can contain data declarations, plus tasks and functions 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 provide a safe form of pointers. Class objects can inherit properties and methods from other classes, as in C++.

SystemVerilog does not require the complex memory allocation and de-allocation of C++. Memory allocation, de-allocation and garbage collection are automatically handled. This prevents the possibility of memory leaks.

An example of a SystemVerilog 3.1 object definition is:

```
class Packet ;
  bit [3:0] command;           // data portion
  bit [39:0] address;
  bit [4:0] master_id;
  integer time_requested;
  integer time_issued;
  integer status;

  function new();             // initialization
    command = 4'hA;
    address = 40'hFE;
    master_id = 5'b0;
  endfunction

  task clean();
    command = 4'h0; address = 40'h0;
    master_id = 5'b0;
  endtask

  // public access entry points
  task issue_request( int delay );
    // send request to bus
  endtask

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

### 3.2 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 character 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()`.

### 3.3 Arrays

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

Dynamic arrays are one-dimensional arrays where the size of the array can be changed dynamically. Object 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 enumerated type names. Associative arrays are sparse arrays, meaning that storage for each element in the array is not actually allocated until the location is accessed. SystemVerilog 3.1 also provides several class methods for working with associative arrays: `exists()`, `first()`, `last()`, `next()`, `prev()` and `delete()`.

### 3.4 Enumerations

SystemVerilog 3.0 adds enumerated types to the Verilog language. Enumerated types are strongly typed. An enumerated type 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++`. The SystemVerilog 3.0 cast operator can be used with enumerated types, so it is possible to do operations such as `State = State_t'(State + 1)`. However, the cast operator can only perform compile time checking. It cannot perform run-time checking that the result of an operation such as adding 1 to an enumerated type will result in a valid value in the enumerated type set.

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 run-time checking, and verify that, for example, `State = $cast(State + 1)` will result in a legal value in the `State` enumerated type set.

### 3.5 Tasks and functions

SystemVerilog 3.0 added several enhancements to Verilog tasks and functions, as discussed

earlier in this paper. SystemVerilog 3.1 adds additional enhancements.

**Passing arguments by name.** Verilog requires that a task or function call pass values in the order of the task/function argument declarations. 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

t1 (.c(out), .a(in1), .b(in2));
```

**Default argument values.** Task and function input arguments can be assigned a default value as part of the task/function 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
```

**Passing arguments by reference.** With SystemVerilog 3.1, a task or function argument can 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` [note: at the time of this writing, different keywords or tokens were being considered instead of using the VERA keyword `var`].

```
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
```

**Discarding function return values.** The return value of a function can be discarded by assigning the return to `void`. For example:

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

### 3.6 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)
    ...
```

### 3.7 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. Therefore, any statements following a `fork-join` are blocked from execution until all the forked parallel threads have completed execution.

SystemVerilog 3.1 adds significant new capabilities to the Verilog `fork-join` statement block, using the modifiers `join_none`, and `join_any`, where:

- `join_none` — statements that follow the `fork-join_none` are not blocked from execution while the parallel threads are executing. That is, each parallel thread is an independent, dynamic process.
- `join_any` — statements which follow a `fork-join_any` are blocked from execution until *any* thread has completed execution. That is, the join is blocked until the first thread completes.

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).

### 3.8 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, it 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`.

### 3.9 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 `randomize()` method, which is part of the `rand` class.

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

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

Using the `rand` and `randc` classes and methods, much more elaborate random number generation is possible than what is shown in the preceding simple example.

### 3.10 Test bench program block

In Verilog the test bench for a design must be modelled using Verilog hardware modeling constructs. Since these constructs were primarily intended for model hardware behavior 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 code block, 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 events in a “verification phase” appropriately synchronized to hardware simulation events

- Can use a special `$exit` system task that will wait to exit simulation until after all program blocks have completed execution (unlike `$finish`, which exits simulation immediately, even if there are pending events)

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

### 3.11 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 is 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 transition 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 may have race conditions, resulting in unpredictable 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 these races between sampling values and values changing on the same clock edge using standard Verilog, it is necessary to calculate a time shortly before the clock edge at 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, special events must be scheduled prior to the next clock edge. A delay must also 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 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.

