Synthesizing SystemVerilog
Busting the Myth that SystemVerilog is only for Verification

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

Don Mills
Microchip Technology, Inc.
mills@microchip.com

ABSTRACT

SystemVerilog is not just for Verification! When the SystemVerilog standard was first devised, one of the primary goals was to enable creating synthesizable models of complex hardware designs more accurately and with fewer lines of code. That goal was achieved, and Synopsys has done a great job of implementing SystemVerilog in both Design Compiler (DC) and Synplify-Pro. This paper examines in detail the synthesizable subset of SystemVerilog for ASIC and FPGA designs, and presents the advantages of using these constructs over traditional Verilog. Readers will take away from this paper new RTL modeling skills that will indeed enable modeling with fewer lines of code, while at the same time reducing potential design errors and achieving high synthesis Quality of Results (QoR).

Target audience: Engineers involved in RTL design and synthesis, targeting ASIC and FPGA implementations.

Note: The information in this paper is based on Synopsys ***Design Compiler*** (also called ***HDL Compiler***) version 2012.06-SP4 and Synopsys ***Synplify-Pro*** version 2012.09-SP1. These were the most current released versions available at the time this paper was written.

Table of Contents

1. Data types 4
    1.1 Value sets 5
    1.2 Net types 6
    1.3 Variable types 6
    1.4 Vector declarations (packed arrays) 6
    1.5 Arrays (unpacked arrays) 7
    1.6 User-defined types 9
2. Parameterized models 13
3. Shared declaration spaces — packages and $unit 14
    3.1 Packages 14
    3.2 $unit 16
4. RTL programming 17
    4.1 Procedural blocks 17
    4.2 Operators 20
    4.3 Casting 22
    4.4 Decision statements 23
    4.5 Loop statements 26
    4.6 Tasks and functions 27
5. Module ports (internal to a module) 29
6. Netlists 30
7. Interfaces 31
8. Miscellaneous synthesizable SystemVerilog constructs 33
    8.1 Ending names 33
    8.2 `begin_keywords and `end_keywords 34
    8.3 Vector fill tokens 35
    8.4 Constant variables (const) 36
    8.5 timeunit and timeprecision 36
    8.6 Expression size functions ($clog2, $bits) 36
    8.7 Assertions 37
9. Other synthesizable constructs 38
10. Difference between Design Compiler and Synplify-Pro 38
11. Wish List and Recommendations 39
    11.1 uwire single source nets 39
    11.2 Foreach loops 40
    11.3 Task/function inputs with defaults 40
    11.4 Task/function ref arguments 41
    11.5 Set membership operator (inside) with expressions 42
    11.6 Package chaining 42
    11.7 Extern module declarations 43
    11.8 Configurations 43
    11.9 User-defined net types and generic net types 43
12. Summary 43
13. Acknowledgements 44
14. References 44

The term “synthesizable” refers to something that can be constructed, produced, or generated. In the context of computer science and hardware design, it often relates to creating hardware components or circuits using a high-level description (such as a hardware description language) that can be automatically translated into actual hardware. For example, when designing a digital circuit, a synthesizable description allows tools to generate the corresponding logic gates or hardware components.

1.0 Introduction — debunking the Verilog vs. SystemVerilog myth

There is a common misconception that “*Verilog*” is a hardware modeling language that is synthesizable, and “*SystemVerilog*” is a verification language that is not synthesizable. ***That is completely false!***

Verilog was first introduced in 1984 as a dual-purpose language to be used to both model hardware functionality and to describe verification testbenches. Many of the Verilog language constructs, such as **`if`**…**`else`** decision statements, were intended to be used for both hardware modeling and verification. A number of the original Verilog constructs were intended strictly for verification, such as the **`$display`** print statement, and have no direct representation in hardware. Synthesis is concerned with the hardware modeling aspect of the language, and therefore only supports a subset of the original Verilog language.

The IEEE officially standardized the Verilog language in 1995, with the standards number **1364-1995**, nicknamed ***Verilog-1995*** [1].The IEEE then began work on extending the language for both design and verification, and in 2001 released the **1364-2001** standard, commonly referred to as ***Verilog-2001*** [2]. A year later, the IEEE published the **1364.1-2002** ***Verilog RTL Synthesis*** standard [3], which defined the subset of Verilog-2001 that should be considered synthesizable.

The IEEE also updated the Verilog standard, as 1364-2005, aka Verilog-2005 [4]. However, Integrated Circuit functionality, complexity, and clock speeds evolved so rapidly in the 2000s, that an incremental update to the Verilog standard was not going to be enough to keep pace with the continually greater demand on the language capability to represent both hardware models and verification testbenches. The new features that the IEEE specified to enhance the Verilog language were so substantial that the IEEE created a new standards number, 1800-2005, and a new nickname, ***SystemVerilog*** [5], just to describe the language additions. ***SystemVerilog-2005 was not a stand-alone language — it was merely a set of extensions on top of Verilog-2005.*** One reason for the two documents was to help companies who provide Verilog simulators and synthesis compilers to focus on implementing all of the new capabilities.

The confusing name change... In 2009, the IEEE merged the Verilog 1364-2005 and the SystemVerilog extensions (1800-2005) into a single document. For reasons the authors have never understood, the IEEE chose to stop using the original Verilog name, and changed the name of the merged standard to SystemVerilog. The original 1364 Verilog standard was terminated, and the IEEE ratified the **1800-2009** ***SystemVerilog-2009*** standard [6] as a complete hardware design and verification language. In the IEEE nomenclature, there is no longer a current Verilog standard. There is only a SystemVerilog standard. ***Since 2009, you have not been using Verilog...you have been designing with—and synthesizing—SystemVerilog!*** (The IEEE has subsequently released a SystemVerilog-2012 standard, with additional enhancements to the original, now defunct, Verilog language.)

It is important to note that the SystemVerilog standard extended both the verification ***and*** *the hardware modeling capabilities of Verilog.* The language growth chart in Figure 1 that follows is not intended to be comprehensive, but serves to illustrate that a substantial number of the SystemVerilog extensions to the original Verilog enhance the ability to model hardware. The focus of this paper is on how these constructs synthesize and the advantages of using these SystemVerilog extensions in hardware design.

