SystemVerilog, ModelSim, and You
(Is there anything in SystemVerilog useful in your work?)

Stuart Sutherland

Sutherland HDL, Inc., Portland, Oregon
www.sutherland-hdl.com

Abstract

SystemVerilog is *not* a new Hardware Description Language. SystemVerilog is a rich set of extensions to the existing Verilog HDL. The name "*SystemVerilog*" is somewhat of a misnomer. While it is true that one of the goals of SystemVerilog is to aid in modeling and verifying system-level designs, SystemVerilog provides extensions to Verilog that every engineer can and should take advantage of.

The SystemVerilog extensions to Verilog can be categorized as:

- Convenience extensions that make Verilog easier to use.
- Modeling extensions that make RTL models more accurate and consistent for both simulation and synthesis.
- Extensions for high-level behavioral modeling, which allow representing more functionality with fewer lines of code.
- Extensions for race-free, high-level verification programs.

This paper describes several extensions in each of these categories. A score card is then given to show which of these extensions work in the current release of the Mentor Graphics ModelSim simulator (version 5.8b). Insights on when ModelSim might support other features of SystemVerilog are also presented. Finally, the paper makes suggestions on how engineers can begin utilizing and benefitting from the SystemVerilog extensions to Verilog immediately.

The SystemVerilog extensions to Verilog are legion, and space within this paper does not permit a comprehensive treatise of all of these extensions. The SystemVerilog Language Reference Manual (LRM) [1] is the best source for a complete list of all the SystemVerilog extensions to the Verilog language.

1 Convenience extensions

The extensions listed in this section of the paper do not add significant new capabilities to the Verilog language, but do help make Verilog easier to use.

1.1 Time unit and precision

In Verilog, time values are a number, with no time units.

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

The time units and precision (where precision is the maximum number of decimal places used in time values) are properties of a software tool, set by the compiler directive `timescale. There is an inherent danger with compiler directives, however, because they are dependent on source code order. This can potentially cause different simulation runs to have different results.

SystemVerilog adds two extensions to specify time units. First, time values can have an explicit unit specified. The unit is one of **`s`**, **`ms`**, **`ns`**, **`ps`** or **`fs`**, representing seconds down to femtoseconds.

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

Second, SystemVerilog allows the time unit and precision to be specified within a module, making time units and precision part of the model. As part of the module, source code order dependencies are eliminated.

```
module chip (...);
  timeunit 1ns;
  timeprecision 10ps;
  ...
```

1.2 Filling vectors

With Verilog, it is easy to fill a vector of any width with all zeros, all Zs, or all Xs. However, Verilog does not have a simple way to fill a vector of any width with all ones. SystemVerilog adds a convenient shortcut to fill all the bits of a vector with the same value. The simple syntax is **`'0`**, **`'1`**, **`'z`** or **`'x`**.

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

1.3 New logic data type

SystemVerilog adds a new data type, **`logic`**, which is a synonym for the Verilog `reg` data type. The `logic` type solves a terminology problem that often confuses new Verilog users. The `reg` keyword would seem to imply "register", which then seems to imply that wherever the `reg` type is used, a hardware register is required. With experience, engineers learn that this implication is false. The `reg` data type is simply a programming variable. It is the context in which a variable is used that determines if a hardware register is required. The `logic` data type is the same as the `reg` type, but does not have a misleading name.

1.4 Relaxed rules for using variables

Verilog only permits variables to be used on the left-hand side of procedural assignments. It is illegal to use a variable on the left-hand side of continuous assignments or on the receiving side of a

module port. These contexts require a net data type, such as `wire`. This restriction on the use of variables can be a source of frustration. When creating a module, a designer must first determine how a signal will receive its values, in order to know what data type to use. If the way functionality is modeled changes, it may be necessary to change data type declarations.

SystemVerilog relaxes the rules on the usage of variables. A variable can be:

a) assigned values by any number of procedural assignment statements, or...

b) assigned a value by a single continuous assignment statement, or...

c) connected to a the output of a single primitive, or...

d) connected to the receiving side of a single module port.

With these relaxed rules, most signals can be declared as a variable, without concern for how the variable will receive its values. The only time a net data type is required is when a signal will have multiple drivers, such as on a bidirectional port.

The SystemVerilog rules for variables require that a variable can only have a single source for its value (from the list above). If a variable were to unintentionally receive values from a second source, an error would be reported. This can prevent unintentional multi-driver logic, which Verilog net types will allow.

