Modeling with SystemVerilog in a Synopsys Synthesis Design Flow Using Leda, VCS, Design Compiler and Formality

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

ABSTRACT

SystemVerilog is important to design engineers. It can significantly reduce the number of lines of RTL code needed to model complex hardware. This allows hardware designers to be more productive, and reduces the risk of coding errors. The challenge for designers, however, is determining which aspects of SystemVerilog are synthesizable and work with all tools used in a synthesis design flow. SystemVerilog adds hundreds of extensions to the Verilog language. Some of these extensions are intended to represent hardware behavior, while other extensions are intended for testbench programming or abstract system level modeling. This paper presents the synthesis subset of SystemVerilog currently supported by Synopsys EDA tools, specifically Leda, VCS, Design Compiler and Formality. The paper reflects discussions with internal Synopsys R&D groups, in order to accurately define a common synthesis subset that is portable across these key design and verification tools.

Table of Contents

1. Introduction

Using the SystemVerilog synthesis constructs enables engineers to model more functionality, reduce coding errors, and get successful silicon to market more quickly. There are important, and exciting reasons for why design engineers should use the SystemVerilog extensions to Verilog in their synthesizable designs. Some of these reasons are:

- A reduction in the amount of code that needs to be written, and thereby a reduction in the number of coding errors.
- The ability to represent complex functionality in concise, easier-to-read, easier-to-reuse RTL code.
- A dramatic reduction in the risk of mismatches in pre-synthesis RTL functionality and post-synthesis gate-level functionality.
- The ability to add white-box assertions to synthesizable code, without convoluted synthesis `translate_off` and `translate_on` pragmas
- A common language used throughout the design flow, including in synthesizable RTL models, test programs, bus-functional models and reference models.

The SystemVerilog extensions leverage proven modeling and verification constructs from several languages, including SUPERLOG, C++, SystemC, VHDL, VERA, OVA, OVL, PSL, and DirectC. The sum is greater than the individual parts. By integrating the best of all these languages into one language, design teams and verification teams can work with a single language, and software tools can execute the entire design, test, assertions, and coverage using a single kernel. The overhead of working with, and executing, multiple languages—some public and some proprietary—is eliminated. All of this integrated functionality is fully backward compatible with legacy Verilog code, and as such, can still be used in environments that currently use a mix of Verilog, VHDL and/or Verilog PLI applications. SystemVerilog also defines new APIs (Application Programming Interfaces) to make it much easier to integrate SystemVerilog models with SystemC models, and to extract and process functional coverage and assertion coverage information.

The SystemVerilog extensions to the Verilog HDL address two major engineering needs: 1) to efficiently model designs of ever increasing complexity and size, and 2) to effectively verify these complex designs. In each of these areas, SystemVerilog adds several dozen significant, extensions to Verilog. 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, as well as other software tools that might also be used in the design process.

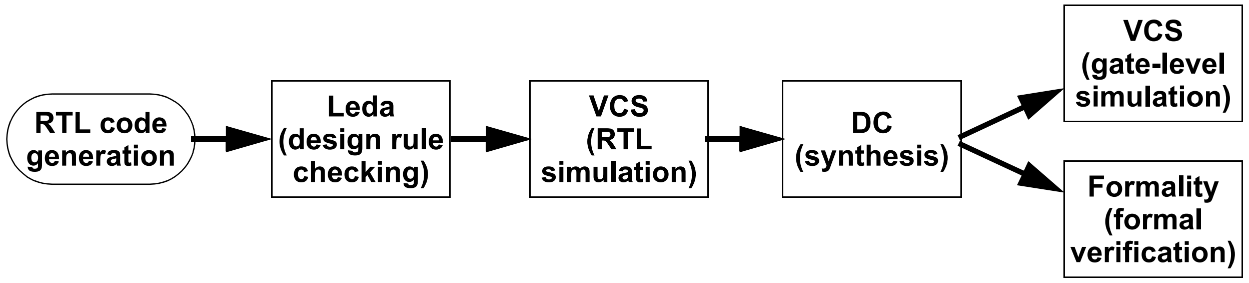
The Synopsys synthesizable subset is very rich and supports the most useful synthesizable constructs of SystemVerilog. Synopsys has done a good job of co-coordinating among products to ensure a common synthesis and verification subset that is supported throughout the implementation and verification flow. However, it is not always easy for end users of these products to find out just what that subset is. Even once the subset has been determined, there may still be uncertainties as to just how various Synopsys tools interpret those SystemVerilog constructs. For example, the SystemVerilog "**`priority case`**" is synthesizable. But, does it mean that VCS, DC and Formality will all evaluate the case statement in the same way? (The answer to this question is in Section 8.3 of this paper).

There are a large number of SystemVerilog constructs that are supported in the Synopsys synthesis design flow! Designers can—and should—be taking advantage of the subset listed in this paper!

This paper does not attempt to fully describe each SystemVerilog construct—that is the role of the IEEE 1800 SystemVerilog Language Reference Manual[1]. This paper only provides an overview of the synthesizable SystemVerilog constructs using current (at the time of writing) versions of the following Synopsys tools:

- ***Leda*** for design rule "lint" checking
- ***VCS*** for digital simulation
- ***Design Compiler (DC)*** for synthesis
- ***Formality*** for formal verification

A common synthesis design flow using these tools is:


RTL code generation
Leda (design rule checking)
VCS (RTL simulation)
DC (synthesis)
VCS (gate-level simulation)
Formality (formal verification)

In this tool flow, not all steps are necessary for all designs and design blocks. For example, it is common not to run gate-level simulations after synthesis on an entire design. The main point being made in this tool flow is to not put the proverbial "cart in front of the horse". While it is possible, for example, to run DC without having first used Leda and VCS, there are good reasons not to do so. Leda checks for, and helps enforce, a coding style that prevents common mismatches between pre-synthesis RTL functionality and post-synthesis gate-level functionality. Leda also helps ensure accurate equivalence checking between pre-synthesis RTL models and post-synthesis gate-level models. In addition, both Leda and VCS can see the "big picture" by reading in all the modules that make up the complete design, whereas a common usage of DC is to synthesize one design block at a time. DC cannot perform the same semantic checks on Verilog and SystemVerilog constructs that cross module boundaries.

Unless otherwise noted, the constructs presented in this paper are supported in released—or soon to be released—versions of these products at the time this paper was written (24 March 2006). This paper only presents the synthesizable synthesis constructs that are portable to all the Synopsys tools used in a typical synthesis design flow. Support for additional constructs may be added in future releases.

About the examples. The examples shown in this paper are not intended to show real-life design code. The examples are contrived to show specific SystemVerilog constructs in a small amount of code. All examples are synthesizable, but might not use an RTL style that will realize the most efficient hardware. Many of the earlier examples use SystemVerilog constructs that are not explained in detail until later in the paper. Verilog and SystemVerilog keywords are shown in bold text.