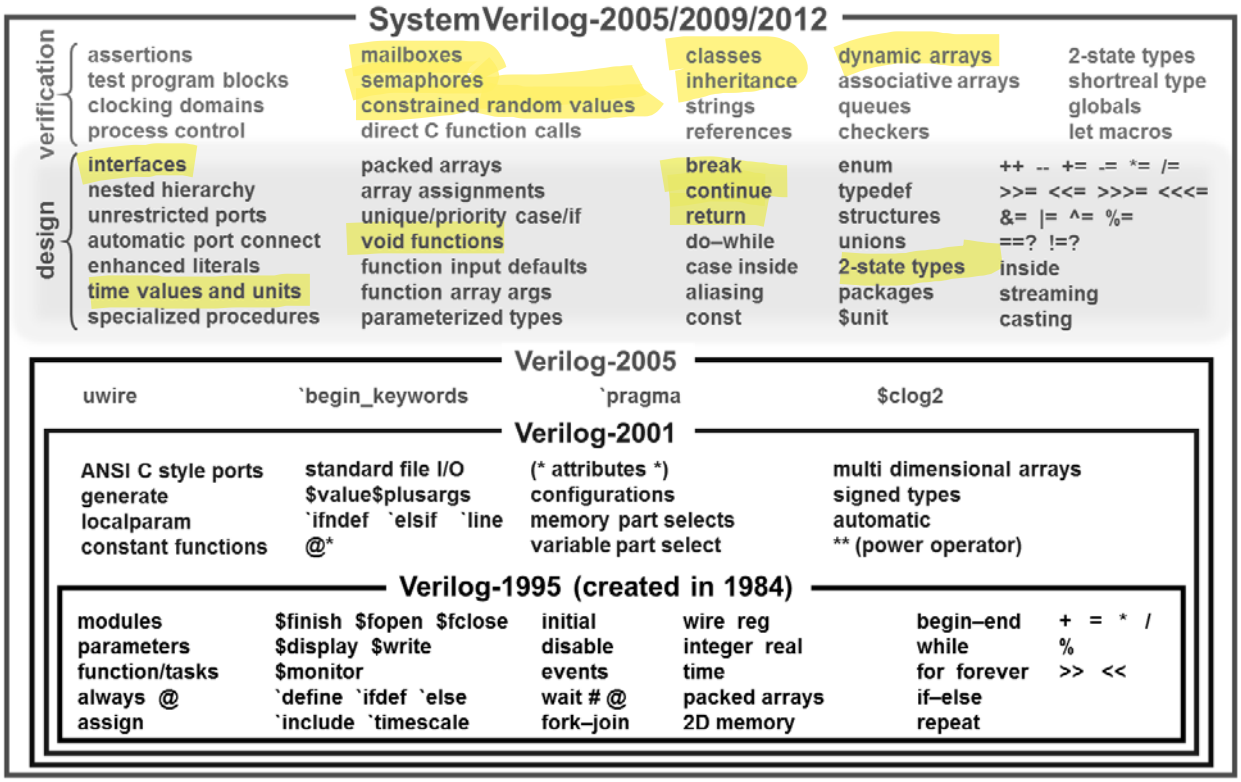


Figure 1. Verilog to SystemVerilog growth chart

The intent of this paper is to provide a comprehensive list of everything that is synthesizable with Synopsys ***Design Compiler*** (DC, also called HDL Compiler) and/or ***Synplify-Pro***. The paper focusses on the constructs that were added as part of SystemVerilog, and on how users can benefit from using these enhancements. Synthesizable modeling constructs that are from the various versions of the Verilog standard are mentioned for completeness, but are not discussed in detail in this paper.

It should be noted that there is no official SystemVerilog synthesis standard. The IEEE chose not to update the 1364.1 Verilog synthesis standard to reflect the many synthesizable extensions that were added with SystemVerilog. The authors feel that this is short-sighted and is a disservice to the engineering community, but hope that this paper, used in conjunction with the old 1364.1-2002 Verilog synthesis standard, can serve as an unofficial standard for the synthesizable subset of SystemVerilog.

2. Data types

Note: In this paper, the term "*value sets*" is used to refer to 2-state values (0 and 1) and 4-state values (0, 1, Z, X). The term "*data types*" is used as a general term for all net types, variable types, and user-defined types. The terms *value sets* and *data types* are not used in the same way in the official IEEE SystemVerilog standard [7], which is written primarily for companies that implement software tools such as simulators and synthesis compilers. The SystemVerilog standard uses terms such as "*types*", "*objects*" and "*kinds*", which have specific meaning for those that implement tools, but which the authors feel are neither commonplace nor intuitive for engineers that use the SystemVerilog language.

2.1 Value sets

The original Verilog language only had 4-state values, where each bit of a vector could be a logic 0, 1, Z or X. SystemVerilog added the ability to represent 2-state values, where each bit of a vector can only be 0 or 1. SystemVerilog added the **bit** and **logic** keywords to the Verilog language to represent 2-state and 4-state value sets, respectively. SystemVerilog net types, such as **wire**, only use the **logic** 4-state value set. Some variable types use 4-state **logic** value sets, while other variables use 2-state **bit** value sets. (There is more to the **bit** and **logic** keywords for those that implement simulators and synthesis compilers, but this generalization suffices for understanding how to model designs using SystemVerilog.)

The **bit** and **logic** keywords can also be used without explicitly defining a net or variable, in which case a net or variable is inferred from context. The keyword **bit** always infers a variable. The keyword **logic** infers a variable in most contexts, but infers a net if used in conjunction with a module **input** or **inout** port declaration. The following declarations illustrate these inference rules:

```
module A;
  ...
endmodule

module M (
  // module ports with inferred types
  input          i1, // infers a 4-state net
  input  logic   i2, // infers a 4-state net
  input  bit     i3, // infers a 2-state variable

  output         o1, // infers a 4-state net
  output logic   o2, // infers a 4-state variable
  output bit     o3  // infers a 2-state variable
);

  // internal signals with inferred and explicit types
  bit   clock;          // infers a 2-state variable
  logic reset;          // infers a 4-state variable
  logic [7:0] data;     // infers a 4-state variable
  wire  [7:0] n1;       // explicitly declares a net, infers 4-state logic
  wire logic [7:0] n2;  // explicitly declares a 4-state net
  var  [7:0] v1;        // explicitly declares a variable, infers logic
  var  logic [7:0] v2;  // explicitly declares a 4-state variable
  ...
endmodule
```

Important: Synthesis treats **bit** and **logic** the same. 2-state and 4-state value sets are for simulation, and have no meaning in synthesis.

> **SystemVerilog Advantage 1 —** You no longer need to worry about when to declare modules ports as **wire** or **reg** (or, more specifically, a net or a variable). With SystemVerilog, you can declare all module ports and local signals as **logic**, and the language will correctly infer nets or variables for you (there might be an occasional exception, where an engineer wishes to explicitly use a type other than what **logic** will infer, but those exceptions are rare).

Note that verification code is a little different. In a testbench, randomly generated test values should be declared as **bit** (2-state), rather than **logic** (4-state). See Sutherland [20] for a detailed exploration of using 2-state and 4-state types in design and verification code.