1.5 Relaxed rules for passing values through ports

Verilog restricts the data types that can be passed through module ports to be only 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 (see 3.4).

```
module chip(input  real a, b,
            output real result);
```

1.6 Simplified module instantiations

Verilog provides two styles for instantiating modules. The first style connects signals to the module instance, using the order of the modules port declarations.

```
module chip;
  wire       q;
  reg  [3:0] d;
  reg        clk, rst;
  dff i1 (q, , clk, d[0], rst);
  ...
endmodule

module dff (output q, qb,
            input  clk, d, rst);
  ...
endmodule
```

The second style of connecting signals to a module instance uses the names of module ports.

```
dff i1 (.d(d[0]), .q(q), .clk(clk), .rst(rst));
```

Both styles have advantages and disadvantages. The port-order style is concise, but it is easy to inadvertently list signals in the wrong order. The named port connection style can eliminate inadvertent connection errors, but is verbose, and requires duplicating names when the signal and port names are the same.

SystemVerilog simplifies named port connection in two ways. First, if a signal name and port name are the same, then only the port name needs to be listed (called *dot-name* port connections).

```
dff i1 (.d(d[0]), .q, .clk, .rst);
```

Second, SystemVerilog adds a `.*` wild card port connection syntax. With dot-star, all ports and signals of the same name are automatically connected together.

```
dff i1 (.d(d[0]), .qb(), .*);
```

1.7 Function argument passing by name

Verilog requires that when a task or function is called, arguments must be passed in the order in which the formal arguments of the task or function are defined. SystemVerilog enhances task/function calls to allow values to be passed to a task or function in any order, using the task/function argument names. The syntax is the same as named module port connections.

1.8 Multiple statements in tasks and functions

Verilog requires that when there are two or more statements in a task or function, the statements must be grouped within `begin`...`end` keyword pairs. SystemVerilog simplifies tasks and functions by allowing any number of statements without having to specify `begin`...`end` keywords.

1.9 Function return values

Verilog has an unconventional mechanism for returning values from functions — the return value is assigned to the name of the function. SystemVerilog adds the C language return statement. This allows function return values to be specified in a more intuitive and self-documenting coding style.

1.10 Named block ends

In Verilog, blocks of code are grouped within `begin`...`end` blocks. Often, several `begin`...`end` blocks are nested, making it difficult to determine which `begin` belongs with which `end`. Verilog allows a name to be added after a `begin` keyword, which can help to document `begin`...`end` blocks.

SystemVerilog extends Verilog's named blocks by allowing a name to also be specified with the `end` keyword. The name with end must match the name of its corresponding `begin`. SystemVerilog also allows an optional name to be specified with other ending keywords, such as `endtask`, `endfunction` and `endmodule`.

```
module chip (...);
  always @(posedge clk) begin: output_register
    ...
    if (select) begin: muxed_inputs
      ...
    end: muxed_inputs
    ...
  end: output_register
endmodule: chip
```

2 RTL modeling extensions

SystemVerilog extends Verilog with several new constructs that simplify modeling at the Register Transfer Level. Some constructs help ensure that simulation and synthesis will interpret RTL modes in the same way. Many of these extensions are synthesizable using synthesis compilers available today.

2.1 New procedural blocks

Verilog uses the **always** procedural block to represent RTL models of sequential logic, combinational logic and latched logic. Synthesis and other software tools must infer the intent of the always procedural block from the context of the statements within the block. This inference can lead to mismatches in simulation and synthesis results.

SystemVerilog adds new procedural blocks that explicitly show the intent of the logic: **always_ff**, **always_comb**, and **always_latch**. An example of using these blocks is:

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

With the designer's intent explicitly stated, software tools can check that the procedural block functionality matches the intent. Errors can be generated if the code does not match the intent.

2.2 Unique and priority decision statements

Verilog defines that if...else and case statements evaluate in source code order. In hardware implementation, this would require additional priority encoding logic. Verilog does not require that a decision statement always execute a branch of code. If no branch is executed, storage may be required, typically in the form of latches. Synthesis compilers provide pragmas such as full_case and parallel_case to help control how synthesis handles decision statements, but these pragmas do not affect simulation behavior, and do not require that software tools verify that a decision statement meets the intent of the pragma.