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 is used in multiple modules, the definition must be repeated within each module. SystemVerilog extends Verilog by adding a new declaration space, **$unit** (some Synopsys tools refer to **$unit** as **$root**, which was a name used in an early draft of the SystemVerilog standard).

$unit is a pseudo-global declaration space. Any declaration outside of a named declaration space falls in the **$unit** space. 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 logic {FALSE, TRUE} bool_t;   // defined in $unit space

module alu (...);
  bool_t success_flag;   // reference to definition in $unit space
  ...
endmodule

module decoder (...);
  bool_t a_ok;           // reference to definition in $unit space
  ...
endmodule
```

The synthesizable items that **$unit** can contain are:

- **parameter** and **localparam** constant definitions (discussed in Section 3.7)
- **const** variable definitions (discussed in Section 3.8)
- **typedef** user-defined types (discussed in Section 3.1)
- Fully automatic **task** and **function** definitions (discussed in Section 9)

File order dependencies. **$unit** is automatically visible to all design (and verification) blocks that are compiled at the same time. However, SystemVerilog requires that items defined in **$unit** must be declared before they are referenced in the design modules. When $unit declarations are defined in one design file, and referenced in a different design file, care must be taken to compile the files in the correct order.

Differences between VCS and DC: Each invocation of a compiler creates a new **$unit** declaration space that is unique to that compilation. Declarations in the **$unit** created by one compilation are not visible in the **$unit** created by another compilation. A common usage of VCS is to compile and simulate multiple design blocks together. In this usage, there is a single **$unit** declaration space which is shared by all design blocks. On the other hand, a common usage of DC is to synthesize each design block separately (or perhaps a small number of blocks at a time). In this usage, multiple **$unit** declaration spaces are created, which are not shared. Because of this difference, care must be taken as to where **$unit** declarations are made, so that both VCS and DC will see the same information.

Modeling guideline. Sutherland HDL uses the following modeling style to help ensure that both VCS and DC see the same declarations in **$unit**.

1. Place all definitions that are to be in the **$unit** declaration space in a single file, separate from the design files. At the top of this declarations file, add a conditional compilation check to prevent the definitions in the file from being read in more than once in the same compilation. The

file sets a **`define** macro definition as a flag that the file has been compiled: For example:

```
// In a file named "declarations.unit"
`ifndef DEFS_COMPILED   // check flag to see if already compiled
  `define DEFS_COMPILED    // flag that is set when this file is compiled
  typedef enum logic {FALSE, TRUE} bool_t;
  typedef struct {logic a; integer b;} packet_t;
`endif
```

2. At the beginning of each design file, add the following **`include** compiler directive:

```
`include "./declarations.unit"  // include $unit definitions
...
```

Note that in the example above the file containing the shared declarations does not end with ".v" (a common way to denote Verilog source code) or ".sv" (a common way to denote SystemVerilog source code). This is to help make it obvious that this file is not stand-alone code that can be compiled separately.

Using this modeling guideline, each file read in by Leda, VCS, DC, Formality or other design/verification tools will automatically include the exact same **$unit** definitions. Within the file containing the $unit definitions, conditional compilation is used to first check to see if the shared declarations already exist in the current **$unit** space. If not, the declarations are included in the compilation. If multiple files are compiled together (a common usage of Leda and VCS), only the first file read in will compile the declarations file into the **$unit** space; All subsequent files read in will see the DEFS_COMPILED flag, indicating that the definitions already exist in the current **$unit** space. If each file is compiled separately (a common usage of DC), each compilation will create a new **$unit** space, and the exact same definitions will be included in each **$unit** compilation space.

3. Enhancements to Verilog data types

SystemVerilog extends the synthesizable Verilog HDL data types in several significant ways.

3.1 Type definitions (typedef)

SystemVerilog allows designers to create user-defined data types using **typedef**, similar to C.

```
typedef integer unsigned uint_t;
uint_t u;          // a variable of type uint_t

typedef enum logic {FALSE, TRUE} bool_t;
bool_t A_OK;   // a variable of type bool_t

typedef struct {
  logic [3:0] check;
  logic [63:0] data;
} data_word_t;
data_word_t dword;   // variable of data_word_t type
```

3.2 New variable types

Verilog has the **reg** variable type, which can be used to create bit-vectors of any vector size. Verilog also has the **integer** and **time** types, which represent 32-bit signed and 64-bit unsigned integer storage, respectively. SystemVerilog extends the synthesizable of variables with several new types:

logic — a 4-state variable with a user-defined vector width. **logic** is a synonym for **reg**. The two keywords are interchangable.

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.

byte — 2-state signed variable with a fixed size of 8 bits

shortint — 2-state signed variable with a fixed size of 16 bits

int — 2-state signed variable with a fixed size of 32 bits

longint — 2-state signed variable with a fixed size of 64 bits

Some example declarations are:

```
logic reset;          // 1-bit 4-state unsigned variable; same as reg
logic [47:0] data;    // 48-bit 4-state unsigned variable
bit master_clock;     // 1-bit 2-state unsigned variable
int i;                // 32-bit 2-state signed variable
```

Potential RTL versus gate-level mismatch. Two-state data types within an RTL model will never have a logic Z or X value. If a value with Z or X bits is assigned to a 2-state type, those bits will map to 0. Synthesis does not preserve this 2-state semantics. In the gate-level models after synthesis, variables become a **wire** type, which will propagate Z or X values, rather than map the values to 0.

Modeling guideline. Sutherland HDL recommends in our training and consulting services to use 4-state types in synthesizable models. 4-state types will not hide design problems by masking X values, and will have the same simulation semantics as the post-synthesis net types. An exception to this guideline is the use of the 2-state **int** type as a for-loop control variable. Loop control variables generally disappear in post-synthesis gate-level models, and do not have a risk of RTL versus gate-level functionality mismatches.

3.3 Enumerated types

Enumerated types allow variables and nets to be defined with a specific set of named values. The basic syntax for declaring an enumerated type is:

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

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. When an enumerated type variable using the default base type is compiled, DC reports the warning:

```
Using default enum base size of 32
```

SystemVerilog goes beyond the abstract C- and VHDL-like declaration above. With SystemVerilog, design engineers can specify an explicit base type, allowing enumerated types to more accurately represent hardware. An example is:

```
enum logic [2:0] {WAITE, LOAD, READY} State;  // 4-state, 3-bit variable
```

The value labels in the enumerated list are constants that have an associated logic value. By default, the first label in the enumerated list has a 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.

SystemVerilog also allows design engineers to specify explicit values for any or all of the labels in the enumerated list, again more accurately representing real hardware. For example:

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

Enumerated types can be *named* types or *anonymous* types. Named types are defined using **`typedef`**.