2.2 Net types

The synthesizable net types are:

- **`wire`** and **`tri`** — interconnecting nets that permit and resolve multiple drivers
- **`supply0`** and **`supply1`** — interconnecting nets that have a constant 0 or 1, respectively
- **`wand`**,**`triand`**, **`wor`**, **`trior`** — interconnecting nets that AND or OR multiple drivers together

Synthesizing these net types are not discussed in detail in this paper, since they have always been part of Verilog. Refer to the 1364.1 Verilog RTL Synthesis standard [3] or synthesis compiler documentation of information on synthesizing these traditional Verilog types.

> **SystemVerilog Advantage 2 —** (or at least it should be an advantage) — SystemVerilog also has a **`uwire`** net type that would be very beneficial for design work, but is not currently supported by synthesis. Section 12 of this paper discusses why **`uwire`** is an important advantage in design work.

2.3 Variable types

Variables are used in procedural code, also referred to as *always blocks*. Verilog/SystemVerilog requires that the left-hand side of procedural assignments must be a variable type. The synthesizable variable types in SystemVerilog are:

- **`reg`** — a general purpose 4-state variable of a user-defined vector size
- **`integer`** — a 32-bit 4-state variable
- **`logic`** — except on module input/inout ports, infers a general purpose 4-state variable of a user-defined vector size
- **`bit`** — infers a general purpose 2-state variable of a user-defined vector size
- **`byte`**, **`shortint`**, **`int`**, **`longint`** — 2-state variables with 8-bit, 16-bit, 32-bit and 64-bit vector sizes, respectively

The **`reg`** and **`integer`** types have always been part of Verilog, and are not discussed further in this paper.

The **`logic`** keyword is not really a variable type, but, in almost all contexts, **`logic`** will infer a **`reg`** variable. Because of this inference, the **`logic`** keyword can be used in place of **`reg`**, letting the language infer a variable.

The **`bit`**, **`byte`**, **`shortint`**, **`int`** and **`longint`** types only store 2-state values. Synthesis treats these types as a 4-state **`reg`** variable with a corresponding vector size. ***Caution:*** there is a risk of a functional mismatch between simulation and the synthesized implementation, because synthesis does not maintain the 2-state behavior. One potential difference is that 2-state variables begin simulation with a value of 0 in each bit, where the synthesized implementation might power-up with each bit a 0 or 1.

> ***Recommendation —*** Use **`logic`** for almost all declarations, and let the language infer a net type or a variable type based on context. Avoid all 2-state types in RTL models. These types can hide design problems (see Sutherland [20]), and can lead to simulation vs. synthesis mismatches. The one exception is to use an **`int`** variable for the iterator variable in **`for`**-loops.

2.4 Vector declarations (packed arrays)

Vectors are declared by specifying a range of bits in square brackets, followed by the vector name. The range is declared as **`[`** *most-significant_bit_number* **`:`** *least-significant_bit_number* **`]`** The msb and lsb can be any number, and the msb can be the largest or smallest number.

```
wire  [31:0] a;   // 32-bit vector, little endian
logic [1:32] b;   // 32-bit vector, big endian
```

Vector declarations, bit selects and part (multiple bit) selects of vectors have always been part of Verilog and are synthesizable. The Verilog-2001 standard added variable part selects, which are also synthesizable.

The SystemVerilog standard refers to vectors as *packed arrays*, to indicate that a vector represents an array of bits that are stored contiguously. The one significant enhancement that SystemVerilog adds is the ability to divide a vector declaration into subfields by using multiple ranges. For example:

```
logic  [3:0][7:0] a;   // 32-bit vector, divided into 4 8-bit subfields

a[2] = 8'hFF;    // assign to subfield 2 of the vector
a[1][0] = 1'b1;  // select a single bit of subfield 1
```

Multidimensional packed arrays and selections within multidimensional packed arrays are synthesizable. Using this SystemVerilog feature could be beneficial when a design needs to frequently reference subfields of a vector. The example above makes it simple to do byte selects out of the 32-bit vector.

2.5 Arrays (unpacked arrays)

SystemVerilog allows declaring single- and multiple-dimension arrays of nets, variables, and user-defined types (see section 2.6). Array dimensions are declared after the name of the array. Two examples are:

```
logic [7:0] LUT [0:255];           // one-dimensional array of 256 bytes

logic [7:0] RGB [0:15][0:15][0:15]; // three-dimensional array of bytes
```

An element of an array is selected using an index number.

```
data = LUT[7];           // select the byte at address 7 from the array

RGB[0][0][0] = 8'h1A;    // assign to address 0,0,0 of the array
```

Verilog arrays and selections of an array are synthesizable.

SystemVerilog extends Verilog arrays in several ways, some of which are very significant for modeling complex designs. These enhanced capabilities are discussed in sections 2.5.1 through 2.5.6.

2.5.1 C-style array declarations

Verilog arrays were declared by specifying the array address range, with the syntax **[** *first_address* **:** *last_address* **]**, such as [0:255]. SystemVerilog allows arrays to also be declared by specifying the array size, in the same way as in the C language. For example:

```
logic [7:0] LUT [256];           // one-dimensional array of 256 bytes

logic [7:0] RGB [16][16][16]; // three-dimensional array of bytes
```

When using this syntax, the array addressing always begins with address 0 and ends the array size minus 1. This minor convenience enhancement is synthesizable.

2.5.2 Copying arrays

Verilog only permitted access to a single element of an array at a time. To copy data from one array to another array required writing loops that indexed through each element of each array. SystemVerilog allows arrays to be copied as a single assignment statement. Either an entire array or part of an array can be copied. For example:

```
logic [31:0] big_array   [0:255];      // array with 256 32-bit elements
logic [31:0] small_array [0:15];       // array with 16 32-bit elements

assign small_array = big_array[16:31];   // copy 16 elements of big_array
```

Array copy assignments require that the number of dimensions and the number of elements in each dimension be identical on both sides of the assignment. The number of bits of each element must also be the same size and of compatible data types. Array copy assignments are synthesizable, and can significantly reduce the complexity of design code for moving blocks of data from one array to another.

2.5.3 Assigning value lists to arrays

All or multiple elements of an array can be assigned using a list of values, enclosed in **'{ }**. The list can contain values for an individual array element, or a default value for the entire array.

```
logic [7:0] a, b, c;
logic [7:0] d_array [0:3];      // array with 4 32-bit elements

always_ff @(posedge clock or negedge rstN)
  if (!rstN) d_array <= '{default:0};  // reset all elements of the array
  else       d_array <= '{8'h00, c, b, a}; // load the array
```

Array lists are synthesizable. The number of array dimensions and the number of values in the array list must be identical. The number of bits of each element must also be the same size as the values in the list.

