A Proposal for a Standard Synthesizable Subset for SystemVerilog-2005: What the IEEE Failed to Define

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

Abstract

SystemVerilog adds hundreds of extensions to the Verilog language. Some of these extensions are intended to represent hardware behavior, and are synthesizable. Other extensions are intended for testbench programming or abstract system level modeling, and are not synthesizable. The IEEE 1800-2005 SystemVerilog standard[1] defines the syntax and simulation semantics of these extensions, but does not define which constructs are synthesizable, or the synthesis rules and semantics. This paper proposes a standard synthesis subset for SystemVerilog. The paper reflects discussions with several EDA companies, in order to accurately define a common synthesis subset that is portable across today's commercial synthesis compilers.

1. Introduction

SystemVerilog extensions to the Verilog HDL address two major engineering needs: to efficiently model designs of ever increasing complexity and size, and to effectively verify these complex designs. In each of these areas, SystemVerilog adds dozens of major constructs to Verilog, and hundreds of smaller, yet significant, extensions to existing Verilog constructs. This paper focusses on the hardware modeling extensions to Verilog. In a typical design flow, hardware models must be written so as to be compatible with both simulation and synthesis compilers (several other software tools might also be used in the design process, but these tools typically closely follow the same rules and restrictions imposed by simulation and synthesis).

The newly ratified IEEE 1800-2005 SystemVerilog standard[1] defines the syntax and semantics of the SystemVerilog extensions to Verilog for event based simulation. The standard does not define the syntax and semantic rules for synthesis compilers. This is different than the Verilog standard, where the IEEE defined syntax and semantic rules for both simulation and synthesis (IEEE 1364-2001[2] and IEEE 1364.1-2002[3].

The lack of a SystemVerilog synthesis standard has lead to frustrating disparity in commercial synthesis compilers. Each commercial synthesis product—HDL Compiler, Encounter RTL Compiler, Precision, Blast and Synplify, to name just a few—supports a different subset of SystemVerilog. Design engineers must experiment and determine for themselves what SystemVerilog constructs can safely be used for a specific design project and a specific mix of EDA tools. Valuable engineering time is lost because of the lack of a standard synthesizable subset definition for SystemVerilog.

This paper proposes a subset of the SystemVerilog design extensions that should be considered synthesizable using current RTL synthesis compiler technology. At Sutherland HDL, we use this SystemVerilog synthesis subset in the training and consulting services we provide. We have worked closely with several EDA companies to ensure that this subset is portable across a variety of synthesis compilers. The guidelines presented in this paper enable the engineers we work with to confidently take advantage of SystemVerilog, and the many benefits these powerful extensions have to offer.

Notes: This paper only presents the synthesizable SystemVerilog constructs. If a SystemVerilog construct is not listed in this paper, it should be considered to be generally non-synthesizable. Due to the 8 page limit for papers at this conference, it is not possible to specify the level of detail that a true SystemVerilog synthesis standard should contain. This paper only provides an overview of the synthesizable SystemVerilog constructs.

2. Shared declaration spaces

In the Verilog language, modules are self-contained design blocks. All data types, tasks and function used by a module must be declared locally within the module. If the same definition needs to be used in multiple modules, the definition must be repeated within each module. SystemVerilog extends Verilog by adding two new declaration spaces, which can be shared by any number of design and verification blocks, packages and **`$unit`**.

2.1 Packages

User-defined packages are defined between the keywords **package** and **endpackage**. The synthesizable items packages can contain are:

- **parameter** and **localparam** constant definitions
- **const** variable definitions
- **typedef** user-defined types (discussed in 5.1)
- Fully automatic **task** and **function** definitions
- **import** statements from other packages

The definitions within a package can be used within a design block (i.e.: a module or interface) in any of four ways, all of which are fully supported for synthesis:

- Explicit reference of a package item, using the package name and scope resolution operator. For example:

```
module alu
(input alu_pkg::opcode_t opcode, ...)
```

- Explicit import of a package item using an import statement. For example:

```
module alu (...);
import alu_pkg::packet_t;
```

- Implicit wildcard import of a package within a design block scope. For example:

```
module alu (...);
import alu_pkg::*;
```