```
typedef enum logic {FALSE, TRUE} bool_t;  // named type enumerated variable
bool_t done;

enum logic [1:0] {A, B, C} State;         // anonymous enumerated variable
```

Modeling guideline. Use typedef to define all enumerated variables. Software tools can perform stronger type checking on named type enumerated variables

Potential RTL versus gate-level mismatch. Enumerated types defined using the default base type and label values are abstract modeling constructs, which do not have the same semantics as hardware. The default base type is **`int`**, which is a 2-state type. DC will map this base type to simple wires (possibly also optimizing out unused bits), which are 4-state types with different simulation semantics. Also, at the RTL level, enumerated types document a legal set of values. After synthesis, an enumerated type is a simple net vector, which no longer has any special semantics that limit the legal set of values that can appear on that net. Another possible cause of RTL versus gate-level mismatches is the difference in the values used to represent the enumerated labels. If explicit values for each label are not specified, synthesis might change the values that represent the labels. The values of the labels will not match between pre-synthesis RTL functionality and post-synthesis gate-level functionality.

Modeling guideline. To reduce the risk of RTL versus gate-level functionality mismatches, all enumerated types used in models that will be synthesized should be explicitly defined as a **`logic`** type (4-state) and the values of all enumerated labels should be explicitly defined.

3.4 Structures

SystemVerilog structures provide a mechanism to collect multiple net and variable declarations together under a common name. Structures are synthesizable, provided the types used within the structure are synthesizable. Structures can be *named* types (using **`typedef`**) or *anonymous* types. For example:

```
struct {                    // anonymous structure variable
  opcode_t  opcode;
  int       a, b;
} alu_instruction_s;

typedef struct {            // named structure type
  logic [ 3:0] parity;
  logic [63:0] data_word;
} data_word_t;

data_word_t data;   // variable of data_word_t type
```

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. The following examples use the `data` structure variable declared above.

```
data.data_word = alu_out;  // assign to structure member

data = {15,0};             // assign list of values to entire structure
```

```
  data = {default:1};         // assign same value to all structure members
```

Note: The use of **{ }** to enclose a list of values was defined in the Accellera version of the SystemVerilog standard. The IEEE 1800 SystemVerilog standard deprecates this syntax, because it is too easily confused with the Verilog concatenation operator. The IEEE standard uses a slightly different syntax of **'{ }** for value lists. The older syntax, is the style that is portable across tools in a Synopsys synthesis design flow.

Note: Value lists will not be supported in the full Synopsys synthesis design flow until later in 2006.

Unpacked and packed structures. By default, the members of a structure are "*unpacked*", which allows software tools to store structure members as independent objects. Structures can be explicitly defined as "*packed*", indicating that the members of the structures must be stored as contiguous bits. 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 -- 64 bits
    logic [15:0] source_port;
    logic [15:0] destination_port;
    logic [31:0] data_sequence;
  } tcp_t;

  tcp_t  data_in, data_out, data_check;    // structure variables

  assign data_check = data_in ^ data_out;  // operation on packed structures
```

Modeling guideline. Structures should not be declared as packed unless necessary. When a structure is packed, software tools cannot perform as much type checking as can be performed on each member separately. The appropriate usage of packed structures is when a structure needs be used as a whole in operations or the structure will be used in a union (see Section 3.5).

Note: DC changes a structure port to a single port made up of a concatenation of the structure members. Section 5, on module ports, discusses this in more details.

3.5 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 are synthesizable in the Synopsys tool flow (simple unions and tagged unions are used in testbench and abstract modeling). Packed unions require that all members within the union be packed types of the same number of bits. Packed types are bit-vectors (also known as 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 (format).

The following example illustrates the use of structures and unions. The example is a complete model of a optical network data register that uses the UNI/NNI switch protocol. A union is used to represent the storage of a cell, which can be in either a UNI cell, or an NNI cell. The UNI and NNI cells are modeled as packed structures that are 424 bits wide. The register can store 32 data cells. Each register location is represented as a structure that contains the packet union plus a flag bit that indicates which type of packet is stored in the union. The entire register is passed back out of the model as an array of structures.

```
  // $unit space declarations
  `ifndef DEFS_COMPILED      // check flag to see if already compiled
    `define DEFS_COMPILED    // flag that is set when this file is compiled

    typedef struct packed {      // UNI cell structure definition
      logic        [ 3:0] GFC;
      logic        [ 7:0] VPI;
      logic        [15:0] VCI;
      logic               CLP;
      logic        [ 2:0] T;
      logic        [ 7:0] HEC;
      logic [0:47] [ 7:0] Payload;  // packed array of 48 8-bit data words
    } uni_t;

    typedef struct packed {     // NNI cell structure definition
      logic        [11:0] VPI;
      logic        [15:0] VCI;
      logic               CLP;
      logic        [ 2:0] PT;
      logic        [ 7:0] HEC;
      logic [0:47] [ 7:0] Payload;  // packed array of 48 8-bit data words
    } nni_t;

    typedef union packed {      // union to store either UNI cell or NNI cell
      uni_t  uni_cell;
      nni_t  nni_cell;
    } cell_t;

    // Flag for what packet type is loaded in the packet register
    typedef enum logic {is_uni=1'b0, is_nni=1'b1} packet_type_t;

    typedef struct {         // structure to store a packet union plus a flag
      packet_type_t  packet_type;
      cell_t         data_cell;
    packet_t;
  `endif
  // End of $unit declaration space

  module packet_register
  (input  logic         clk,
   input  logic         reset,
   input  logic         load_en,
   input  packet_type_t packet_type,      // enumerated type as input port
   input  cell_t        data_cell,        // union type as input port
   input  logic [4:0]   write_pointer,
   output packet_t      packet_reg [0:31] // array of structures output port
  );

    // Write inputs into the register at address of write_pointer
    always_ff @(posedge clk or posedge reset)
      if (reset)
        packet_reg = {default:0};  // load entire array with 0
      else if (load_en) begin
        packet_reg[write_pointer].packet_type <= packet_type;
        if (packet_type == is_uni)
          packet_reg[write_pointer].data_cell.uni_cell <= data_cell.uni_cell;
        else
          packet_reg[write_pointer].data_cell.nni_cell <= data_cell.nni_cell;
      end
  endmodule
```

3.6 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'(a * 16'h3);`
- Size casting, e.g.: `sum = 16'(a + 16'h3);`
- Sign casting, e.g.: `sum = signed'(a) + signed'(b);`

One application of type casting is to assign the result of an operation to an enumerated type variable. Enumerated types are semi-strongly typed; only a value of the same enumerated type can be assigned to an enumerated variable. For example:

```
typedef enum logic [1:0] {s0=2'b00, s1=2'b01, s2=2'b10, s3=2'b11} states_t;
  states_t state, next_state;

always_latch
  if (enable)
    next_state = states_t'(state + 1);
```

Note: Casting can force an out-of-range value into an enumerated variable. The design code needs to prevent illegal conditions, or check that illegal conditions never occur (a good usage of SystemVerilog assertions). In the example above, the enumerated variable is defined as 2 bits wide, which can hold values from 0 to 3. Enumerated labels are defined for all four possible values. When state has a value of 3, incrementing that value will overflow and wrap back to a value of 0, which is a legal in-range value.

