Verilog, The Next Generation: Accellera's SystemVerilog

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

Abstract

This paper provides an overview of the proposed Accellera SystemVerilog standard. SystemVerilog is a blend of C, C++, SUPERLOG and Verilog, which greatly extends the ability to model designs at an abstract architectural level. The paper also discusses the current status of the proposed standard.

1. Introduction

Moore's law is still proving to be true; the designs we are creating keep getting larger and larger at an amazing pace. Hardware design and verification languages need to keep up with the pace of ever increasing design sizes. The IEEE 1364 Verilog-2001[1,2] standard helps, with many significant enhancements to the Verilog language. But, the IEEE standardization process is slow and cumbersome. It took four years from the time work was started on Verilog-2001 until the IEEE ratified the standard. By time Verilog-2001 was approved, our design needs were already demanding more from the language. We cannot wait another 4 years for the next set of enhancements to the Verilog language.

To provide Verilog designers with greater capability, at a faster pace, Accellera—the combined VHDL International and Open Verilog International organizations—has proposed a set of high-level extensions to the Verilog language, known as SystemVerilog[3]. These extensions provide powerful enhancements to Verilog, These extensions provide powerful enhancements to Verilog, such as C language data types, structures, packed and unpacked arrays, interfaces, assertions, and much more.

Origins of SystemVerilog: SystemVerilog began with donations of major portions of the SUPERLOG language by Co-design[4], and assertions work by Verplex. These donations were made about July, 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.

2. Interfaces

Background: 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. Early in a design cycle, this detail may not be well established, yet it is difficult to change the port configurations of a design once the initial module ports have been defined. In addition, 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. Using interfaces makes it possible to begin a design without first establishing all of the interconnections between modules. As the details of design become more established, the specific signals within the interfaces can be easily be represented; any changes within the interfaces will be reflected in all modules using the interfaces, without having to change each module..

```
interface chip_bus; // Define the interface
  wire read_request, read_grant;
  wire [7:0] address, data;
endinterface: chip_bus

module RAM(chip_bus io, //use the interface
           input clk);
  //io.read_request references a signal in
  // the interface
endmodule
```

```
module CPU(chip_bus io, input clk);
  ...
endmodule

module top;
  reg clk = 0;
  chip_bus a; //instantiate the interface

  //connect interface to module instances
  RAM mem(a, clk);
  CPU cpu(a, clk);
endmodule
```

SystemVerilog interfaces go beyond just representing bundles or interconnecting signals. An interface can also include functionality that is shared by each module that uses the interface. In addition, an interface can include built-in protocol checking. Shared functionality and protocol checking is possible because SystemVerilog interfaces can include parameters, constants, variables, structures, functions, tasks, initial blocks, always blocks, and continuous assignments.

3. Global declarations and statements

Background: Verilog does not have a global space, other than that the names of modules can be referenced from any other module, as module instances. Verilog also allows any number of top-level modules, which creates unrelated hierarchy trees.

SystemVerilog adds an implicit top-level hierarchy, called **`$root`**. Any declarations or statements outside of a module boundary will be 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.

```
reg error _flag; //global variable

function compare (...); //global function

always @(error_flag) //global statement
  ...

module test;
  chip1 u1 (...)
endmodule

module chip1 (...);
  FSM u2 (...);
  always @(data)
    error_flag = compare(data, expected);
endmodule

module FSM (...);
  ...
  always @(state)
    error_flag = compare(state, expected);
endmodule
```

4. Time unit and precision

Background: 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 or time precision (where precision is the maximum number of decimal points 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 elaborated 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`**, **`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. The units and precision must be a power of 10, ranging from seconds down to femtoseconds.

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

5. Abstract data types

Background: 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 includes the C language `char` and `int` data types, which allows C and C++ code to be directly used in Verilog models and verification routines. The Verilog PLI is no longer needed to integrate Bus Functional Models, algorithmic models, and C functions. SystemVerilog also adds several new data types to Verilog, which allow hardware to modeled at more abstract levels.