- Explicit or implicit import of package items into the $unit declaration space (see 2.2)

2.2 $unit

SystemVerilog provides a special, built-in package called **$unit**. Any declaration outside of a named declaration space is defined in the $unit package. In the following example, the definition for `bool_t` is outside of the two modules, and therefore is in the $unit declaration space.

```
typedef enum bit {FALSE, TRUE} bool_t;

module alu (...);
  bool_t success_flag;
  ...
endmodule

module decoder (...);
  bool_t a_ok;
  ...
endmodule
```

$unit can contain the same kinds of user definitions as a named package, and has the same synthesis restrictions. **$unit** is automatically visible to all design (and verification) blocks that are compiled at the same time. Each invocation of the synthesis compiler creates a new **$unit** implicit package that is unique to that compilation. Declarations in the **$unit** created by one compilation are not visible in the **$unit** created by another compilation.

3. Variable data types

SystemVerilog extends the Verilog HDL variable types by adding 2-state types and C-like integer types.

3.1 Bit-vector variables

Verilog has the **reg** variable type, which can be defined to represent any vector size. SystemVerilog refers to Verilog bit-vectors and integer types as "*packed arrays*" (see section 6.1). SystemVerilog extends the Verilog bit-vector type with two new keywords:

logic — a 4-state variable with a user-defined vector width. **logic** is a synonym for the Verilog **reg** type. The two keywords are completely interchangable, with one exception that is noted in section 4 on net data types.

bit — a 2-state variable with a user-defined vector width. The **bit** type can be used any place a Verilog **reg** type can be used, but will never contain a logic Z or X value.

The **logic** and **bit** variable types are synthesizable, and follow the same rules for synthesizing **reg** variables.

3.2 Integer variables

Verilog has **integer** and **time** variable types, which represent 32-bit signed and 64-bit unsigned integer storage, respectively. SystemVerilog extends this category of variables with several new types:

byte — 2-state variable with a fixed vector width of 8 bits.

shortint — 2-state variable with a fixed width of 16 bits.

int — 2-state variable with a fixed width of 32 bits.

longint — 2-state variable with a fixed width of 32 bits.

These new variable types are synthesizable, and follow the same rules defined in the Verilog 1364.1-2002 standard[3] for synthesizing **integer** variables.

Note: In simulation, the **bit**, **byte**, **shortint**, **int** and **longint** variable types only store 2-state values. These types become simple interconnecting nets after synthesis, which can transfer 4-state values. This can lead to differences in RTL versus post-synthesis gate-level simulation results. This difference can also impact formal verification of pre-synthesis versus post-synthesis models.

3.3 Variable declarations (var)

SystemVerilog allows an optional **var** keyword to be specified before any variable type. The **var** keyword (short for "*variable*") documents that an object is a variable. The **var** keyword is synthesizable, with no restrictions.

```
var reg [7:0] a; // 4-state vector variable
```

This explicit documentation can help make code more readable and maintainable when variables are created from

user-defined types. For example:

```
typedef enum bit {FALSE, TRUE} bool_t;
var bool_t b; // variable of user-defined type

var struct {
  logic       tag;
  logic [7:0] data;
} c; // anonynous structure variable
```

A variable can also be declared using **var** without an explicit data type. In this case, the variable is assumed to be of type **logic**.

```
var [7:0] c; // 8-bit logic variable
```

4. Net Data Types

SystemVerilog extends Verilog net data types by allowing compound nets to be defined using either user-defined types or 4-state built-in types. SystemVerilog syntax requires that any user-defined type used with a net declaration must be made up of only 4-state types. The syntax is:

<net_type> <4-state_type> <net_declaration>
or **<net_type> <user_type> <net_declaration>**

Some examples are:

```
wire logic [7:0] w1; // 8-bit 4-state net

wire integer w2; // 32-bit 4-state net

typedef enum logic {FALSE, TRUE} bool_t;
wire bool_t w3; // net with bool_t values

typedef struct {reg a; integer b;} packet_t;
wire packet_t w4; // net with sub fields
```

User-defined net types are synthesizable following the same rules defined in 1364.1-2002 Verilog synthesis standard[3] for synthesizing net data types.