SystemVerilog adds **unique** and **priority** keywords that specify how both simulation and synthesis should interpret decision statements. These modifiers also allow software tools to check that a decision statement meets the designer's intent, and issue error messages if the intent is not met.

```
priority if (a[2:1]==0) y = in1; //a is 0 or 1
    else if (a[2] == 0) y = in2; //a is 2 or 3
    else                y = in3; //a is any
                                 // other value

unique case(a)
  0, 1: y = in1;
  2:    y = in2;
  4:    y = in3;
endcase // unspecified values will be an error
```

When the priority decision modifier is specified, all tools must maintain the decision order of the source code. In addition, all tools must report an error if they detect that the decision was evaluated and no branch was executed.

When the unique decision modifier is specified, tools can optimize out the decision order. However, all tools are required to report an error, should the tool determine that two code branches could be true at the same time. In addition, all tools must report an error if it is detected that the decision was evaluated and no branch was executed.

2.3 2-state modeling

Verilog **reg** data types holds 4-state values (0, 1, Z and X). The Z value represents high-impedance, and is generally only used to represent tri-state logic. The X value is not a modeling value. It is a simulation value indicating an unknown condition.

SystemVerilog adds a 2-state **bit** data type, that can be used in place of the Verilog **reg** type. The bit type allows modeling at the RTL level using just logic 0 and 1. With the exception of tri-state outputs, synthesis compilers only use logic 0 and 1. Using the bit data type can help ensure that simulation and synthesis see the same logic values. The 4-state reg or logic types and the 2-state bit type can be mixed in the same model, giving the hardware designer complete control over what logic values can exist in different parts of the design.

2.4 Enumerated types

In Verilog, all signals must be a net or variable data type, or constant (such as parameter). Signals of these data types can have any value within their legal range. Verilog does not provide a way to limit the set of legal values for a variable.

SystemVerilog allows users to define enumerated types, using a C-like syntax. An enumerated type has one of a set of named values. These named values are the only legal values for that enumerated variable.

```
enum {WAIT, LOAD, DONE} states;
```

2.5 User defined types

SystemVerilog extends Verilog with a method for users to define new data types using **typedef**, similar to C. User-defined types can then be used in declarations, the same as with any data type.

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

2.6 Operators

Verilog does not have the C language **++**, **--** or assignment operators. Without these operators, code is more verbose.

```
for (data = 0; data <= 255; data = data + 1 )
```

SystemVerilog adds several new operators, including:

- **++** and **--** increment and decrement operators
- **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **^=**, **|=**, and other assignment operators

These operators simplify the coding of many types of operations. For example,

```
for (data = 0; data <= 255; data++ )
```

2.7 Enhanced for loops

Verilog for loops require a variable be declared prior to the loop, for use as the loop control.

```
module chip (...);
  integer i;
  always @(posedge clock)
    for (i=0, i < 127; i=i+1)
      ...
```

SystemVerilog enhances `for` loops to allow the loop control variable to be declared as part of the loop. SystemVerilog also allows the loop to contain multiple initial and step assignments.

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

3 High-level modeling extensions

A number of SystemVerilog extensions to the Verilog language allow modeling more functionality with fewer lines of code. These abstract modeling constructs enable representing very large designs in a concise yet intuitive manner.

3.1 Packages and compilation units

Verilog does not have a mechanism to define a block of code, such as a function, that can be shared by several modules.

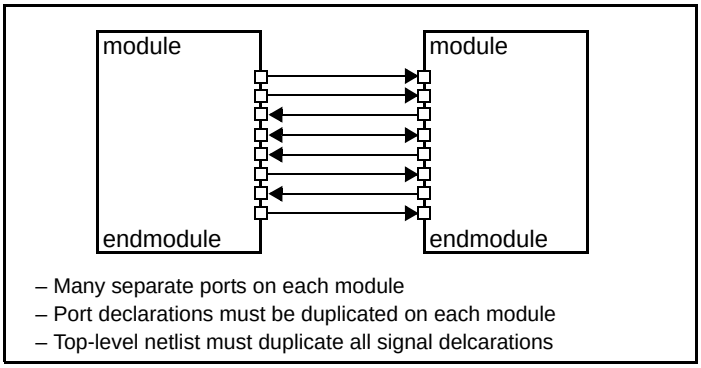
SystemVerilog adds a concept of packages to Verilog. Definitions of tasks, functions, user-defined types and shared variables can be defined in a package. Part or all of a package can then be imported into any number of modules. Packages also provide a mechanism to overload Verilog operators, thereby redefining the behavior of an operator for specific data types.