- **`char`** — a 2-state signed variable, that is the same as the C `char` data type, which may be an 8 bit integer (ASCII) or a `short int` (Unicode).

- **int** — a 2-state signed variable, that is similar to the C `int` data type, but is defined to be exactly 32 bits.
- **shortint** — a 2-state signed variable, that is defined to be exactly 16 bits.
- **longint** — a 2-state signed variable, that is defined to be exactly 64 bits, similar to the C `long long` type.
- **byte** — a 2-state signed variable, that is defined to be exactly 8 bits.
- **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 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 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 **bit** and other data types allow modeling designs using 2-state logic, which is much more efficient for simulation performance. Since the Verilog language does not have a two-state data type, many simulators have provided the capability as an option to the simulator. These options are not portable to all simulators, however, and 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 `bit` data type 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, 2-state models will be portable to all SystemVerilog simulators.

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

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

Since the `logic` type can be used in place of either a Verilog `reg` or a `wire` (with restrictions), it allows 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` data type does not represent strength levels and does not have resolution functions for wired logic, which makes the `logic` type more efficient to simulate and synthesize than the Verilog `wire` type.

6. Signed and unsigned modifiers

Background: By default, the Verilog `net` and `reg` data types are unsigned types, and the `integer` type is a signed type. The Verilog-2001 standard allows the unsigned types to be explicitly declared as signed data types using the `signed` keyword.

SystemVerilog adds the counterpart, the capability to explicitly declare signed data types to be unsigned, using the **unsigned** keyword. For example:

```
int unsigned j;
```

It should be noted that `unsigned` is a reserved word in the Verilog language, but is not used by the Verilog standard.

7. User defined types

Background: 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;
```

A user-defined type can be used before it is defined, provided it is first identified using an empty `typedef`.

```
typedef int48; //full definition is elsewhere
int48 c;
```

8. Enumerated types

Background: Verilog does not have enumerated types. Identifiers must be explicitly declared as a net, variable or parameter, and assigned values.

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, yellow, green} RGB;
enum {WAIT=2'b01, LOAD, DONE} states;
```

Using `typedef`, an enumerated type can be given a name, allowing the type to be used in many places.

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

9. Structures and unions

Background: The Verilog HDL does not have structures or unions, which are useful for grouping several declarations together.

SystemVerilog adds structures and unions. The declaration syntax is similar to C.

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

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

Fields within a structure or union are referenced using a period between the variable name and the field name.

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

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

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

instruction IR; //allocate a structure
```

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

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

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

10. Arrays

Background: Verilog HDL data types can be declared as arrays. The `reg` and net types can also have a vector width declared. A dimension declared before the object name is the *"vector width"* dimension. The dimensions declared after the object name are the *"array"* dimensions.

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

SystemVerilog uses different terminology. The term *"packed array"* is used to refer to the dimensions declared before the object name, instead of vector width. The term *"unpacked array"* is used to refer to the dimensions declared after the object name. Packed arrays can only be made of the types: `bit`, `logic`, `reg`, `wire`, and the other net types. Multiple dimensions can be declared for packed arrays as well for unpacked arrays.

```
bit [7:0] a; //a 1-d packed array
bit b [7:0]; //a 1-d unpacked array
bit [0:11] [7:0] c; //a 2-d packed array
bit [3:0] [7:0] d [1:10]; /* a 10 element
  unpacked array of a packed array, consisting
  of 4 8-bit bytes */
```

Unpacked dimensions are referenced before packed dimensions. This allows referencing an entire packed array as a single element. In the last example above, `d[1]` refers to a single element in the unpacked array. That element is an array of four bytes.

11. Declarations in unnamed blocks

Background: Verilog allows variables to be declared in a named `begin`–`end` or `fork`–`join` statement group. These variables are local to the group, but can be referenced hierarchically.

SystemVerilog allows declarations to be made in unnamed blocks as well as in named blocks. In an unnamed block, a hierarchical name cannot be used to access the variable. All variable types, including user-defined types, enumerated types, structures and unions can be declared within a `begin`–`end` or `fork`–`join` statement group

12. Constants

Background: Verilog has three specific types of constants: `parameter`, `specparam` and `localparam`.

SystemVerilog allows any data type to be declared as constant, using the **`const`** keyword.