5. User-defined types

SystemVerilog allows designers to create new, user-defined data types. Both variables and nets can be declared as user-defined types. If neither the **var** or a net type keyword is specified, then user-defined types are assumed to be variables (**var**). SystemVerilog adds several user-defined type constructs for use in design and verification. Only type definitions (**typedef**), enumerated types, structures and unions are synthesizable.

5.1 Type definitions (typedef)

Designers can specify new data types that are constructed from built-in types and other user-defined types using **typedef**, similar to C. Two simple examples are:

```
typedef int unsigned uint_t;

typedef enum bit {FALSE, TRUE} bool_t;
```

5.2 Enumerated types

Enumerated types allow variables and nets to be defined with a specific set of named values. Only the synthesizable aspects of enumerated types are presented in this paper. The basic syntax for declaring an enumerated type is:

```
// a variable that has 3 legal values
enum {WAITE, LOAD, READY} State;
```

Enumerated types have a base data type, which by default is **int** (a 2-state, 32-bit type). In the example above, `State` is an **int** type, and `WAITE`, `LOAD` and `READY` will have 32-bit **int** values. Designers can specify an explicit base type, allowing enumerated types to more specifically model hardware. An example is:

```
// a 4-state, 2-bit enumerated variable
enum logic [1:0] {WAITE, LOAD, READY} State;
```

The named values in the enumerated list are constants that have an associated logic value. By default, the first label in the enumerated list has a logic value of 0, and each subsequent label is incremented by one. Thus, in the example above, `WAITE` is 0, `LOAD` is 1, and `READY` is 2.

Designers can specify explicit values for any or all labels in the enumerated list. For example:

```
// 2 enumerated variables with one-hot values
enum logic [2:0] {WAITE = 3'b001,
                  LOAD  = 3'b010,
                  READY = 3'b100} State;
```

Enumerated types can be named user-defined types, using **typedef**, or anonymous user-defined types. For example:

```
// a named enumerated type
typedef enum bit {FALSE, TRUE} bool_t;
bool_t done;

// an anyonymous enumerated variable
enum logic [1:0] {WAIT, LOAD, READY} State;
```

SystemVerilog also provides several methods for working with enumerated types. The synthesizable methods are: **.first**, **.last**, **.next**, **.prev** and **.num**.

5.3 Structures

SystemVerilog structures provide a mechanism to collect multiple variables together under a common name. Structures are synthesizable, provided the variable types used within the structure are synthesizable.

```
struct {  // anonymous structure variable
  logic [7:0] tag;
  int         a, b;
} packet_s;

typedef struct {           // typed structure
  logic [ 1:0] parity;
  logic [63:0] data_word;
} data_word_t;

data_word_t data;
```


The members of a structure can be referenced individually, or as a whole. The entire structure can be assigned using a list of values, enclosed in '{ }. The list can contain default values for one or more members of the structure.

```
data_word_t  data = '{3,0};
```

Structures can also be defined as "*packed*", indicating that the members of the structures must be stored as contiguous bits. Packed structures can only contain bit-vector and integer types. Packed structures are a bit vector, and can be used as a vector in operations. Bits of a packed structure can be referenced by member name, or by an index.

```
typedef struct packed {   // packed structure
  logic [ 1:0] parity;
  logic [63:0] data_word;
} data_word_t;

data_word_t  data_in, data_out, data_check;
assign data_check = data_in ^ data_out;
```

5.4 Unions

SystemVerilog unions allow a single storage space to represent multiple storage formats. SystemVerilog has three types of unions: a simple union, a packed union, and a tagged union. Only packed unions and tagged unions are synthesizable.

Packed unions require that all members within the union be packed types of the same number of bits. Packed types are bit-vectors (packed arrays), integer types, and packed structures. Because all members within a packed union are the same size, it is legal to write to one member (format) of the union, and read back the data from a different member.

```
typedef struct packed {     // TCP data packet
  logic [15:0] source_port;    // 64 packed bits
  logic [15:0] dest_port;
  logic [31:0] sequence;
} tcp_t;
typedef struct packed {     // UDP data packet
  logic [15:0] source_port;    // 64 packed bits
  logic [15:0] dest_port;
  logic [15:0] length;
  logic [15:0] checksum
} udp_t;
union packed {      // can store value as either
  tcp_t tcp_data;   //   packet type; can read
  udp_t udp_data;   //   value back as either
} data_packet_u;    //   packet type
```

Tagged unions allow each member within a union to be any size, including void (zero size).

```
union tagged {
  logic [23:0] short_instruction;
  logic [47:0] long_instruction;
} instruction_u;
```

Simulation must maintain an internal tag that monitors what member was used when a value is stored in the union, and check the tag when a value is read from the union. If a value is read not from the same member as the last write, a run-time simulation error occurs. This run-time checking ensures that, in simulation at least, the union is being used consistently.