3.7 Parameter and localparam constants

SystemVerilog extends Verilog by allowing parameters and localparams to be declared in the shared declaration spaces, **`$unit`** and packages. Parameters declared in these shared declaration spaces cannot be redefined, making them equivalent to localparams.

SystemVerilog also extends Verilog parameter definitions to allow parametrizing data types. For example:

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

module top;
  int          a1, b1, r1;
  logic [15:0] a2, b2, r2;

  adder i1 (a1, b1, r1);                         // a 32-bit 2-state adder
  adder #(.dtype(logic[15:0])) i2 (a2, b2, r2); // a 16 bit 4-state adder
endmodule
```

3.8 const constants

SystemVerilog adds a **`const`** constant, which allows declaring any variable type as read-only. A **`const`** variable can be initialized to a value at the time of declaration, but cannot be changed afterwards. The primary usage of **`const`** constants is in dynamic testbench programs and abstract, non-synthesizable models. For synthesis purposes, a **`const`** constant is treated the same as a Verilog **`localparam`** constant. The main difference in synthesizable code is that a **`const`** constant can also be declared in automatic functions, where localparams are not allowed.

Note: The `const` construct will be fully supported in the Synopsys synthesis design flow later in 2006.

4. 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).

4.1 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 sub fields.

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

Multidimensional packed arrays are synthesizable. Declaring sub fields within a array is just for RTL modeling convenience. A packed array with multiple dimensions is still just a vector, and synthesizes to the same gate-level implementation as a Verilog one-dimensional vector.

The example listed in Section 3.5 illustrates using a packed array to represent a UNI or NNI payload vector that is 48 8-bit data words.

4.2 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 synthesizable enhancements to unpacked arrays are:

- Arrays of user-defined types
- Copying arrays
- Assigning a list of values to arrays as a whole (same syntax as assigning to unpacked structures)
- Assign a default value to an entire array
- Assigning to slices of an array
- Passing arrays through module ports (see Section 5)
- Passing arrays to tasks and functions (see Section 9)
- Array query system functions: **`$dimensions`**, **`$left`**, **`$right`**, **`$low`**, **`$high`**, **`$increment`** and **`$size`**

The example listed in Section 3.5 illustrates using an unpacked array of user-defined types to represent a register that can contain 32 cells, containing either UNI or NNI data cells. This example also illustrated passing the complete array through a module port.

```
  output packet_t      packet_reg [0:31] // array of structures output port
```

Note: DC changes an unpacked array port to a single port made up of a concatenation of multiple signals. Section 5 discusses this in more details.

5. 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 net or variable type (Verilog only permitted net types)
- User-defined types can be passed through ports
- Enumerated types can be passed through ports
- Typed structures can be passed through ports
- Typed unions can be passed through ports
- Unpacked arrays and array slices can be passed through ports
- Ports can be defined as "*interface ports*" (see Section 10)

The example listed in Section 3.5 illustrates using a few of these synthesizable enhancements to module port declarations. The module header portion is repeated below. Refer to the full example for the user-defined type declarations used in the module ports.

```
module packet_register
(input  logic          clk,
 input  logic          reset,
 input  logic          load_en,
 input  packet_type_t  packet_type,      // enumerated type as input port
 input  cell_t         data_cell,        // union type as input port
 input  logic [4:0]    write_pointer,
 output packet_t       packet_reg [0:31] // array of structures output port
);
```

Potential RTL versus gate-level mismatch. DC will change the ports for structure, union and array ports. Synthesis flattens these compound ports into a single vector port made up of a concatenation of the members of the structure, union or array (possibly with unused bits optimized out). This flattening of compound ports means that the gate-level module is not plug-and-play compatible with an instance of the original RTL module in a testbench or design netlist. In order to simulate the post-synthesis gate-level module, the port connections of the module instance must be changed to match the flattened ports of the gate-level module.

New port mapping option. Synopsys is adding an option to DC (to be available in the planned 2006.06 release) to generate a wrapper module around synthesized modules that have compound ports. The wrapper module has the same ports as the original RTL module, and contains an instance of the synthesized module with the compound port members appropriately connected to the ports of the gate-level module instance. The wrapper module can then be instantiated in the same way as the original RTL module. Note that the wrapper module will add an additional level of hierarchy to the netlist, which may affect a testbench uses hierarchical paths to access signals in the synthesized module.

6. Operators

SystemVerilog adds a number of operators to Verilog. The synthesizable operators in the Synopsys synthesis design flow are:

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

Increment/decrement operators: Both pre- and post-increment/decrement operations are supported. Some examples are:

```
for (int i=0; i<=7; i++) begin  // i++ is same as i = i + 1
  ...
end

while (a) begin
  ...
  a--;                          // decrement operation as a statement
end
```

Assignment operators: These operators perform an operation involving two operands and assign the result to the left-hand side operand.

```
case (opcode)
  ADD : a += b;      // same as a = a + b
  DIV2: a >>= 1;     // same as a = a >> 1
...
endcase
```

Synthesis restrictions. DC restricts the use of the increment/decrement and assignment operations to simple statements. The operators cannot be used in a compound statement, where more than one operation is performed. The following examples are syntactically legal, but are not synthesizable:

```
a = i++;        // NOT SYNTHESIZABLE: same as i = i + 1; a = i;
if ((--a))      // NOT SYNTHESIZABLE: same as a = a - 1; if (a) ...
b = (a+=5);     // NOT SYNTHESIZABLE: same as a = a + 5; b = a;
```

Reduced potential RTL versus gate-level mismatch. The synthesis restrictions on the increment/decrement operators and assignment operators prevent using the operators in ways that can result in ambiguous RTL simulation results, due to simulation event order race conditions. This ambiguity could lead to RTL functionality versus gate-level functionality mismatches. The synthesis restrictions eliminate this cause of mismatches.

7. Procedural blocks

The SystemVerilog enhancements to Verilog procedural blocks are simple, but very important for reducing mismatches between pre-synthesis RTL functionality and post-synthesis gate-level functionality. 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` indicates that the intent is to model combinatorial functionality. **`always_comb`** automatically infers a complete combinatorial logic sensitivity list. It is illegal to specify a sensitivity list with these constructs (incorrect user-specified sensitivity lists are a frequent cause of RTL versus gate-level functionality mismatches). An example combinatorial block is:

```
always_comb begin       // infers @(sel, a, b)
  if (sel) y = a;
  else     y = b;
```

```
end
```

always_latch indicates that the intent is to model latched functionality. **always_latch** also automatically infers a complete combinatorial logic sensitivity list.