SystemVerilog also allows definitions such as user-defined types to be defined outside of module boundaries. These external definitions are then visible to all design blocks that are compiled at the same time. Design blocks that are compiled at the same time, along with any external definitions, make up a *compilation unit*. A primary usage of external definitions is to allow user-defined types to be passed through module ports.

3.2 Interfaces

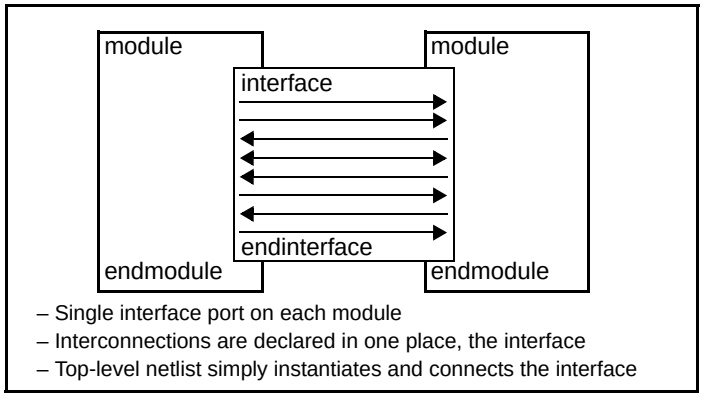
Verilog connects one module to another module through module ports. In order to begin writing a Verilog model, a detailed knowledge of the design interconnection is required. In large designs, it is common for several modules to have many of the same ports, requiring redundant port definitions for each module. Every module connected to a PCI bus, for example, must have the same ports defined.

Figure 1: Verilog module interconnect



SystemVerilog interfaces provide a new, high level of abstraction for module connections. An interface is defined independent from modules, between the keywords **`interface`** and **`endinterface`**. The interface encapsulates the interconnect information between modules, as well as communication information such as handshake sequences. 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. Interfaces go far beyond just representing bundles of interconnecting signals, however. 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.

Figure 2: SystemVerilog module interconnect




```
interface chip_bus;
  bit        clk, request, grant, ready;
  bit [63:0] address, data;

  // communication rotocol can go here
endinterface

module CPU (chip_bus io);
  ...
endmodule

module RAM(chip_bus pins);
  ...
endmodule

module top;
  chip_bus io(); //instantiate the interface
  RAM mem(io); //connect interface to module instance
  CPU cpu(io); //connect interface to module instance
endmodule
```

3.3 Abstract data types

Verilog provides hardware-centric net and variable data types. These types represent 4-state logic values, and are used to model and verify hardware behavior at a detailed level. Verilog's 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 allows modeling designs at more abstract levels.

- **`byte`** — an 8-bit 2-state signed variable.
- **`shortint`** — a 16-bit 2-state signed variable.
- **`int`** — a 32-bit 2-state signed variable, similar to the C `int`

data type, but is exactly 32 bits.

- **longint** — a 64-bit 2-state signed variable, similar to the C `long long` type.
- **shortreal** — a 2-state single-precision floating point variable that is the same as the C `float` type.
- **void** — represents no value, and can be specified as the return value of a function, the same as in C.

These new data types make it possible to write Verilog models at a higher level of abstraction, more like the C programming language. Because these data types are closely aligned with the C language, these data types also make it easy to integrate C and SystemC models with Verilog models. Another SystemVerilog extension to Verilog, the Direct Programming Interface, also facilitates model integrations (see item 4.9).

3.4 Structures and unions

SystemVerilog adds structures to the Verilog language. Structures allow multiple variables to be grouped together under a common name. These variables can then be assigned independently, as with any variable, or the entire group can be assigned in a single statement. The syntax is similar to C.

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

Individual members of a structure are referenced using a period between the variable name and the field name.

```
IR.opcode = 1; //set the opcode field in IR
```

All the members of a structure can also be assigned as a whole, using a list of values, as in C.

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

Structures can be assigned to structures, simplifying transferring one group of variables to another. Structures can also be passed through module and to or from a function or task.