Synthesis does not create the implied simulation tag in the hardware implementation. Synthesis compilers may, or may not, perform checking that the tagged union is written/ read in a consistent format.

5.5 Parameterized types

SystemVerilog extends Verilog parameter definitions, and redefinitions, to allow paramterizing data types. For example:

```
module adder #(parameter type dtype = int)
(input  dtype a, b,  output dtype sum);
  assign sum = a + b;
endmodule

module top;
  adder i1 (a, b, r1); // a 32-bit 2-state adder
  adder i2 #(dtype=logic[15:0]) (a, b, r2);
    // a 16 bit 4-state adder
endmodule
```

Parameterized data types are synthesizable.

6. Data arrays

SystemVerilog extends the Verilog static array construct in several ways that are synthesizable. (SystemVerilog also adds several types of dynamically sized arrays, which are not synthesizable).

6.1 Packed arrays

SystemVerilog refers to Verilog bit-vector and integer types as "*packed arrays*", indicating that these types are an array of bits, packed together contiguously.

In Verilog a bit-vector has a single dimension. Bits within the vector can be referenced using a bit-select operator.

```
reg [31:0] data;  // array of 32 contiguous bits
data[31] = 1'b1;  // select a single bit
```

SystemVerilog allows bit-vectors (packed arrays) to be declared with multiple dimensions.

```
logic [1:0][1:0][7:0] bus;  // array of 32
                            //  contiguous bits
```

A packed array with multiple dimensions is still a vector made up of contiguous bits. That vector, however, is now divided into subfields.

```
bus[1][0] = 8'hFF; // select single subfield
bus[1] = 16'b0;    // select multiple subfields
bus[1][1][7] = 1'b1;  // select a single bit
```


Multidimensional packed arrays and selections within multidimensional packed arrays are synthesizable.

6.2 Unpacked arrays

SystemVerilog refers to Verilog arrays as "*unpacked arrays*", indicating that each element of the array is a separate variable or net that need not be stored contiguously. The major enhancements to unpacked arrays that are synthesizable include:

- Arrays of user-defined types
- Copying arrays
- Assigning of literal values to arrays as a whole
- Assigning to slices of an array
- Passing arrays through module ports
- Passing arrays to tasks and functions
- Array query system functions: **`$dimensions`**, **`$left`**, **`$right`**, **`$low`**, **`$high`**, **`$increment`** and **`$size`**
- Array traversal **`foreach`** loop

7. Casting

SystemVerilog adds a cast operator to Verilog, **`'( )`**. There are three types of cast operations, all of which are synthesizable:

- Type casting, e.g.: `sum = int'(r * 3.1415);`
- Size casting, e.g.: `sum = 16'(a + 5);`
- Sign casting, e.g.: `s = signed'(a) + signed'(b);`

SystemVerilog also adds a dynamic cast system function, **`$cast`**, which is synthesizable.

8. Module ports

SystemVerilog relaxes the rules on Verilog module port declarations and the data types that can be passed through ports. The new port declaration rules that are synthesizable are:

- The internal data type connected to a module **`input`** port can be a variable type.
- Arrays and array slices can be passed through ports.
- Typed structures, typed unions and user-defined types can be passed through ports.