```
always_latch           // infers @(enable, state)
  if (enable)
    next_state <= states_t'(state + 1);
```

always_ff indicates that the intent is to model sequential logic (flip-flop) functionality. With **always_ff**, the sensitivity list must be specified. The list cannot be inferred because clock name and synchronous versus asynchronous reset cannot be inferred from the procedural block code. **always_ff** requires that all signals listed in the sensitivity list be qualified with **posedge** or **negedge** qualifiers.

```
always_ff @(posedge clock or negedge resetN) begin
  if (!resetN) q <= 0;
  else         q <= d;
end
```

These specialized procedural blocks have additional semantic restrictions that help ensure the procedural blocks synthesize correctly. In brief, these restrictions are:

- Variables assigned by these any of procedural blocks cannot be assigned by any other procedural block.
- No time controls are permitted within any of these procedural blocks (i.e.: the @, # and **wait** tokens).
- No statements are permitted within any of these procedural blocks that might block the flow of execution of execution (e.g. a call to a task contains a time control).
- **always_comb** and **always_latch** execute once at simulation time zero even if no events trigger the inferred sensitivity list. This ensures that at time zero, the output values correctly reflect the input values to that block.

These specialized procedural blocks do not force or guarantee that synthesis will generate the intended type of logic. If the actual functionality of the procedural code does not represent the intended functionality, then synthesis will generate logic that represents the actual functionality. In this case, DC will issue a warning that the actual functionality does not match the intent indicated by the type of procedural block. The following example specifies an intent to model combinatorial logic, but the functionality requires latched behavior.

```
always_comb begin    // combinational logic intended
  if (enable)
    q_out <= d_in;   // transparent latch behavior
end
```

Leda and DC generate warnings that the functionality does not match the specified intent. VCS does not perform this additional checking. The warning generated by DC for this example is.

```
Warning: Netlist for always_comb block contains a latch (ELAB-974)
```

8. Programming statements

SystemVerilog adds several synthesizable extensions to the programming capabilities of Verilog.

8.1 Variables on the left-hand side of continuous assignments

Verilog restricts the data types used on the left-hand side of continuous to only net data types. SystemVerilog also allows any variable type on the left-hand side of continuous assignments, as long as the variable does not receive a value from any other source (another continuous assignment, module ports, procedural block assignments or in-line initialization assignments). DC further restricts the SystemVerilog rules to only allowing bit-vector types (**`reg`**, **`logic`** and **`bit`**), integer types (**`byte`**, **`shortint`**, **`int`**, **`integer`** and **`longint`**) and packed structures on the left-hand side of continuous assignments.

```
module adder
(input  logic [31:0] a, b,
 output logic [31:0] sum
);
  assign sum = a + b;
endmodule
```

One obvious advantage of allowing variables on the left-hand side of continuous assignments is that the designer no longer has to worry about when to use variables in RTL code, and when to use nets. Moreover, if the modeling style changes from continuous assignments to procedural blocks, or vice versa, the data type does not need to be changed. Very simply, variables can be used everywhere, except when multi-source logic is required (such as with a shared data bus).

Using variables on the left-hand side of continuous assignments can also eliminate a common cause of design errors when modeling at the RTL level in Verilog. When net types are used, an unintentional second source putting values on the net is not a syntax error; it is a functional problem that must be detected and debugged. When a variable type is used on the left-hand side of continuous assignments, unintentional multiple sources are a syntax error which tools can detect at compile/elaboration time.

Potential RTL versus gate-level mismatch. In a typical simulation with VCS, many modules are compiled together. VCS can look across multiple modules to determine if a variable used on the left-hand side of a continuous assignment is also written to by some other source. On the other hand, designs are typically synthesized with DC one module, or possible a few modules, at a time. This means DC cannot see as much of the design as VCS to determine if a variable is written to by multiple sources.

Modeling guideline. Before synthesizing a design block with DC, check the design block along with the blocks to which it is connected using Leda or VCS. These tools can more easily detect inadvertent errors of multiple source logic that span multiple blocks of a design.

8.2 Enhanced for loop declarations

In Verilog, the control variable used in **`for`** loops must be declared outside of the **`for`** loop. The variable is then visible throughout the scope in which it is declared, which can lead to misuse of the variable. SystemVerilog allows the **`for`** loop control variable to be declared as part of the **`for`** loop declaration. As a local variable, the loop control variable is not visible outside of the loop. Local **`for`** loop variables synthesize the same as externally declared variables.

```
for (int i=0; i<=255; i++) begin   // local for-loop control variable
  ...
end
```

SystemVerilog also allows more than one loop control variable to declared as part of a **`for`** loop, and multiple step assignments to be executed at the end of each pass of the loop. Multiple declarations and multiple step assignments are separated by commas.

```
for (int i=0, j=i+OFFSET; i<=15; i++, j++) begin  // 2 for-loop variables
  ...
end
```

Note: Multiple locally declared **`for`** loop variables will be fully supported in the Synopsys synthesis design flow later in 2006.

8.3 unique and priority decision modifiers

SystemVerilog adds two decision modifiers to Verilog, **`unique`** and **`priority`**. These modifiers can be used on all forms of case statements (**`case`**, **`casez`** and **`casex`**) as well as **`if...else`** statements. The full details of the simulation checking performed by—and the importance of—**`unique`** and **`priority`** decisions is beyond the scope of this paper. The semantics of these decision modifiers are only briefly described here. Papers presented at previous SNUG conferences provide much more detail on these decision modifiers (see [7] and [8] in the References section at the end of this paper).

The SystemVerilog **`unique`** decision modifier adds run-time checking to **`if...else`** and **`case`**, **`casez`** and **`casex`** decision statements. A **`unique`** decision requires that simulation report a run-time warning if simulation detects that when a decision is entered, either more than one branch selection control of the decision could be true at the same time, or no branch selection control of the decision statement is true.

The **`unique`** modifier also states that tools *may* evaluate the decision selection controls in any order. VCS does not change the order from that which is listed in the RTL source code. For synthesis, DC will check the decision selection controls, and:

- If DC determines that the control values are mutually exclusive, it will create gate-level logic that evaluates the selection values in parallel, instead of in the order listed in the RTL code.
- If DC determines that the control values are not mutually exclusive, it will issue a warning that the decision select controls do not meet the designer's intent of being unique, and maintain the priority that is shown in the RTL code.
- If DC cannot determine whether or not the control values are mutually exclusive, it will create gate-level code per what the designer specified in the RTL code. If unique is specified, then the selection values will be evaluated in parallel.

DC will perform the same checking and optimizations on a **`unique case`** (or **`casez`** or **`casex`**) as with a **`case`** statement that has the combined **`full_case`**/**`parallel_case`** synthesis pragmas. DC will also apply these checks and optimizations for a **`unique if`** decision. There is no equivalent to this using synthesis pragmas.

