SystemVerilog Is For Everyone
(not just system designers)

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

SystemVerilog is *not* a new Hardware Description Language. SystemVerilog is a rich set of extensions to the existing Verilog HDL. In my work as a Verilog and SystemVerilog consultant and trainer, I have occasionally heard engineers make comments such as, *"I'm not doing system level design, so I don't need SystemVerilog."* ***That is a misconception!*** It is true that the primary goal of SystemVerilog is to enable modeling and verifying large, complex designs. However, SystemVerilog provides enhancements to Verilog that every engineer can and should take advantage of. SystemVerilog makes it easier to model with Verilog, and helps ensure that models will both simulate and synthesize correctly.

This article dispels the false impression that only system-level designers need SystemVerilog. The article briefly describes 14 of the enhancements to Verilog that will be of interest to all Verilog users, no matter what type of design they are modeling. SystemVerilog has much, much more to offer. As engineers become familiar with all of SystemVerilog, they will discover many other enhancements that will be useful in their engineering work.

1. Time unit and precision

In Verilog, time values are specified as a number, without any time unit. For example:

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

The Verilog standard does not specify a default unit or time precision (where precision is the maximum number of decimal places used in time values). The time units and precision 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 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`**, **`ns`**, **`ps`** or **`fs`**, representing seconds down to femtoseconds. 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 a module, thus making time units and precision part of the model, instead of a command to a software tool.

```
timeunits 1ns;
timeprecision 10ps;
```

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`**. 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
```

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** — 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 C `int` data type, 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 C `long long` type.
- **bit** — a 2-state unsigned data type of any vector width that can be used in place of the Verilog `reg` data type.
- **logic** — a 4-state unsigned data type of any vector width that can be used in place of the `reg` data 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.

The SystemVerilog 2-state data types allow modeling designs at a more natural level. Most digital logic works with just zeros and ones. The special value of Z is only needed to represent tri-state logic, which is rare in most designs. The special value of X is not a modeling value. It is a simulation value indicating an unknown condition.

The SystemVerilog **logic** data type is a synonym for the Verilog `reg` data type. It solves a terminology problem that has plagued new Verilog users since the dawn of RTL synthesis. The `reg` keyword would seem to imply "register", which would then seem to imply that each place a `reg` data type is used a hardware register is required With experience, Verilog users learn that this implication is false. The `reg` data type is simply a programming variable. It is the context in which the variable is used that determines whether or not a hardware register is required. The `logic` data type is the same as the `reg` type, but does not have a misleading name.

4. Relaxed rules for variables

With Verilog, variables can only 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 variables is often 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 the functionality of a design is modeled changes, it is often necessary to go back and change data type declarations.

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

- assigned values by any number of procedural assignment statements, or...
- assigned a value by a single continuous assignment statement, or...
- connected to a the output of a single primitive, or...
- connected to the receiving side of a single module port.

These relaxed rules simplify creating Verilog models. Almost all 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, for example, a variable were used on the left-hand side of a continuous assignment, and the same variable was unintentionally connected to the input port of the module, an error would be reported. Verilog would have required net types in this context, which would allow unintentional multi-driver logic to compile, resulting in functional errors.

5. User defined types

Verilog does not allow users to define new data 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;
```

6. Enumerated types

In Verilog, all signals must be a net, variable, or parameter data type. Signals of these data types can have any value within their legal range. The Verilog language 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;
```

An enumerated type can be used as a user-defined data type, allowing the type to be used in many places.

```
typedef enum {FALSE, TRUE} boolean;
boolean ready;
boolean test_complete;
```

7. 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 declaration syntax is similar to C.

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

A structure definition can be given a name using `typedef`.

```
typedef  struct {
    bit [7:0] opcode;
    bit [23:0] addr;
} instruction_t; //named structure type

instruction_t IR, stack; //allocate structure
```

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.

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

Structures can be assigned to structures, simplifying transferring one group of variables to another.

```
IR = stack;
```

Structures can also be passed to or from a function or task, and can be passed through module ports.

8. Arrays

Verilog data types can be declared as arrays. The `reg` and net types can also have a vector width declared. The arrays can have 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*. 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 initialed to a default value with a single assignment.

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

9. Module port connections

Verilog restricts the data types that can be connected to module ports. Only net types and the variable types `reg`, `integer`, or `time` can be passed through module ports.

SystemVerilog removes all restrictions on connections to module ports. Any data type can be passed through ports, including reals, arrays, and structures.

```
typedef  struct {
    bit [7:0] opcode;
    bit [23:0] addr;
} instruction_t; //named structure type

module chip(input  instruction_t IW,
            input  bit           clk, reset,
            output bit [31:0]  result);
```

10. 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 **`>>>=`** assignment operators

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

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

11. Unique and priority decision statements

Verilog defines that `if...else` and `case` statements evaluate in source code order. In hardware implementation, this would require extra, priority encoding logic. Synthesis will optimize out this extra logic if it can determine that all branches of the decisions are mutually exclusive (unique).

The Verilog language does not require that a decision statement always execute a branch of code. Should this occur, synthesis will add latches to the implementation.

SystemVerilog adds the ability to specify when each branch of decision statements is unique or requires priority evaluation, using the keywords **`unique`** and **`priority`**.

```
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
```

The `unique` and `priority` modifiers instruct simulators, synthesis compilers, and other tools as to the type of hardware intended. Tools can use this information to check that the code properly models the desired logic.

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.

12. 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 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`**, **`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 intent explicitly stated, software tools can check that the procedural block functionality matches the type of procedure. Errors or warning can be generated if the code does not match the intent.

13. Task and function enhancements

SystemVerilog adds several enhancements to the Verilog task and function constructs. Only two of these enhancements are mentioned in this article.

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. The difference between a void function and a task is that functions have several restrictions, such as not allowing time controls. These restrictions help ensure that the logic in a function will synthesize correctly. By modeling with void functions instead of tasks, engineers can have greater confidence that their models will synthesize correctly.

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 inouts, including none.

14. Assertions

SystemVerilog adds assertions to the Verilog standard. These assertions constructs are aligned with the PSL assertion standard, but are adapted to fit syntactically in the Verilog language. There are two types of assertions, immediate and sequential. 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 @(state)
  assert (state == $onehot); else $fatal;
```

Sequential assertions execute in parallel with the Verilog code, and evaluate on clock cycles. A sequential 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);
```

Conclusion

SystemVerilog is for every Verilog engineer!

SystemVerilog provides a major set of extensions to the Verilog-2001 standard. Some of the extensions to Verilog are most useful for modeling and verifying very large designs more easily and with less coding. However, many of the SystemVerilog extensions to Verilog make it easier to model accurate, synthesizable models of any size designs. These extensions make Verilog easier to use, and are truly beneficial to every engineer that works with Verilog.

References

[1] *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling,* Stuart Sutherland, Kluwer Academic Publishers, Boston, MA, 2004, 0-4020-7530-8.

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

[3] *SystemVerilog 3.1: Accellera's Extensions to Verilog*, Accellera, Napa, CA, 2003.

[4] *IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language*, IEEE, Pascataway, NJ, 2001.

About the author

Mr. Stuart Sutherland is a member of the Accellera HDL+ technical subcommittee that is defining 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`.