```
const char colon = ":";
```

13. Redefinable data types

SystemVerilog extends the Verilog `parameter` to include **`type`**. This powerful feature allows the data types within a module to be redefined for each instance of the module.

```
module foo
  #(parameter type VAR_TYPE = shortint;)
  (input logic [7:0] i, output logic [7:0] o);
  VAR_TYPE j = 0;  /* j is of type shortint
                      unless redefined */
    ...
endmodule

module bar;
  logic [3:0] i,o;
  foo #(.VAR_TYPE(int)) u1 (i, o);
    //redefines VAR_TYPE to a type of int
endmodule
```

14. Module port connections

Background: 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.

15. Literal values

Background: Verilog has several limitations when specifying or assigning 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 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 of braces must exactly match the number of dimensions (unlike C).

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

16. Type casting

Background: The Verilog language does not have the ability to cast values to a different data type.

SystemVerilog adds the ability to change the type of a value to a 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
```

17. Operators

Background: Verilog does not have the C language **++**, **--** or the C increment and decrement assignment operators.

SystemVerilog adds several new operators:

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

18. Unique and priority decision statements

Background: 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 decision statements 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 if ((a==0) || (a==1)) y = in1;
else if (a == 2) y = in2;
else if (a == 4) y = in3;
//values 3,5,6,7 will cause a warning

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 //values 3,5,6,7 will cause a warning

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

19. Bottom testing loop

Background: 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.

20. Jump statements

Background: The C language provides several means to jump to a new statement in execution flow: `return`, `break`, `continue` and `goto`. Verilog does not have any of these statements, and instead 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.

21. Block names and statement labels

Background: 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 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
```

22. Event control enhancements

Background: Verilog uses the @ token to control execution flow based on specific events.

SystemVerilog enhances the @ event control.

Conditional event control: One common usage of @ is to infer latch behavior with an enable input. The following example illustrates a common style for modeling a latch:

```
always @(data or en) //RTL latch model
  if (en) y <= data;
```

This coding style can be inefficient for simulation, because even when the enable input is not asserted, the event control will trigger on every change of the data input.

SystemVerilog adds an **`iff`** condition that can be specified in event controls. The `iff` condition must be true in order for the control to trigger. By moving the enable decision into the event control, the control will only trigger when the latch output can change.

```
always @(a or en iff en==1)
  y <= a;
```

Expressions in event controls: Verilog allows expressions to be used in an @ event control list. For example:

```
always @( (a * b) )
always @(memory[address])
```

In first example, should the event control trigger when the operands change, or only if the result of the operation changes? In second example, should the event control trigger when the address to the memory changes, or only if the value in the selected memory address changes? The IEEE Verilog standard allows simulators to optimize differently when the @ event control contains an expression. This can lead to different behavior in different simulators, and a mismatch in simulation and synthesis results.

SystemVerilog adds a **`changed`** keyword, which is used as a modifier in the event control list. The `@(changed` *expression*`)` explicitly defines that the event control only triggers on a change of the result of the expression.

```
always @(changed (a*b) )
always @(changed memory[address])
```

Assignments in event controls: Verilog does not allow an assignment to made within an event control.

SystemVerilog allows assignment expressions to be used in an event control. The event control is only sensitive to changes on the right side of the assignment.

```
always @( (y = a * b) )
```

23. New procedures

Background: Verilog uses the **`always`** procedure to represent RTL models of sequential logic, combinational logic and latched logic. 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 @(a or b or sel) 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

check that an `always_comb` procedure is sensitive to all external values read within the procedure, makes assignments to the same variables for every branch of logic, and that branches covers every possible condition. If any of these conditions are not true, then a software tool can report that the procedure does not properly model combinational logic.

24. Dynamic processes

Background: Verilog provides a form of static concurrent processes using `fork–join`. Each branch of a fork is a separate, parallel process. Execution of any statements which follow the `fork–join` will not be executed until every process in the group has completed.