2.5.4 Passing arrays through module ports and to tasks and functions

The ability to assign to multiple elements of an array also makes it possible to use arrays as module ports or task/function arguments. The following example defines a user-defined type that represents a 8x256 2-dimensional array of 32-bit elements, and then passes that array to and from a function, and through module ports. Section 2.6 discusses user-defined types in more detail, and section 4.1 discusses the appropriate place to define user-defined types.

```
typedef logic [31:0] d_array_t [0:7][0:255];

module block_data (input  d_array_t d_in,           // input is an array
                   output d_array_t q_out,          // output is an array
                   input  logic clock, rstN);

  function d_array_t transform (input d_array_t d);  // input is an array
    // ... perform operations on all elements of d
    return d;                                        // return is an array
  endfunction

  always_ff @(posedge clock or negedge rstN)
    if (!rstN) q_out <= '{default:0};    // reset entire q_out array
    else       q_out <= transform(d_in); // transform and store entire array
endmodule
```

Note that SystemVerilog requires that the values passed through an array port or task/function argument have the same number of dimensions, and that each element be the same vector size and a compatible type.

2.5.5 Array query system functions

SystemVerilog provides a number of special system functions that make it easier to manipulate an array without hard coding the size of the array. The synthesizable array query functions are: **$left()**, **$right()**, **$low()**, **$high()**, **$increment()**, **$size()**, **$dimensions()**, and **$unpacked_dimensions()**. Following is an example of using some of these functions:

```
typedef logic [31:0] d_array_t [0:15][0:15];

function d_array_t transform (input d_array_t d);
  for (int i = $low(d,1); i <= $high(d,1); i++) begin: outer_loop
    for (int j = $low(d,2); j <= $high(d,2); j++) begin: inner_loop
      // ... perform some sort of operation on each element of d
    end: inner_loop
  end: outer_loop
  return d; // function return is an array
endfunction
```

Note: This example could be significantly simplified by using a **`foreach`** loop. Unfortunately, **`foreach`** is not supported by DC or Synplify-Pro. See section 12.2 for more details on **`foreach`** loops.

2.5.6 Non-synthesizable array enhancements

SystemVerilog extends Verilog arrays in several other ways that are not synthesizable. These enhancements include a **`foreach`** array iterator loop, array operation functions, array locator functions, array sorting functions, and array bit-stream casting.

2.6 User-defined types

The original Verilog language only had built-in data types. SystemVerilog allows design and verification engineers 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. The synthesizable user-defined types are:

- enum — variables or nets with an enumerated list of legal values — see section 2.6.1
- struct — a structure comprising multiple nets or variables — see section 2.6.2
- union — a variable that can represent different types at different times — see section 2.6.3
- **`typedef`** — type definitions — see section 2.6.4

2.6.1 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, DONE} 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 `DONE` will have 32-bit **`int`** values. The labels in the enumerated list are constants that have an associated logic value. By default, the first label in the 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 `DONE` is 2.

Designers can specify an explicit base type, allowing enumerated types to more specifically model hardware. Designers can specify explicit values for any or all labels in the enumerated list. An example is:

```
// Two 3-bit, 4-state enumerated variables with one-hot values
enum logic [2:0] {WAITE = 3'b001,
                  LOAD  = 3'b010,
                  DONE  = 3'b100} State, NextState;
```

Enumerated types have stronger rule checking than built-in variables and nets. These rules include:

- The value of each label in the enumerated list must be unique
- The variable size and the size of the label values must be the same
- An enumerated variable can only be assigned:
  - A label from its enumerated list
  - The value of another enumerated type from the same enumerated definition

The stronger rules of enumerated types provide significant advantages over traditional Verilog. The following two examples contrast a simple state machine modeled in Verilog and in SystemVerilog. Both models have several coding errors, noted in the comments.

```
    // Names for state machine states (one-hot encoding)
    parameter [2:0] WAITE=3'b001, LOAD=3'b010, DONE=3'b001; // FUNCTIONAL BUG

    // Names for mode_control output values
    parameter [1:0] READY=3'b101, SET=3'b010, GO=3'b110;    // FUNCTIONAL BUG

    // State and next state variables
    reg [2:0] state, next_state, mode_control;

    // State Sequencer
    always @(posedge clock or negedge resetN)
      if (!resetN) state <= 0;                              // FUNCTIONAL BUG
      else         state <= next_state;

    // Next State Decoder (sequentially cycle through the three states)
    always @(state)
      case (state)
        WAITE: next_state = state + 1;                      // DANGEROUS CODE
        LOAD : next_state = state + 1;                      // FUNCTIONAL BUG
        DONE : next_state = state + 1;                      // FUNCTIONAL BUG
      endcase

    // Output Decoder
    always @(state)
      case (state)
        WAITE: mode_control = READY;
        LOAD : mode_control = SET;
        DONE : mode_control = DONE;                         // FUNCTIONAL BUG
      endcase
  endmodule
```

The six bugs in the preceding example are all syntactically legal. Simulation will compile and run. Hopefully, the verification code will catch the functional problems. Synthesis might warn about some of the coding errors, but some of the bugs would still end up in the gate-level implementation of the design.

The following example shows these same coding bugs, but with the use of enumerated types instead of Verilog parameters and **`reg`** variables (the example also uses some other SystemVerilog constructs presented later in this paper). The comments show that every functional bug in traditional Verilog has become a syntax error when using SystemVerilog — the compiler catches the bugs, instead of having to detect a functional bug, debug the problem, fix the bug, and re-verifying functionality.

```
module bad_fsm_systemverilog_style (...); // only relevant code shown

enum logic [2:0] {WAITE=3'b001, LOAD=3'b010, DONE=3'b001}  // SYNTAX ERROR
                  state, next_state;
enum logic [1:0] {READY=3'b101, SET=3'b010, GO=3'b110}     // SYNTAX ERROR
                  mode_control;

  // State Sequencer
  always_ff @(posedge clock or negedge resetN)
    if (!resetN) state <= 0;                               // SYNTAX ERROR
    else         state <= next_state;

  // Next State Decoder (sequentially cycle through the three states)
  always_comb
    case (state)
      WAITE: next_state = state + 1;                       // SYNTAX ERROR
      LOAD : next_state = state + 1;                       // SYNTAX ERROR
      DONE : next_state = state + 1;                       // SYNTAX ERROR
    endcase

  // Output Decoder
  always_comb
    case (state)
      WAITE: mode_control = READY;
      LOAD : mode_control = SET;
      DONE : mode_control = DONE;                          // SYNTAX ERROR
    endcase
endmodule
```

NOTE: DC does not catch the second syntax error in this example, but VCS does detect it.