```
always_comb begin
  unique if (select == 2'b00) y = a;
    else if (select == 2'b01) y = b;
    else if (select == 2'b10) y = c;
end
```

In simulation, should a select value of 2'b11 occur, VCS will report the following warning:

```
RT Warning: No condition matches in 'unique if' statement.
       "ex20_unique-priority.sv", line 20, at time 30
```

DC reports the following warning when this example is illustrated:

```
Warning: If statement marked unique does not cover all possible conditions.
(VER-506)
```

The **`priority`** decision modifier also adds run-time checking to **`if...else`** and **`case`**, **`casez`** and **`casex`** decision statements. Like **`unique`** decisions, a **`priority`** decision requires that simulation report a run-time error if simulation detects that when a decision is entered, no branch selection control of the decision statement is true. DC treats a **`priority case`** (or **`casez`** or **`casex`**) statement as a "full case" statement, and performs the same checking and optimizations as with a **`case`** statement that has just the **`full_case`** synthesis pragma. DC will also apply these checks and optimizations for a **`priority if`** decision. There is no equivalent to this using synthesis pragmas.

The following example illustrates an interrupt handler. More than one interrupt can occur at a time, and priority is given to the lowest numbered interrupt. A **`priority case`** statement is used to document that the designer intends the case statement to evaluated in the order the case select expressions are listed.

```
always_comb begin
  // set outputs to defaults for when there is no interrupt
  priority casez (IRQ)
    4'b???1: begin   // test if IRQ bit 0 is set, ignore other bits
               // process interrupt 0
             end
    4'b??1?: begin   // test if IRQ bit 1 is set, ignore other bits
               // process interrupt 1
             end
    4'b?1??: begin   // test if IRQ bit 2 is set, ignore other bits
               // process interrupt 2
             end
    4'b1???: begin   // test if IRQ bit 3 is set, ignore other bits
               // process interrupt 3
             end
  endcase
end
```

For this example, DC reports the following messages when the design is elaborated:

```
Warning: Case statement marked priority does not cover all possible
conditions. (VER-504)
Warning: Case statement is not a full case. (ELAB-909)

Statistics for case statements in always block at line 32...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            35            |     user/no      |
===============================================
```

Potential RTL versus gate-level mismatch. The name of the **`priority`** modifier would seem to suggest that tools *must* evaluate the decision selection controls in the order listed in the RTL source code. ***DC might not maintain the selection priority shown in the RTL code.*** DC will check the decision selection controls, and if it determines that the control values are mutually exclusive, will optimize the gate-level logic to evaluate the selection values in parallel, instead of in the order listed in the RTL code. This means the evaluation order of the RTL code and the post-synthesis gate-level logic might not be equivalent.

The next example illustrates an interrupt handler that expects that only one interrupt can occur at a time. Priority is given to the lowest numbered interrupt, which is documented using a **`priority case`** statement. In simulation, should multiple interrupt requests occur at the same time, the priority case will cause a warning to be issued that no case item expression was true.

```
always_comb begin
  // set outputs to defaults for when there is no interrupt
  priority case (IRQ)
    4'b0001: begin    // test if IRQ bit 0 is set, ignore other bits
               // process interrupt 0
             end
    4'b0010: begin    // test if IRQ bit 1 is set, ignore other bits
```

```
              // process interrupt 1
            end
    4'b0100: begin    // test if IRQ bit 2 is set, ignore other bits
              // process interrupt 2
            end
    4'b1000: begin    // test if IRQ bit 3 is set, ignore other bits
              // process interrupt 3
            end
  endcase
end
```

In this example, DC detects that the case expression items are mutually exclusive, and might not maintain the priority that is specified in the RTL code. The report generated by DC for this example is:

```
Warning: Case statement marked priority does not cover all possible
conditions. (VER-504)
Warning: Case statement is not a full case. (ELAB-909)

Statistics for case statements in always block at line 62...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            65            |    user/auto     |
===============================================
```

For the example above, explicit synthesis constraints must be specified to force DC to maintain the case statement priority.

8.4 break, continue and return jump statements

SystemVerilog adds two loop jump statement to Verilog:

- **`continue`** — jump to the end of loop, and continue execution of the loop in accordance with the loop controls
- **`break`** — jump out of a loop prematurely, without evaluating the loop controls

These jump statements can be used in place of the Verilog **`disable`** statement to control the execution of **`for`**, **`repeat`**, **`while`** and **`do...while`** loops. The **`break`** and **`continue`** jump statements are more C-like, and are more intuitive than the Verilog **`disable`** statement.

```
// find first bit set within a range of bits
always_comb begin
  first_bit = 0;
  for (int i=0; i<=63; i=i+1) begin
    if (i < start_range) continue;   // skip rest of this pass of loop
      if (i > end_range) break;      // exit loop
      if ( data[i] ) begin
        first_bit = i;
        break;                       // exit loop
      end
  end // end of the loop
    //... // process data based on first bit set
end
```

SystemVerilog also adds the C-like **`return`** statement to Verilog. The **`return`** statement serves two purposes:

- To specify the return value of a function (instead of Verilog's Pascal-like style of assigning the return value to the name of the function)
- To exit a task or function prematurely, before reaching the end of the task or function.

9. Task and function enhancements

SystemVerilog enhances Verilog tasks and functions in several ways. Many of the enhancements help write RTL code in a simpler, more intuitive style than with the Verilog HDL. The enhancements to tasks and functions that can be used in a Synopsys synthesis design flow are:

- Formal arguments have a default direction of **`input`**
- Function return values can be specified using **`return`**
- Multiple statements in a task or function do not need to be grouped using **`begin`**...**`end`**
- 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`**

The ability to pass any data type in and out of tasks and functions significantly extends the capabilities of these constructs. Being able to specify functions with output arguments as well as a return value further extends modeling capabilities, especially when using **`void`** functions. The declaration of **`void`** functions is very useful in synthesizable RTL models. Functions must execute in zero simulation time and cannot call other tasks. These restrictions help ensure that pre-synthesis RTL functionality and post-synthesis gate-level functionality match. A **`void`** function is called in the same way as a task, but has the language restrictions—and synthesis benefits—of a function.