The following example illustrates a few of these synthesizable enhancements to module port declarations.

```
package user_types;
  typedef enum bit {FALSE, TRUE} bool_t;
endpackage

typedef struct {    // declared in $unit space
  logic [31:0] i0, i1;
  logic [ 7:0] opcode;
} instruction_t;

module ALU (output logic [63:0]        result,
            output user_types::bool_t  ok,
            input  instruction_t       IW,
            input  logic               clock);
```

Note: Synthesis compiler may—and probably will—mangle the port names of structure and array ports. At the time this paper was written, synthesis compilers might change structure and array ports into a single vector port made up of a concatenation of the members of the structure or array (possibly with unused bits optimized out). The example above might synthesize to:

```
module ALU (output [63:0] result,
            output        ok,
            input  [31:0] \IW.i0 ,
            input  [31:0] \IW.i1 ,
            input  [ 7:0] \IW.opcode ,
            input         clock);
```

Other synthesis compilers might expand structure and array ports to separate ports. The module port list for example above, after synthesis, might be:

```
module ALU (output [63:0] result,
            output        ok,
            input  [31:0] i0,
            input  [31:0] i1,
            input  [ 7:0] opcode,
            input         clock);
```

See section 16.1 for recommendations on the synthesis of compound ports.

9. Operators

SystemVerilog adds a number of operators that are synthesizable:

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

  These operators have the same synthesis restrictions as their counterpart non-assignment operators in the Verilog language.

```
a++;    // same as a = a + 1
b += a;    // same as b = b + a
for (int i; i<=7; i++)...  // same as i = i + 1
```

  NOTE: Synthesis compilers might restrict the use of assignment operations within an expression. Synthesis might also prohibit multiple assignments to the same variable in an expression

```
if ((--a))       // might not be synthesizable
  b = (a--);     // might not be synthesizable
if ((a=b))       // might not be synthesizable
  b = (a+=5);    // might not be synthesizable
i = i++;         // might not be synthesizable
i++ = 5;         // might not be synthesizable
```


- Wild equality/inequality operators: **==?** and **!=?**

  These operators allow excluding specific bits from a comparison, similar to the Verilog **casex** statement. The excluded bits are specified in the second operand using logic **X**, **Z** or **?**.

```
if (address ==? 16'hFF??)  // lower 8 ignored
```

  Wild card operators synthesize the same as **==** and **!=**, but with the some bits in the comparison masked out, following the same synthesis rules and restrictions as the Verilog **casex** statement

10. Procedural blocks

SystemVerilog enhances the Verilog **always** procedural block with three specialized procedural blocks. These specialized procedural blocks indicate the designer's intent for the type of hardware behavior that the procedural code should represent:

- **always_comb** — intent is to represent combinational logic functionality
- **always_latch** — intent is to represent latched logic functionality
- **always_ff** — intent is to represent sequential logic functionality

These specialized procedural blocks are synthesized the same as a standard **always** procedural block, and have the same synthesis restrictions as **always** procedural blocks. The specialized procedural blocks have several syntax and semantic rules that are different than an **always** procedural block, which can help ensure that RTL simulation and post-synthesis simulation have the same behavior.

These specialized procedural blocks do not force or guarantee that synthesis will generate the intended type logic. If the actual functionality within these specialized procedural blocks does not represent the intended functionality, then synthesis will generate logic that represents the actual functionality. In this case, the synthesis compiler will issue a warning that the actual functionality does not match the intent indicated by the type of procedural block.

11. Programming statements

SystemVerilog substantially extends the programming capabilities of Verilog. Many of these extensions are targeted towards testbench programming. The synthesizable enhancements to Verilog programming statements are:

- Variables on the left-hand side of continuous assignments
- One or more loop control variables declared as part of a **for** loop

```
for (int i=0; i<=255; i++) ...
for (int i=0; j=15; i<=15; i++; j--) ...
```

- **foreach** loop
- **do**...**while** bottom testing loop (for synthesis, the loop control must be able to be statically determined; the same synthesis rules as the Verilog **while** loop).
- **unique case** and **unique if** decision statements
- **priority case** and **priority if** decision statements
- **break**, **continue** and **return** jump statements
- Named ending statements (e.g.: **endmodule:** adder)
- Statement labels

The **unique** decision modifier directs synthesis to apply the same optimizations as the combined synthesis **full_case**/**parallel_case** pragmas, as defined in the IEEE 1364.1-2002[3]. The **priority** decision modifier directs synthesis to apply the same optimizations as the synthesis **full_case** pragma. The **unique** and **priority** modifiers also specify semantic rules for simulation, which, while not directly impacting synthesis, can reduce modeling errors that may lead to RTL simulation versus post-synthesis simulation mismatches.