3.5 Unpacked arrays

Verilog data types can be declared as arrays with any number of dimensions. Verilog restricts access to the elements of an array to just one element at a time.

```
bit [7:0] r1 [1:256]; //256 8-bit variables
bit [7:0] r2 [1:256];

for (i=1; i<=256; i=i+1)
  r2[i] = r1[i]; // copy 1 element at a time
```

SystemVerilog refers to a Verilog array as *an unpacked array.* With SystemVerilog, any number of dimensions of an unpacked array can be referenced at the same time. This allows all or part of an array to be copied to another array.

```
r2 = r1; // copy the entire array
```

SystemVerilog also allows all elements of an unpacked array to be set to a default value with a single assignment.

```
r1 = {default:8'hFF}; // initialize an array
```

3.6 Bottom testing loops

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. In many cases, a bottom-testing loop can reduce the number of lines of code required to model functionality.

3.7 Jump statements

Verilog provides the ability to jump to the end of a named statement group using the `disable` statement. One usage of disable is to exit a loop early, but this coding style can be awkward and non intuitive. SystemVerilog adds the C **break** and **continue** keywords to exit loops early in a more natural way. SystemVerilog also adds a **return** keyword, which can be used to exit a task or function at any point. SystemVerilog does not include the C `goto` statement.

3.8 Task and function argument pass by reference

Inputs to a task or function are copied in when the task or function is called. Outputs are copied back out when the task or function returns. With SystemVerilog, task or function arguments can also be passed by reference, instead of by copy. 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 argument direction is declared as a **ref**, instead of `input`, `output` or `inout`.

3.9 Redefinable data types

Verilog allows for parameterized modules, where constants such as vector widths can be redefined for each instance of a module.

```
module register #(parameter size = 16)
                 (output reg  [size-1:0] q,
                  input  wire [size-1:0] d,
                  input  wire clock, reset);
```

SystemVerilog extends this parameterized module capability to also allow for data types to be specified as parameters. This allows for polymorphic modules, where the data types of a module can be redefined for each instance of a module.

```
module multiplier #(parameter type VAR_TYPE = int)
            (input VAR_TYPE i, output VAR_TYPE o);
  ...
endmodule

module chip;
  ...
  multiplier #(.VAR_TYPE(real)) u1 (...);
  ...
```

4 Verification extensions

SystemVerilog adds many significant extensions to the Verilog language to enable representing advanced test programs using concise, object-oriented programming techniques. A few of these verification extensions are listed in this section.

4.1 Assertions

SystemVerilog adds assertions to the Verilog standard. These assertions constructs are aligned with the PSL assertion standard, but are adapted to fit syntactically and semantically in the Verilog language. There are two types of assertions, *immediate* and *concurrent*. Immediate assertions execute as a programming statement, similar to an `if...else` decision. These assertions are simple to use, and can simplify the verification and debug of even simple models. The following example asserts that at every change of state, the state value only has a single bit set.

```
always @(posedge clock) begin
  state <= next_state;
  assert $onehot(state); else $error;
end
```

Concurrent assertions execute in parallel with the Verilog code, and evaluate on clock cycles. A concurrent assertion is described as a *property*. A property can span multiple clock cycles, which is referred to as a *sequence*. SystemVerilog's PSL-like assertions can describe simple sequences and very complex sequences in short, concise sequence expressions. The example below asserts that when a request occurs, it must be followed by an acknowledge within one to three clock cycles.

```
property req_ack;
  @(posedge clk) req ##[1:3] ack;
endproperty

assert property (req_ack);
```

4.2 Test program blocks

In Verilog the testbench for a design must be modelled using Verilog hardware modeling constructs. Since these constructs are primarily intended to model hardware behavior, they have no special semantics to indicate how test values should be applied to the design. SystemVerilog adds a special type of code block, declared between the keywords **`program`** and **`endprogram`**. The program block has special semantics and syntax restrictions for modeling a testbench. A program block:

- can contain a single `initial` block
- executes events in a "*reactive phase*" of the current simulation time, appropriately synchronized to hardware simulation events
- can use a special **`$exit`** system task that will wait to exit simulation until after all concurrent program blocks have completed execution (unlike `$finish`, which exits simulation immediately, even if parallel tests are still running).