```
typedef struct {
  logic valid;
  logic [ 7:0] check;
  logic [63:0] data;
} packet_t;

module packet_register
(input  logic        clock, reset,
 input  logic [63:0] data,
 output packet_t     packet);

  function void fill_packet (        // void function
    input  logic [63:0] data_in,
    output packet_t data_out );

    data_out.data = data_in;
    for (int i=0; i<=7; i++)
       data_out.check[i] = ^data_in[(8*i)+:8];
    data_out.valid = 1;
  endfunction

  always @(posedge clock)
    if (reset)
      fill_packet (0, packet);       // call void function
    else
      fill_packet (data, packet);
endmodule
```

10. 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. In brief, interfaces provide a means for designers to centralize the definition of a bus, as opposed to having the definition scattered throughout several modules in a design. This simplifies the design engineer's work at the RTL level, and lets synthesis do the work of distributing the gate-level bus hardware appropriately throughout the design.

The synthesizable aspects of interfaces include:

- Interface definitions
- Interface definition ports
- Interface data type declarations
- Interface **`modport`** definitions
- Interface tasks and functions (methods)
- Importing tasks and functions in modport definitions
- 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)

The Synopsys synthesis design flow imposes the following restrictions on the definition and usage of interfaces.

- An interface definition must be read by DC in before it is referenced as a module port
- In a netlist, interface instances must be instantiated before they are used
- Arrays of interface instances cannot be used
- When an interface method (a task or function) is imported to a module, the task or function be declared as **`automatic`**

The syntax, semantics, and capabilities of interfaces is a large topic that cannot be fully addressed in this paper. For the same reason, an example interface and usage of an interface is not included here. Readers are encouraged to refer to other SNUG papers that explain SystemVerilog interfaces and provide extensive examples (see [9], [10] and [11] in the References section of this paper).

How interface methods are synthesized. Synthesis will duplicate the logic of an imported interface method in each module that calls that method. This means that, after synthesis, each module using a method has its own copy of the method functionality, which replaces the shared RTL version. To prevent RTL versus gate-level functionality mismatches, the Synopsys synthesis design flow requires that interface tasks and functions that are called from modules must be declared as **`automatic`**. Automatic tasks and functions create new storage for each call to the task or function. When multiple calls are made to an automatic task or function, each module sees a unique copy of the interface method storage, even though at the RTL level there is only the one definition of the method. This ensures that RTL simulation behavior of a method matches the gate-level representation of the method functionality created by synthesis.

Potential RTL versus gate-level mismatch. DC flattens an interface port into separate ports, representing the collection of ports specified in the interface. This changing of the interface port means that the synthesized module cannot be directly instantiated into a netlist or testbench in the same way as the original RTL module. The module instantiation must be modified to match the post-synthesis gate-level module port list.

New port mapping option. Synopsys is adding an option to DC (to be available in the planned 2006.06 release) to generate a wrapper module around synthesized modules that have compound ports. The wrapper module has the same ports as the original RTL module, and contains an instance of the synthesized module, with the compound port members appropriately connected to the ports of the gate-level module instance. The wrapper module can then be instantiated in the same way as the original RTL module. Note that the wrapper module will add an additional level of hierarchy to the netlist, which may affect a testbench uses hierarchical paths to access signals in the synthesized module.

11. 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));   // qb not used
```

SystemVerilog provides two shortcuts for connecting signals to an instance of a module or interface, referred to as "*dot-name*" and "*dot-star*".

With the ***dot-name*** shortcut, when port names and signal names are the same, only the port needs to be named. The shortcut infers that a local signal the same name as the port will be connected to that instance.

```
dff i2 (.q, .d, .clk(mclk), .rst);   // qb not used
```

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

```
dff i3 (.*, .clk(mclk), .qb());   // qb not used
```

DC compilation. A module that contains a module instance using the *dot-name* shortcut can be read in by DC separately from the module being instantiated. The port names of the module being instantiated are explicitly named, which provides DC the information needed to synthesize the module instance. A module that contains a module instance using the *dot-star* shortcut *cannot* be read in by DC separately from the module being instantiated. Since .* does not provide the port names of the instantiated module, DC must read in both the module instance and the instantiated module's port definitions.

12. Assertions

SystemVerilog adds a powerful assertion language to Verilog. Assertions can, and should, be specified as part of the RTL. The usage of assertions by design engineers is discussed in the SNUG papers "*Assertions Are For Design Engineers, Too!*" [13] and "*Being Assertive With Your X (SystemVerilog Assertions for Dummies)*" [14].

Synthesis ignores assertions that are embedded in RTL code. This makes it possible for design engineers to place assertion checks within the design to validate RTL simulation and to guide formal verification, without having to modify the code before reading it into DC.

Potential RTL versus gate-level mismatch. SystemVerilog assertions can execute "pass statements" when an assertion evaluates as true, and "fail statements" when an assertion evaluates as false. These pass and fail statements can be any programming statement. Care must be taken to not put code in a pass or fail statement that modifies the design behavior. Since DC ignores the assertion, any functionality modeled in pass/fail statements will not be implemented in the gate-level output from synthesis.

13. Miscellaneous enhancements

There are several additional enhancements in SystemVerilog that are supported in the Synopsys synthesis tool flow, but which are not covered in detail in this paper. These enhancements include:

Auto-fill literal values. SystemVerilog provides a shortcut syntax for setting all bits of a vector of any size to the same value, using `'0`, `'1`, `'z` or `'x` .

Time unit specification. SystemVerilog adds the keywords `timeunit` and `timeprecision` for specifying simulation time units. These keywords can be used as declarations within a module definition or in `$unit`, and have several advantages over the Verilog `` `timescale `` compiler directive.

Macro enhancements. SystemVerilog enhances the Verilog `` `define `` text substitution macro to make it more versatile.

Variables in unnamed blocks. SystemVerilog allows local variables to be declared in a begin...end block, without naming the block. These local variables can only be used within the block. They are not visible outside of the block, and cannot be referenced using a hierarchical path.

Named ending statements. SystemVerilog allows specifying names with the end statement of modules, interfaces, named blocks, tasks and functions. This is a convenience enhancement that helps document code, but does not affect functionality.

Bottom testing loop. SystemVerilog adds a `do`...`while` loop, which functions the same as its C counterpart. In a synthesis design flow, the `do`...`while` loop has the same usage restrictions as a Verilog `while` loop.

New system tasks and functions. SystemVerilog adds several new system tasks and functions that can be used in a synthesis design flow (see Table 4 in Section 14).

14. Keyword support

The following tables summarize the SystemVerilog keyword support in the Synopsys synthesis tool flow.