### 3.12 Assertion extensions

The additional PSL (“Sugar”) assertion donation adds significant capabilities to the existing SystemVerilog 3.0 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. This 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. Therefore, no examples are included in this paper.

### 3.13 Direct foreign language interface

SystemVerilog 3.1 will provide a means for 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-level 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 extended this donation to provide an efficient and versatile interface between Verilog and C/C++.

### 3.14 Assertion API

SystemVerilog 3.1 will also include an Application Programming Interface (API) for assertions. This will extend the Verilog VPI routines to access to assertions, and enable external programs to find and read assertion statements anywhere in the design hierarchy. The external program will be able to enable or disable an assertion. The API will also provide a means to invoke an external program whenever an assertion triggers or an assertion fails or as an assertion steps through a sequence of events.

## 4.0 Synopsys support for SystemVerilog

Synopsys is taking a leading role in the EDA industry for implementing the SystemVerilog standard. The company has recognized the urgent need hardware modelers and verification engineers have for greater capability in the Verilog language. Synopsys is aggressively making the SystemVerilog extensions to Verilog available for engineers to use in today’s design and verification projects.

Current company plans<sup>6</sup> call for the Synopsys VCS simulator to support SystemVerilog 3.0 in version 7.0, due for release in March of 2003. This version is currently undergoing beta testing at a few customer sites. SystemVerilog 3.1 will be supported later in the year within VCS, pending ratification by Accellera. The Synopsys Design Compiler synthesis product is planning a limited beta release with SystemVerilog 3.0 support for key partners before the middle of 2003, with full support to follow. Support of SystemVerilog in other Synopsys products is also being planned.

## 5.0 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 providing a standard that can be implemented by simulator and synthesis companies quickly, without waiting for the protracted 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.

The SystemVerilog extensions create a whole new type of engineering language, an **HDVL**, or **Hardware Description and Verification Language**. This unified language will greatly increase the ability to model huge designs, and verify that these designs are functionally correct.

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. At the time this paper was written, SystemVerilog 3.0 was supported in the beta version of the Synopsys VCS 7.0 simulator, with plans for synthesis support in Design Compiler around mid 2003.

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 PSL (Sugar) 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 completes the bridge between high-level system design and low-level hardware design. It is expected that the Synopsys VCS simulator and DC synthesis compiler will begin supporting SystemVerilog 3.1 standard later in 2003.

## 6.0 References

- [1] “*IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language*”, IEEE, Piscataway, New Jersey, 2001.
- [2] S. Sutherland, “*Verilog 2001: A Guide to the new Verilog Standard*”, Kluwer Academic Publishers, Boston, Massachusetts, 2001.
- [3] “*SystemVerilog 3.0: Accellera’s Extensions to Verilog*”, Accellera, Napa, California, 2002.
- [4] “*SUPERLOG<sup>®</sup> Extended Synthesizable Subset Language Definition*”, Draft 3, May 29, 2001, © 1998-2001 Co-Design Automation Inc.
- [5] “*SystemVerilog 3.1, draft 2: Accellera’s Extensions to Verilog*”, Accellera, Napa, California, 2003.
- [6] Synopsys plans for supporting SystemVerilog were provided to the authors by David Kelf, Senior Director of Marketing, Verification Technology Group, Synopsys, Inc., on 28 January 2003.

## 7.0 About the authors

Mr. Stuart Sutherland is a member of the Accellera SystemVerilog technical subcommittee 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 chair of the PLI task force. Mr. Sutherland is an independent Verilog consultant, and specializes in providing comprehensive expert training on the Verilog HDL, SystemVerilog and PLI. Mr. Sutherland can be reached by e-mail at [stuart@sutherland-hdl.com](mailto:stuart@sutherland-hdl.com).

Mr. Don Mills is a member of the Accellera SystemVerilog technical subcommittee. He is an independent Verilog and VHDL consultant and works with two companies as an expert trainer on Verilog HDL. Mr. Mills has also been directly involved with Synopsys User Conferences (SNUG) for the past five years and is currently the technical chair of the European SNUG. Mr. Mills can be reached by e-mail at [mills@lcdm-eng.com](mailto:mills@lcdm-eng.com).