SystemVerilog also provides several methods for working with enumerated types. The synthesizable methods are: **`.first`**, **`.last`**, **`.next`**, **`.prev`** and **`.num`**, The ordering is based on the declaration order in the enumerated list. The round-robin next state decoder in the previous example can be written more succinctly by using an enumerated method, as follows:

```
always_comb
  next_state = state.next; // tranistions from WAITE to LOAD, from LOAD to
                           // DONE, and from DONE back to WAITE
```

Although the enumerated methods can simplify code in some situations, their application in real-world designs is somewhat limited. The authors feel that it is a better coding style to assign enumerated labels, rather than use the enumerated methods. Using labels makes the code more self-documenting and provides more flexibility in state machine branching.

Note: At the time this paper was written, Synplify-Pro did not support enumerated methods.

> **SystemVerilog Advantage 3 —** Enumerated types can prevent coding errors that could be difficult to detect and debug! Use enumerated types whenever a variable or net can have only a limited set of legal values.

2.6.2 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 {
  logic [31:0] source_address;
  logic [31:0] destination_address;
  logic [63:0] data;
  logic [3:0]  ecc;
} packet;
```

Individual members of a structure can be accessed using a dot operator (**.**).

```
packet.source_address = 32'h0000dead;
```

Much more useful, is that structures can be read or written as a whole. An entire structure can be copied to another structure, provided the two structures come from the same definition. This requires using **typedef**, which is shown in section 2.6.4. Using **typedef** to define a structure type also makes it possible to pass entire structures through module ports or to tasks and functions.

All members of a structure can also be assigned using a list of values, enclosed in **'{ }**. The list can contain values for individual structure members, or a default value for one or more members of the structure.

```
always_ff @(posedge clock or negedge rstN)
  if (!rstN) packet <= '{default:0};  // reset all members of packet
  else       packet <= '{old_addr, new_addr, data_in, ecc_func(data_in)};
```

By default, the members of a structure are stored in any way a software tool deems optimal. Most of the time, this will yield the best simulation and synthesis quality of results. Designers can take control over how members of a structure are stored by declaring the structure as packed. A packed structure stores all members contiguously, with the first member being the left-most (most significant bits) of the storage. Packed structures are useful in conjunction with packed unions (see section 2.6.3).

```
struct packed {   // members will be stored contiguously
  logic [ 1:0] parity;
  logic [63:0] data;
} data_word;
```

SystemVerilog Advantage 4 — By using structures to collect related variables together, the collection can be assigned and transferred to other modules as a group, reducing the lines of code and ensuring consistency. Only use packed structures when the structure will be used in a union.

2.6.3 Unions

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. Packed unions require that all representations 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.

The following example represents a single 64-bit hardware register that can store either a data packet or an instruction packet.

```
union packed {
  struct packed {
    logic [31:0] data;
    logic [31:0] address;
  } data_packet;
```

```
    struct packed {
      logic [31:0] data;
      logic [31:0] operation;
    } instruction_packet;
  } packet_u;

  always_ff @(posedge clock or negedge rstN)
    if (!rstN)                packet_u <= {'0, '0};  // reset
    else if (op_type == DATA) packet_u.data_packet <= {d_in, addr};
    else                      packet_u.instruction_packet <= {d_in, instr};
```

The assignment statements in the last three lines of this example are assigning a concatenation of values into either the `data_packet` structure or the `instruction_packet` structure. This is legal and functionally correct because the members of these structures are "packed".

Note: It is also syntactically legal — and would be a preferred coding style — to assign a list of values into a structure (see section 2.1), but DC does not support assigning value lists to structures containing union members.

2.6.4 Type definitions (typedef)

New data types are constructed from built-in types and other user-defined types using **`typedef`**, similar to C. Some simple examples are:

```
typedef logic [31:0] bus64_t;   // 64-bit bus

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

typedef enum logic {FALSE=1'b0, TRUE=1'b1} bool_t;

module D (input  data_t  a, b,
          output bus64_t result,
          output bool_t  aok );
  ...
endmodule
```

SystemVerilog also provides a **`package`** construct to encapsulate typedef definitions and other definitions. See section 4.1 for details on using packages with synthesis.

> **SystemVerilog Advantage 5 —** User-defined types, even for simple vector declarations, can guarantee that declarations are consistent throughout a project. Using user-defined types can prevent accidental mismatches in size or type.
>
> ***Recommendation* —** Use **`typedef`** liberally. Even simple vectors of a specific size that will be used throughout a design, such as address and data vectors, should be defined as a user-defined type. All **`typedef`** declarations should be encapsulated in one or more packages (see section 4.1).

3. Parameterized models

Verilog has the ability to use parameters and parameter redefinition to make models configurable and scalable. Parameterized models are not discussed in this paper, as they have always been part of synthesizable Verilog.

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

```
module adder #(parameter type dtype = logic [0:0]) // default is 1-bit size
              (input  dtype a, b,
               output dtype sum);
  assign sum = a + b;
endmodule

module top (
  input  logic [15:0] a, b,
  input  logic [31:0] c, d,
  output logic [15:0] r1,
  output logic [31:0] r2);

  adder #(.dtype(logic [15:0]))        i1 (a, b, r1); // 16 bit adder
  adder #(.dtype(logic signed [31:0])) i2 (c, c, r2); // 32-bit signed adder
endmodule
```

Parameterized data types are synthesizable. Note that SystemVerilog-2009 made the **`parameter`** keyword optional when using the **`#(...)`** module parameter list, but DC still requires the parameter keyword.

4. Shared declaration spaces — packages and $unit

4.1 Packages

The original Verilog language did not have a shared declaration space. Each module contained all declarations used within that module. This was a major language limitation. If the same parameter, task or function definition was needed in multiple modules, designers had to connive awkward work-a-rounds, typically using a combination of **`‘ifdef`** and **`‘include`** compiler directives. The addition of SystemVerilog user-defined types, object-oriented class definitions, and randomization constraints made the lack of a shared declaration space a severe problem.

SystemVerilog addresses the Verilog shortcoming with the addition of user-defined packages. Packages provide a declaration space that can be referenced from any design module, as well as from verification code. The synthesizable items packages can contain are:

- **`parameter`** and **`localparam`** constant definitions
- **`const`** variable definitions
- **`typedef`** user-defined types
- Fully automatic **`task`** and **`function`** definitions
- **`import`** statements from other packages
- **`export`** statements for package chaining

An example package is:

```
package alu_types;
  localparam DELAY = 1;

  typedef logic [31:0] bus32_t;
  typedef logic [63:0] bus64_t;

  typedef enum logic [3:0] {ADD, SUB, ...} opcode_t;
```

```
  typedef struct {
    bus32_t i0, i1;
    opcode_t opcode;
  } instr_t;

  function automatic logic parity_gen(input d);
    return ^d;
  endfunction
endpackage
```

Note that a **parameter** defined in a package cannot be redefined, and is treated the same as a **localparam**. Also note that synthesis requires that tasks and functions defined in a package be declared as **automatic**.

4.1.1 Referencing package definitions

The definitions within a package can be used within a design block (i.e.: a module or interface) in any of three ways, all of which are synthesizable:

- Explicit package reference
- Explicit import statement
- Wildcard import statement

Explicit references of a package item use the package name followed by **::** . For example:

```
module alu
  (input  alu_types::instr_t instruction, // use package item in port list
   output alu_types::bus64_t result );
  alu_types::bus64_t temp;              // use package item within module
  ...
endmodule
```

An explicit reference to a package item does not make that item visible elsewhere in the module. An explicit package reference must be used each time the definition is used within the module.

Explicit imports of a package item use an **import** statement. Once imported, that item can be referenced any number of times within the module. For example:

```
module alu
  import alu_types::bus64_t;
  (input  alu_types::instr_t instruction, // explicit package reference
   output bus64_t result );              // bus64_t has been imported
  bus64_t temp;                          // bus64_t has been imported
  ...
endmodule
```

Wildcard imports use an asterisk to represent all definitions within the package. Wildcard imports make all items of the package visible within a module. For example:

```
module alu
  import alu_types::*;
  (input  instr_t instruction,           // instr_t has been imported
   output bus64_t result );              // bus64_t has been imported
  bus64_t temp;                          // bus64_t has been imported
  ...
endmodule
```

SystemVerilog Advantage 6 — Packages can eliminate duplicate code, the risk of mismatches in different blocks of a design, and the difficulties of maintaining duplicate code.

***Recommendation* —** Use packages! Packages provide a clean and simple way to reuse definitions of tasks, functions, and user-defined types throughout a design and verification project.

4.1.2 Placement of import statements

The placement of the import statement in the previous two examples is important. A package item must have been already imported in order to use that definition in a module port list. In SystemVerilog-2005, the import statement could only appear after the module port list, which was too late. SystemVerilog-2009 added the ability to place the import statement before the module port list (and before the parameter list, if used). SystemVerilog-2009 has been a released standard for more than three years, but Synopsys was not very quick at implementing this subtle, but important, change in the SystemVerilog standard.

Note: DC supports package imports before the port list, but requires using **`set hdlin_sverilog_std 2009`** to enable that support. At the time this paper was written, Synplify-Pro did not yet support package imports before the module port list.

4.1.3 Importing a package into another package

A package can also reference definitions from another package, and can import definitions from another package. Importing packages into other packages is synthesizable. SystemVerilog also allows *package chaining*, which simplifies using packages that reference items from other packages.

Note: Package chaining is not supported by DC. See section 12.6 for more details on package chaining.

4.1.4 Package compilation order

SystemVerilog syntax requires that package definitions be compiled before they are referenced. This means that there is file order dependency when compiling packages and modules. It also means that a module that references package items cannot be compiled independently; the package must be compiled along with the module (or have been pre-compiled, if the tool supports incremental compilation).

4.2 $unit

Prior to packages being added to the SystemVerilog standard, a different mechanism was provided to create definitions that could be shared by multiple modules. This mechanism is a pseudo-global name space 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.

Note: **`$unit`** is a dangerous shared name space that is fraught with hazards. Briefly, some of the hazards of using **`$unit`** are:

- Definitions in **`$unit`** can be scattered across many files, making code maintenance a nightmare.
- When definitions in the $unit space are in multiple files, the files must be compiled in a very specific order, so that each definition is compiled before it is referenced.
- Each invocation of a compiler starts a new **`$unit`** space that does not share declarations in other **`$unit`** spaces. Thus, a compiler that compiles multiple files at a time, such as VCS, will see a single **`$unit`** space, whereas a compiler that can compile each file independently, such as DC, will see several disconnected **`$unit`** spaces.
- It is illegal in SystemVerilog to define the same name multiple times in the same name space. Therefore, if one file defines a `bool_t` type in the **`$unit`** space, and another file also defines `bool_t` in **`$unit`**, a compilation or elaboration error will occur if the two files are compiled together.
- Named packages can be imported into **`$unit`**, but care must be taken to not import the same package more than once. Multiple imports of the same package into the same name space is illegal.

> ***Recommendation — Avoid using $unit like the Bubonic plague!*** Instead, use named packages for shared definitions. Named packages avoid all of the hazards of **`$unit`**.

5. RTL programming

SystemVerilog adds a number of significant programming capabilities over traditional Verilog. The intent of these enhancements is three-fold: 1) to be able to model more functionality in fewer lines of code, 2) to reduce the risk of functional errors in a design, and 3) to help ensure that simulation and synthesis interpret design functionality in the same way.

5.1 Procedural blocks

In traditional Verilog, procedural **`always`** blocks are used to model combinational, latch, and sequential logic. Synthesis and simulations tools have no way to know what type of logic an engineer intended to represent. Instead, these tools can only interpret the code within the procedural block, and then "*infer*", which just a nice way to say "*guess*", the engineer's intention.

Simple coding errors in combinational logic can generate latches with no indication that latch behavior exists until the synthesis result reports are examined. Many re-spins have occurred due to missing unexpected latches in lengthy synthesis reports.

SystemVerilog adds three new types of procedural **`always`** blocks that document intent and also provide some synthesis rule checking. These are: **`always_comb`**, **`always_latch`**, and **`always_ff`**. Simulators, lint checkers, and synthesis compilers can issue warnings if the code modeled within these new procedural blocks does not match the designated type. These warnings are optional, and are not required by the SystemVerilog standard. At this time, there are no simulation tools that provide such warnings, but lint checkers and synthesis compilers do.

5.1.1 always_comb

The **`always_comb`** procedural block indicates that the designer's intent is to represent combinational logic. The following side-by-side examples contrast the traditional Verilog **`always`** procedural block with a SystemVerilog **`always_comb`** procedural block:

```
always @(a or b or sel) begin
  if (sel) y = a;
  else     y = b;
end
```

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

Using **always_comb** has several major benefits over the generic Verilog **always** procedure.

The first important benefit to note is that tools can infer a combinatorial sensitivity list, because tools know the intent of the procedural block. A common coding mistake with traditional Verilog is to have an incomplete sensitivity list. This is not a syntax error, and results in RTL simulation behaving differently than the post-synthesis gate-level behavior. Inadvertently leaving out one signal in the sensitivity list is an easy mistake to make in large, complex decoding logic. It is an error that will likely be caught during synthesis, but that only comes after many hours of simulation time have been invested in the project. Verilog-2001 added the always @* construct to automatically infer a complete sensitivity list, but this construct is not perfect, and, in some corner cases, simulation and synthesis infer different lists. The SystemVerilog **always_comb** procedural block has very specific rules that ensure that all software tools will infer the same, accurate combinatorial sensitivity list.

Another important benefit is that, because software tools know the intent is to represent combination logic, tools can verify that this intent is being met. The following code example illustrates a procedural block that uses always_comb, but does not correctly behave as combinational logic. Following the example is the report and warning that are issued by DC for this code.