12. Task and function enhancements

SystemVerilog enhances Verilog tasks and functions in several ways. The synthesizable enhancements are:

- Formal arguments have a default direction of **input**.
- Untyped formal arguments default to **logic**.
- Formal arguments can be any data type, including arrays, structures, unions and user-defined types.
- Functions can have **output** and **inout** formal arguments.
- Functions can be declared as **void** (no return value; called as a statement, like a task).
- Function return values can be specified using **return**.
- Multiple statements in a task or function do not need to be grouped using **begin**...**end**.
- Task/function calls can use named argument passing.
- Task/function calls can use **.**<name> and **.*** argument passing (see section 14).
- Mix of static and automatic storage.

13. Interfaces

SystemVerilog adds interface ports to Verilog. An interface port is a complex port type that can include data type declarations (both nets and variables), user-defined methods, and procedural code. The syntax, semantics, and capabilities of interfaces is a large topic that cannot be addressed in this paper, due to space limitations. In brief,

interfaces provide a means for designers to centralize the definition of a bus, as apposed to having the definition scattered in several modules throughout the design. Synthesis can then distribute the bus hardware appropriately throughout the design.

The synthesizable aspects of interfaces include, but are not limited to:

- Interface definition ports
- Interface data type declarations
- Interface modport definitions
- Interface tasks and functions (methods); must be fully automatic
- Interface procedural code; must follow synthesis rules
- Generic module interface ports (an instance of any interface can be connected to the port)
- Explicit module interface ports (only an instance of the explicitly named interface can be connected to the port)

NOTE: The IEEE has not defined rules for how synthesis should represent a module interface port in the synthesized module. Most, if not all, synthesis compilers will expand a module interface port to separate ports for each signal within the interface used by the module. See section 16.1 for recommendations on how module interface ports should be synthesized.

14. Module and interface instances

Verilog's explicit port connection syntax for instantiating a module can be verbose, with considerable repetitive typing. For example, connecting signals to an instance of a D-flip-flop might look like:

```
dff i1 (.q(q), .d(d), .clk(mclk), .rst(rst));
```

SystemVerilog provides two shortcuts for connecting signals to an instance of a module or interface.

The *dot-name* shortcut requires that when port names and signal names are the same, only the port needs to be explicitly named. The shortcut infers that a signal the same name as the port will be connected to that instance. The dot-name shortcut is synthesized the same as Verilog explicit port connections.

```
dff i1 (.q, .d, .clk(mclk), .rst);
```

The ***dot-star*** shortcut is a wildcard, that infers that all ports and signals of the same name are connected together.

```
dff i1 (.*, .clk(mclk));
```

The dot-star shortcut can be synthesized, but requires that both the module containing the instance and the port definitions of the module or interface being instantiated are compiled at the same time.

To facilitate separate file compilation with the *dot-star* shortcut, SystemVerilog provides a mechanism to prototype the module or interface being instantiated, using an **`extern`** statement. This allows synthesis compilers to see what the instantiated module's or interface's ports are, without having to compile the module or interface.

```
module top;
 extern module dff(output q, input d, clk, rst);
 dff i1 (.*, .clk(mclk));
endmodule
```

15. Keyword support

The following three tables summarize the synthesis support for SystemVerilog keywords.

Synthesizable SystemVerilog keywords:

always_comb	enum	priority
always_ff	extern	return
always_latch	foreach	shortint
bit	import	struct
break	int	tagged
byte	interface	typedef
const	logic	union
continue	longint	unique
do	modport	uwire
endinterface	package	var
endpackage	packed	void

SystemVerilog keywords that are ignored by synthesis:

assert	endsequence	sequence
assume	expect	timeprecision
endproperty	property	timeunit

Non synthesizable SystemVerilog keywords:

alias	export	rand
before	extends	randc
bind	final	randcase
bins	first_match	randsequence
binsof	forkjoin	ref
chandle	iff	shortreal
class	ignore_bins	solve
clocking	illegal_bins	static
constraint	inside	string
context	intersect	super
cover	join_any	this
covergroup	join_none	throughout
coverpoint	local	type
cross	matches	virtual
dist	new	wait_order
endclass	null	wildcard
endclocking	program	with
endgroup	protected	within
endprogram	pure	