4.3 Classes

SystemVerilog adds object oriented classes to the Verilog language, similar to C++. A class can contain data declarations (referred to as "*properties*"), plus tasks and functions for operating on the data (referred to as "*methods*"). The properties and methods together define the contents and capabilities of an "*object*". Classes can have inheritance and public or private protection, as in C++. An example SystemVerilog object definition is:

```
class Packet;
  bit [ 3:0] command;
  bit [39:0] address;
  bit [ 4:0] master_id;

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

An object is created from a class definition using a **`new`** command, as in C++.

4.4 Dynamic arrays

SystemVerilog enhances Verilog arrays by adding dynamic arrays and associative arrays. Dynamic arrays are one-dimensional arrays where the size of the array can be changed dynamically. Built-in methods provide a means to set and change the size of dynamic arrays during run-time. Associative arrays are one-dimensional sparse arrays that can be indexed using values such as enumerated type names. Special built-in methods for working with associative arrays are provided: `exists()`, `first()`, `last()`, `next()`, `prev()`, and `delete()`.

4.5 Process synchronization

SystemVerilog provides built-in class objects for representing two common ways of synchronizing parallel activities within a testbench: *semaphores* and *mailboxes*.

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 built-in semaphore class provides several built-in methods for working with semaphores.

Mailboxes allow messages to be exchanged between processes. A message can be added to the mailbox by one process, and retrieved 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). The built-in mailbox class also includes several built-in methods.

4.6 Constrained random value generation

The Verilog standard includes a very basic random number function, called `$random`. This function, however, gives no control over the values generated. SystemVerilog adds two random number classes, **`rand`** and **`randc`**. 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 class called `Bus`, that can generate a random address and data value. A constraint on the address ensures that the lower two bits of a 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
Bus bus = new;
repeat(50) begin
  int 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.

4.7 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 adds two new ways for forked process to join:

- **`join_none`** — statements that follow the `fork...join_none` are not blocked from execution while the parallel threads are executing. Each parallel thread is an independent process.
- **`join_any`** — statements which follow a `fork...join_any` are blocked from execution until the first of any of the forked threads has completed execution.

4.8 Final blocks

Verilog has `initial` blocks that begin execution at the very beginning of simulation. SystemVerilog adds **`final`** blocks, which execute at the very end of simulation, just before simulation exits. Final blocks can be used in verification to print simulation results, such as code coverage reports.

4.9 Direct Programming Interface

SystemVerilog provides a means for Verilog code to directly call functions written in C, without having to use the Verilog Programming Language Interface (PLI). Verilog and SystemVerilog values can be passed directly to the C function, and values can be directly received from the function. With the DPI, Verilog code thinks it is calling a Verilog function, and is unaware that it has actually called a C function. Conversely, a C function is unaware that it was called from Verilog code.

The DPI provides a simple and intuitive mechanism to add C libraries to Verilog (such as the C math library). The DPI also makes it much easier to integrate SystemC models with Verilog simulations.

5 Using SystemVerilog with ModelSim

Mentor Graphics is aggressively working to implement the SystemVerilog extensions to Verilog in the ModelSim simulator. Version 5.8b of ModelSim already supports a number of features, and other features are being implemented. The next major release of ModelSim is expected to support most of the constructs covered in this paper.

Information on specific support for SystemVerilog is available in each release of ModelSim can be found in the release notes for each version. ModelSim users can receive notification as new SystemVerilog features are implemented by signing up for the ModelSim “Informant” E-newsletter.

6 Suggestions for adopting SystemVerilog

SystemVerilog provides a rich set of extensions to the Verilog language. Learning to properly use all of these extensions will take time and perhaps expert training. For novice Verilog users, some of the more advanced extensions, such as object-oriented testbench programming, may require a daunting learning curve.

Fortunately, SystemVerilog is based on the IEEE standard Verilog. It is not necessary to learn an entirely new language in order to begin taking advantage of the benefits of SystemVerilog. The author suggests applying the following incremental approach to adopting SystemVerilog extensions in current or next design projects:

1) Begin using the extensions designated as convenience extensions today. These extensions to Verilog are simple and intuitive, and make it easier to code any size design. Indeed, those new to Verilog will find that these convenience extensions make it easier to learn Verilog.