Table 1: Synthesizable SystemVerilog keywords

always_comb	endinterface	priority
always_ff	enum	return
always_latch	import	shortint
bit	int	struct
break	interface	typedef
byte	logic	union
const	longint	unique
continue	modport	void
do	packed	

Table 2: SystemVerilog keywords that are ignored by synthesis

assert	endsequence	sequence
assume	expect	timeprecision
endproperty	property	timeunit

Table 3: Non synthesizable SystemVerilog keywords

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

Table 4: Synthesizable SystemVerilog system tasks and functions

$bits	$increment	$onehot0
$countones	$left	$right
$dimensions	$low	$size
$high	$onehot	$unit

Table 5: Non-synthesizable SystemVerilog system tasks and functions

$assertkill	$fatal	$sampled
$assertoff	$fell	$set_coverage_db_name
$asserton	$get_coverage	$shortrealtobits
$bitstoshortreal	$info	$stable
$cast	$isunbounded	$typename
$coverage_control	$isunknown	$urandom
$coverage_merge	$load_coverage_db	$urandom_range
$coverage_save	$past	$warning
$error	$root	$writememb
$exit	$rose	$writememh

SystemVerilog also defines a number of built-in methods for working with enumerated types, arrays, and queues. These methods are not supported in the current Synopsys synthesis design flow.

15. Recommendations

15.1 Additional synthesizable constructs

At the time this paper was written, there were a few of SystemVerilog constructs that were not supported in the Synopsys synthesis tool flow, and which Sutherland HDL hopes will be supported in future versions of the Synopsys tools used in a synthesis design flow. These additional constructs include:

- Packages

- Enumerated type methods
- User-defined net data types
- Leaving out the optional parameter keyword in module/interface parameter lists
- Task/function calls with named argument passing (similar to named port connections)
- Task/function **`ref`** ports
- **`case`**...**`inside`** select statements
- **`case`**...**`matches`** select statements
- **`foreach`** loops
- Statement labels
- Wildcard equality and inequality operators (`==?` and `!=?`)
- Operator overloading
- Array reduction methods
- Bounded queues and queue methods
- **`uwire`** single driver nets (defined in the IEEE 1364-2005 Verilog standard[4])
- Nested modules

Do not wait to adopt SystemVerilog in current synthesis design flows! The synthesis subset supported by Synopsys tools presented in this paper is powerful and useful. The benefits that can be seen from this subset are considerable. Design engineers can—and should—take advantage of SystemVerilog in current and forthcoming designs.. The recommendations specified in this section are useful, but do not prevent benefitting from SystemVerilog in synthesizable designs right now.

These additional constructs are not described in detail in this paper. Two of these constructs, however, packages and **`uwire`**, deserve special recommendations to the Synopsys R&D groups.

15.2 Packages

Packages provide a named shared declaration space, similar to the built-in **`$unit`** shared declaration space discussed in Section 2. The advantage of packages is that they enclose shared declarations in a well-defined syntactic space, between the keywords **`package`** and **`endpackage`**. Packages prevent the declaration "spaghetti code" that could result if **`$unit`** declarations are scattered throughout the many files that make up a typical design. Packages also provide the ability to define operator overloading.

15.3 uwire single driver nets

The new **`uwire`** data type is not a SystemVerilog enhancement; it is part of the new IEEE 1364-2005 Verilog standard[4]. The **`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 (either in the design specification, or due to a typographical error in design code). Currently, the **`uwire`** data type is not supported across all the tools used in a Synopsys synthesis design flow.

The author hopes that when Synopsys R&D implements the **`uwire`** type, they do not make the mistake of some of their competitor synthesis companies. Some synthesis compilers map **`uwire`** in RTL code to **`wire`** in the gate-level netlist. This mapping means that the single-driver semantics are lost, which can result in mismatches in the pre-synthesis RTL functionality and the post-synthesis gate-level functionality. The author recommends that DC maintain the **`uwire`** type in the resultant synthesized netlist, thus preserving the single-source semantics in post-synthesis simulation.

This same concept can also be applied to single-source variables. SystemVerilog syntax and semantics require that variables only have a single source when connected to a module output, the left-hand side of a continuous assignment, or are used in **`always_comb`**, **`always_latch`** or **`always_ff`** procedural blocks. If DC mapped variables used in these contexts to the **`uwire`** type in the gate-level netlist, the single-source semantics would be maintained in post-synthesis simulation. This could help find design problems at compile/elaboration time, instead of showing up as functional errors.

16. Summary

This paper has defined a synthesis subset for SystemVerilog that is already supported by the major tools used in a Synopsys tool synthesis design flow, including Leda, VCS, DC, and Formality. Sutherland HDL uses this synthesis subset in the consulting and training services we provide. We hope that other design engineers will find the synthesis subset specified in this paper a useful guideline for writing synthesizable models that are portable across multiple tools in a synthesis design flow.

17. 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] *"1364-2005 IEEE Standard for Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN:0-7381-4770-2.

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

[6] *"SystemVerilog for Design Engineers"*, book by Stuart Sutherland, Simon Davidmann, and Peter Flake, Springer-Verlag, Boston, Massachusetts. Copyright 2004. ISBN: 1-4020-7530-8.

[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) San Jose conference, 2005.

[8] *"SystemVerilog's priority & unique—A Solution to Verilog's 'full_case' & 'parallel_case' Evil Twins!"*, paper by Clifford Cummings, Synopsys Users Group (SNUG) Israel conference, 2005.

[9] *"Modeling FIFO Communication Channels Using SystemVerilog Interfaces"*, paper by Stuart Sutherland, Synopsys Users Group (SNUG) Boston conference, 2004.

[10] *"SystemVerilog in Use: First RTL Synthesis Experiences with Focus on Interfaces"*, paper by Peter Jensen, Wolfgang Ecker and Thomas Kruse, Synopsys Users Group (SNUG) Europe conference, 2004.

[11] *"A User's Experience with SystemVerilog"*, paper by Jonathan Bromley and Michael Smith, Synopsys Users Group (SNUG) Europe conference, 2004.

[12] *"SystemVerilog Implicit Port Connections-Simulation & Synthesis"*, paper by Clifford Cummings, DesignCon conference, 2005 (see www.sunburst-design/papers for an updated version).

[13] *"Assertions Are For Design Engineers, Too!"*, paper by Don Mills and Stuart Sutherland, Synopsys Users Group (SNUG) San Jose conference, 2006 and at SNUG Europe conference, 2006.

[14] *"Being Assertive With Your X (SystemVerilog Assertions for Dummies)"*, paper by Don Mills, Synopsys Users Group (SNUG) San Jose conference, 2004.

18. Acknowledgements

The author thanks the time spent by several Synopsys R&D engineers, Corporate Applications Engineers, and marketing personnel in reviewing drafts of this paper, answering questions, and helping to ensure the paper is accurate and useful.

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, the Verilog PLI, and the SystemVerilog DPI. You can contact Mr. Sutherland at stuart@sutherland-hdl.com, or by calling +1-503-692-0898.