```
initial
  begin
    fork
      send_packet_task(1,255,0);
      send_packet_task(7,128,5);
      watch_result_task(1,255,0);
      watch_result_task(7,128,5);
    join //all tasks must complete to get here
  end
```

SystemVerilog adds a new, dynamic process, using the **`process`** keyword. This forks off a process, and then continues execution without waiting for the process to complete. The process does not block the flow of execution of statements within the procedure or task. This allows multi-threaded processes to be modeled.

```
initial
  begin
    process send_packet_task(1,255,0);
    process send_packet_task(7,128,5);
    process watch_result_task(1,255,0);
    process watch_result_task(7,128,5);
  end //all processes run in parallel
```

25. 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 or block, 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 block.

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, as discussed in section 20 of this paper. Using this keyword, a task or function can be exited 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`. It also legal to create 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 functions have several 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 inouts, including none.

26. Continuos assignment enhancements

Background: 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 except `reg` to be used on the left hand side of a continuous assignment. Unlike nets, however, all other 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.

27. $bits system function

Background: Verilog does not have an equivalent to the C `sizeof` function.

SystemVerilog adds a new **`$bits`** built-in system function. This function returns the number of hardware bits required

to hold a value (a 4-state value requires one hardware bit, even though it might require multiple bits to store within simulation). This function can also be used to determine the number of bits represented by a structure.

28. ‘define enhancements

SystemVerilog enhances the capabilities of the `‘define` compiler directive to support strings as macro arguments. The macro text string can include an isolated quote, which must be preceded by a back tick (`` `" ``), which allows macro arguments to be included in strings. The macro text can include a backslash (\ ) at the end of a line to show continuation on the next line. If the macro text string is to contain a backslash, the backslash should be enclosed in back ticks (`` `\` ``), so that it will not be treated as the start of a Verilog escaped identifier. The macro text string can also include a double back tick (`` ``), which allows identifiers to be constructed from arguments. These enhancements make the `` `define `` directive much more versatile. For example, the `` `include `` directive can be followed by a macro name instead of a literal string.

```
`define f1 "../project_top/opcode_defines"
`include `f1
```

29. State machine modeling

The Accellera HDL+ committee defining the SystemVerilog standard is currently evaluating additional constructs that will allow modeling complex state machines at a higher level of abstraction than is possible with Verilog. These constructs include:

- Enumerated types
- A special `state` data type
- A transition statement
- A transition operator

Enumerated types are discussed in section 8 of this paper. The specific features and syntax for the special `state` data type and transitions were not finalized at the time this paper was prepared for publication.

30. Assertions

The committee defining the SystemVerilog standard is currently evaluating a proposal to add assertions to the proposed SystemVerilog standard. This effort is well underway, but the specific features and syntax were not finalized at the time this paper was prepared for publication.

31. Current status of SystemVerilog

The standardization of the first generation of SystemVerilog is nearly complete, and is expected to be ratified by the Accellera board in July, 2002. Once approved, Accellera plans to donate SystemVerilog to the IEEE 1364 Verilog Standards Group, for incorporation into a future version of the IEEE 1364 Verilog standard.

32. Future plans for SystemVerilog

The first generation of SystemVerilog as presented in this paper is not the end of the road—it is the beginning. Accellera will continue to review the needs of Verilog design and verification. New features will be added to the SystemVerilog standard as they become well defined. These extensions will also be donated to the IEEE 1364 Verilog Standards Group.

33. 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 fully expected that the IEEE Verilog standards group will adopt the SystemVerilog extensions as part of the next generation of the IEEE 1364 Verilog standard.

34. References

[1] *“IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language”*, IEEE, Pascataway, 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, 2001.

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

35. 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 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 and PLI. Mr. Sutherland can be reached by e-mail at `stuart@sutherland-hdl.com`. Updated copies of this paper and presentation slides are available at `www.sutherland-hdl.com`.