```
module always_comb_test
  (input  logic a, b,
   output logic c);

    always_comb
      if (a) c = b;

endmodule: always_comb_test
```

```
============================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
============================================================================
|        c_reg        | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
============================================================================
Warning: test.sv:5: Netlist for always_comb block contains a latch. (ELAB-
974)
```

When this previous code example was read into the DC synthesis tool, DC generated a “Register table” indicating the register type as “Latch”. Additionally, an elaboration warning was issued, noting that a latch was modeled in an **always_comb** block.

5.1.2 always_latch

The special **always_latch** procedural block is very similar to **always_comb**, except that it documents the designer’s intent to represent latch behavior. Tools can then verify that this intent is being met. The next code example uses **always_latch**, but models combinational logic.

```
module always_latch_test
  (input        a, b, c,
   output logic out);
```

```
  always_latch
    if (a)  out = b;
    else    out = c;

endmodule: always_latch_test
```

The warning that is issued by DC for this code is:

```
Warning:  /test.sv:7: Netlist for always_latch block does not contain a
latch. (ELAB-975)
```

When this previous code example was read into the synthesis tool, an elaboration warning was issued, noting that no latch was modeled from the **always_latch** block.

5.1.3 always_ff

The **always_ff** procedural block documents the designer's intent to represent flip-flop behavior. **always_ff** differs from **always_comb**, in that the sensitivity list must be specified by the designer. This is because software tools cannot infer the clock name and clock edge from the body of the procedural block. Nor can a tool infer whether the engineer intended to have synchronous or asynchronous reset behavior. This information must be specified as part of the sensitivity list.

Software tools can still verify whether the intent for flip-flop behavior is being met in the body of the procedural block. In the following example, **always_ff** is used for code that does not actually model a flip-flop. The warning that is issued by DC follows the example.

```
module always_ff_test
  (input        a, b, c,
  output logic out);

  always_ff @(a, b, c)
    if (a)  out = b;
    else    out = c;

endmodule: always_ff_test

Warning:  test.sv:5: Netlist for always_ff block does not contain a flip-
flop. (ELAB-976)
```

5.1.4 Additional advantages of specialized procedures

In addition to indicating designer's intent (combinational logic, latch logic, or flip-flops) the **always_comb**, **always_latch** and **always_ff** SystemVerilog procedural blocks provide other rule checks and features that help ensure RTL code will synthesize into the gate-level implementation intended:

- The variables on the LHS of assignments cannot be written to by any other processes
- **always_comb** and **always_latch** will execute once at time zero of the simulation, ensuring the variables on the left-hand side of assignments within the block correctly reflect the values on the right-hand side at time 0.
- **always_comb** and **always_latch** are sensitive to signal changes within a function called by the procedural block, and not just the function arguments, which was a bug with **always @(*)**

> **SystemVerilog Advantage 7 —** The benefits of the SystemVerilog **always_comb**, **always_latch** and **always_ff** procedural blocks are ***huge***! They can prevent serious modeling errors, and they enable software tools to verify that design intent has been met.

Recommendation — Use **`always_comb`**, **`always_latch`** and **`always_ff`** in all RTL code. Only use the general purpose **`always`** procedure in models that are not intended to be synthesized, such as bus-functional models, abstract RAM models, and verification testbenches.

5.2 Operators

The original Verilog language provides many operators, most of which are synthesizable. This paper lists these operators, but does not go into detail on their use and the synthesis rules for them.

- *Bitwise operators* — perform operations on each bit of a vector
  `~ & | ^ ^~ ~^`
- *Unary reduction operators* — collapse a vector to a 1-bit result
  `~ & | ^ ~& ~| ^~ ~^`
- *Logical operators* — return a true/false result, based on a logical test
  `&& ||`
- *Equality, relational operators* — return true/false, based on a comparison
  `= = != < <= > >=`
- *Shift operators* – shift the bits of a vector left or right
  `<< <<< >> >>>`
- *Concatenate operators* — join multiple expressions into a vector
  `{ } {n{ }}`
- *Conditional operator* — selects one expression or another
  `?:`
- *Arithmetic operators* — perform integer and floating point math
  `+ - * / % **`

SystemVerilog adds a number of operators to traditional Verilog that are synthesizable:

5.2.1 Case equality operators (==?, !=?)

The *case equality operators*, also referred to as *wildcard equality operators*, compare the bits of two values, with ability to mask out specific bits. The operator tokens are: **`==?`** 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 bits are masked from the comparison
```

These operators synthesize the same as **`==`** and **`!=`**, but with the masked bits ignored in the comparator, following the same synthesis rules and restrictions as the Verilog **`casex`** statement

5.2.2 Set membership operator (inside)

The **`inside`** set membership operator compares a value to a list of other values enclosed in **`{ }`**. The list of values can be a range of values between **`[ ]`**, or can be the values stored in an array. The inside set membership operator allows bits in the value list to be masked out of a comparison in the same way as the case equality operators.

```
if (data inside {[0:255]}) ...  // if data is between 0 to 255, inclusive
if (data inside {3'b1?1}) ...   // if data is 3'b101, 3'b111, 3'b1x1, 3'b1z1
```

Note: The values enclosed in the { } set must be constant expressions in order for the **`inside`** operator to be synthesizable. There are other ways in which the **`inside`** operator can be used which are not synthesizable. See section 12.5 for more details.

5.2.3 Streaming (aka pack and unpack) operators (<<, >>)

The streaming operators pull-out or push-in groups of bits from or to a vector in a serial stream. The streaming operators can be used to either pack data into a vector or unpack data from a vector.

- **{<<M{N}}** — stream M-size blocks from N, working from right-most block towards left-most block
- **{>>M{N}}** — stream M-size blocks from N, working from left-most block towards right-most block

Packing occurs when the streaming operator is used on the right-hand side of an assignment. The operation will pull blocks as a serial stream from the right-hand expression, and pack the stream into a vector on the left-hand side. The bits pulled out can be in groups of any number of bits. The default is 1 bit at a time, if a size is not specified.

```
logic [ 7:0] a;
logic [ 7:0] b = 8'b00110101;

always_comb
  a = { << { b }} // sets a to the value 8'b10101100 (bit reverse of b)
```

Unpacking occurs when a streaming operator is used on the left-hand side of an assignment. Blocks of bits are pulled out of the right-hand expression, and assigned to the expression(s) within the streaming operator.

```
logic [ 7:0] a [0:3];
logic [31:0] e = 32'AABBCCDD;

always_comb
  {>>8{a}} = e; // sets a[0]=AA, a[1]=BB, a[2]=CC, a[3]=DD
```

Note: DC does not support using the streaming operators for selecting blocks of bits, such as the byte reordering example above.