16. Recommendations

16.1 Additional synthesizable constructs

At the time this paper was written, there were a few of SystemVerilog constructs that were not supported by most commercial synthesis tools that Sutherland HDL feels should be synthesizable. These include:

- Task/function **`ref`** ports
- Operator overloading
- Bounded queues and queue methods
- **`case`**...**`inside`** select statements
- **`case`**...**`matches`** select statements
- **`uwire`** single driver nets (defined in the IEEE 1364-2005 Verilog standard[4])

The new **`uwire`** data type ("*uwire*" is short for "*unresolved wire*") only permits a single source to drive a net. This can prevent modeling errors where single-source functionality is intended, but the same net name was inadvertently used more than once. This paper recommends that synthesis compilers maintain the **`uwire`** declaration in the resultant synthesized netlist. Thus, the single-source semantics are preserved in post-synthesis simulation.

16.2 Synthesis of compound ports

SystemVerilog allow modules to have complex, compound ports, in the form of arrays, structures, unions and interfaces. As noted in sections 8 and 13, synthesis compilers may change these compound ports to separate ports or a single port made up of a concatenation of multiple signals. This mangling of compound ports means that the synthesized module cannot be directly instantiated into a netlist or testbench in the same way as the original RTL module.

This paper recommends that synthesis compilers provide an option to generate a wrapper module around synthesized modules that have compound ports. The wrapper module should have the same compound ports as the original module, and contain an instance of the synthesized module, with the compound port members and elements appropriately connected to the ports of the module instance. The wrapper module can then be instantiated in the same way as the original module. Note that the wrapper module will add an additional level of hierarchy to the netlist. Hierarchical paths in verification code may need to be modified.

17. Summary

This paper has addressed a shortcoming in the IEEE 1800-2005 SystemVerilog standard[1]. The 1800-2005 standard defines the extensions to Verilog for simulation, but fails to define which constructs are synthesizable, and the synthesis rules for those constructs. This paper defines a synthesis subset for SystemVerilog that is already supported by a number of synthesis compilers (at the time this paper was written, some synthesis compilers did not support the entire subset, but have planned support for all of the subset defined in this paper).

Sutherland HDL uses this synthesis subset in the consulting and training services we provide. We hope that either the IEEE or Accellera will realize the importance of a well defined synthesis subset for SystemVerilog, and begin work on defining an official synthesis subset. Until then, it is hoped that design engineers will find the synthesis subset specified in this paper a useful guideline for writing synthesizable models that are portable to multiple synthesis compilers.

18. References

[1] *"1800-2005 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN: 0-7381-4811-3.

[2] *"1364-2001 IEEE Standard Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN:0-7381-2827-9.

[3] *"1364.1-2002 IEEE Standard for Verilog Register Transfer Level Synthesis"*, IEEE, Pascataway, New Jersey. Copyright 2002. ISBN:0-7381-3502-X.

[4] *"P1364 (D7) Draft Standard for Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN:0-7381-4770-2.

[5] *"SystemVerilog for Design Engineers"*, Stuart Sutherland, Simon Davidmann, and Peter Flake, Kluwer Academic Publishers, Boston, Massachusetts. Copyright 2004. ISBN: 1-4020-7530-8.

[6] *"SystemVerilog from a Synthesis Perspective"*, paper by Karen Pieper, Design and Verification Conference (DVCon), March 2004.

[7] *"SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes"*, paper by Stuart Sutherland, Synopsys Users Group (SNUG) conference, March 2005.

19. Contacting the author

Mr. Stuart Sutherland is a member of the IEEE 1800 SystemVerilog standards committee, and is the technical editor of the 1800 SystemVerilog Reference Manual. He is also a member of the IEEE 1364 Verilog Standards Group. Mr. Sutherland founded Sutherland HDL, Inc. in 1992. His company specializes in providing expert training on Verilog, SystemVerilog and the Verilog PLI. You can contact Mr. Sutherland at stuart@sutherland-hdl.com, or by calling 503-692-0898.