2) If supported by your software tools (particularly synthesis compilers), immediately begin using the RTL extensions to Verilog. Extensions such as the new procedural blocks allow modeling accurate functionality that will simulate and synthesize in the same way. More importantly, many of these RTL extensions allow software tools to better understand the designer’s intent, and issue warnings or errors if the model functionality does not match that intent.
3) As early as possible, begin adding basic assertions to designs. While the full capabilities of SystemVerilog assertions is complex, the basic usage and syntax of these assertions is relatively straight forward. Even basic designs will benefit from the run-time automatic checking of assertions. A good place to add an assertion is anywhere a comment is used to document the design intent or expectation.
4) As needed, add SystemVerilog verification constructs to the testbench. Some types of designs will not benefit significantly from these advanced verification capabilities. For more complex designs, however, these extensions to Verilog will both simplify the creation of test programs and the accuracy of these programs.

The table on the following page is provided as an aid in the decision process on adopting SystemVerilog. The table lists the SystemVerilog extensions to Verilog that are presented in this paper. This can be used as a check list for which constructs could be beneficial in current or planned design projects.

(Note: This table only lists the SystemVerilog extensions to Verilog included in this paper. There are many additional extensions that were not covered. Refer to the SystemVerilog

LRM [1] for a full description of all SystemVerilog extensions to Verilog.)

Table 1: Tool support for SystemVerilog

SystemVerilog Extension	Tool Support
Time unit and precision specification	
Filling vectors with all ones	
New `logic` data type (replaces `reg`)	
Relaxed rules for using variables	
Relaxed rules for passing values through ports	
Simplified module instances (dot-name, dot-star)	
Function argument passing by name	
Multiple statements in tasks and functions	
Function return values using `return`	
Named block ends	
New procedural blocks (`always_comb`, ...)	
`unique` and `priority` decision modifiers	
2-state `bit` data type	
New operators (`++`, `--`, `+=`, ...)	
Enumerated types	
User-defined types (`typedef`)	
`for` loop local declarations, multiple variables	
Packages and external declarations	
Interfaces	
New data types, `int`, `shortint`, `bit`, ...	
Structures and unions	
Assign to multiple array elements, copy arrays	
`do`...`while` bottom testing loop	
`break`, `continue`, `return` jump statements	
task/function argument passing by reference	
Parameterized data types	
Assertions	
Testbench program blocks	
Classes	
Dynamic arrays	
Test synchronization (semaphores, mailboxes)	
Constrained random values	
Final blocks	
Direct Programming Interface (DPI)	

7 Other EDA companies and SystemVerilog

This paper is not a competitive analysis of EDA products. However, when adopting new language features, it is important to know if these features are portable across a variety of EDA tools. For the objectives of this paper on using ModelSim with the SystemVerilog extensions to Verilog, it is sufficient to state that all major and many minor EDA companies either have implemented, or are rapidly implementing, SystemVerilog constructs in their existing Verilog products. Many SystemVerilog extensions to Verilog can be used with a variety of EDA tools in currently shipping versions of the products. This includes the key enabling tools of simulation and synthesis.

8 Conclusion

SystemVerilog provides a major set of extensions to the Verilog-2001 standard. These extensions make Verilog:

- easier to use
- more accurate for RTL simulation and synthesis
- more efficient for modeling at high levels of abstraction
- easier to verify large designs with concise, accurate test benches.

Mentor Graphics has already implemented many of these extensions to Verilog in the current release of the ModelSim simulator. Other EDA companies have implemented—or are currently implementing—SystemVerilog extensions in synthesis compilers, lint checkers, formal tools, hardware accelerators, and other electronic design tools. This broad range of industry support for SystemVerilog means that you can begin using these great extensions to Verilog today!

9 References

[1] *SystemVerilog 3.1a: Accellera's Extensions to Verilog*, Accellera, Napa, CA, 2003. Available in PDF form at www.systemverilog.com

[2] *IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language*, IEEE, Pascataway, NJ, 2001. ISBN: 0-7381-2827-9 (printed), 0-7381-2827-9 (PDF).

[3] *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*, by Stuart Sutherland, Simon Davidmann and Peter Flake. Kluwer Academic Publishers, Boston, MA, 2004, ISBN: 0-4020-7530-8.

[4] *Verilog 2001: A Guide to the new Verilog Standard*, by Stuart Sutherland. Kluwer Academic Publishers, 2001, ISBN: 0-7923-7568-8.

10 About the author

Mr. Stuart Sutherland is a member of the Accellera technical subcommittee that defined SystemVerilog, and is the technical editor of the SystemVerilog Language 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, and specializes in providing comprehensive expert training on the Verilog, SystemVerilog and the Verilog PLI. Mr. Sutherland can be reached by e-mail at `stuart@sutherland-hdl.com`. Other papers by Stuart Sutherland are available at `www.sutherland-hdl.com`.