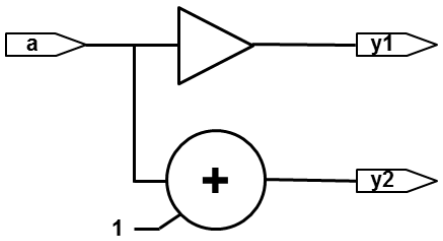
The SystemVerilog streaming operators can be used in a number of creative ways to pack and unpack data stored in vectors, arrays, and structures. The operators are synthesizable.

5.2.4 Increment/decrement and assignment operators (++, --, +=, -=, etc.)

The **++** and **--** increment and decrement are two of the most commonly used operators in C, and yet the Verilog language did not have these operators. SystemVerilog adds these operators. Increment/decrement operators perform a blocking assignment to their operand. The **++** and **--** increment and decrement operators can be used on the right-hand side of assignment statements. For example:

```
always_comb begin
  y1 = a++;        // same as: y1 = a; a = a + 1;
  y2 = a;
end
```

The functionality represented by this example is:



Note: DC does not support the use of increment and decrement operators on the right-hand side of assignment statements, as shown in the preceding example.

SystemVerilog also adds the C-like assignment operators to the Verilog language. These operators combine an operation and an assignment to a variable. The operators are:

```
+=   -=   /=   %=   &=   ^=   !=   |=   <<=   >>=   <<<=   >>>=
```

An example of using assignment operators is:

```
always_comb
  accumulator += b;   // same as: accumulator = accumulator + b;
```

These increment, decrement, and assignment operators might seem like they would be a convenient short cut for modeling synthesizable RTL functionality. It turns out that their usefulness in synthesizable RTL code is somewhat limited. The problem is that these operators execute as blocking assignments. When used in flip-flop based RTL code, these blocking assignments can introduce subtle and dangerous simulation race conditions.

```
always_ff @(posedge clk, negedge rstn)
  if(!rstN) cnt <= '0;
  else if (en) cnt++;   // same as cnt = cnt + 1, which is a potential race
```

The blocking assignment behavior limits the usefulness of these operators in RTL code. The reality is, operators such as **++** and **+=** are primarily only useful as **for**-loop step assignments.

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

Recommendation **—** Do not use the increment, decrement, and assignment operators in functionality that updates on a clock edge.

5.3 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);`**

One usage of casting is to eliminate those annoying lint checker “size mismatch” warning messages. The following example is a synthesizable coding style to rotate an expression a variable number of times.

```
logic [31:0] a, y;
logic [ 5:0] b;

always_comb
y = {a,a} >> b;  // rotate a by b times, assign 64-bit result to 32-bit LHS
```

This code relies on the Verilog/SystemVerilog rule that when the left-hand side of an assignment is fewer bits than the right-hand side, the upper bits are automatically truncated off. This is the desired functionality, and there is no problem with the code, despite a pesky “size mismatch” warning message.

Using a SystemVerilog size cast eliminates this warning message.

```
y = logic [31:0]'({a,a} >> b);  // cast result to 32 bits before assigning
```

Recommendation **—** Use casting to eliminate type or size mismatch warning messages. The casting also serves to document that change in type of size is intended.

Note that SystemVerilog also adds a **`$cast()`** dynamic cast system function, which is not synthesizable.

5.4 Decision statements

The primary constructs used for RTL modeling are the decision statements **`if`**…**`else`** and **`case`** (with its wildcard variations). These decision constructs are the heart of RTL modeling, and are used to model combinational logic, latches, and flip-flops. Care must be taken to ensure that **`if`**…**`else`** and **`case`** are coded to generate the intended hardware. Failing to follow proper coding guidelines can cause simulation versus synthesis mismatches (see Mills and Cummings [11]). Since this paper was first published in 1999, significant SystemVerilog language enhancements have been added to Verilog to help reduce or eliminate these mismatches. One of the ugliest gotchas of traditional Verilog is the **`casex`**/**`casez`** problem. Many conference papers have focused on the problems caused by these constructs, and have recommended limited usage. The SystemVerilog **`case`**...**`inside`** statement replaces the **`casex`** and **`casez`**. Another significant SystemVerilog enhancement are **`unique`**, **`unique0`** and **`priority`** decision modifiers.

5.4.1 case...inside

The SystemVerilog **`case`**...**`inside`** decision statement allows mask bits to be used in the *case items*. The don't care bits are specified, using X, Z or ?, as with **`casex`**. The important difference is that **`case`**...**`inside`** uses a one-way, asymmetric masking for the comparison, unlike **`casex`**, where any X or Z bits in the *case expression* are also masked out from the comparison. With **`case`**...**`inside`**, any X or Z bits in the *case expression* are not masked. In the following example, any X or Z bits in `instruction` will not be masked, and an invalid instruction will be trapped by the default condition:

```
always_comb begin
  case (instruction) inside
    4'b0???: opcode = instruction[2:0];    //only test msb bit
    4'b1000: opcode = 3'b001;
    ... // decode all other valid instructions
  endcase
end
```

The asymmetric masking of the *case expression* is similar to how synthesis interprets the *case expression*. That is, synthesis assumes the *case expression* is a known value (all bits are 0 or 1).

> **SystemVerilog Advantage 8 — `case`...`inside`** can prevent subtle design problems, and ensure that simulation closely matches how the synthesized gate-level implementation will behave.

> ***Recommendation* —** Synthesizable RTL code should never use **`casex`** or **`casez`**.

Note: Synplify-Pro did not add support for **`case`**...**`inside`** until version 2013.03.

Best practices coding styles should also include an assertion check on the *case expression* for RTL. This check could be done as the default case item, shown section 9.7.

5.4.2 Unique, unique0, and priority decisions

From the early days of synthesis, DC has provided synthesis directives (also called "*pragmas*") to guide the mapping of **`case`** statements to gates. Two key directives are **`full_case`** and **`parallel_case`**. The primary problem with these synthesis directives is that they represent the design differently to the synthesis tool than how the design is simulated. There have been many papers authored documenting this issue (see Mills and Cummings [11], Cummings [15], Mill [19], and others). These authors recommend these synthesis directives only be used with an "*inverse case statement*". This is a case statement in which the *case expression* is a constant, and the *case items* contain variables (which is the opposite, or inverse, of typical **`case`** statements, and hence the nick-name "*inverse case statement*"). The most common application for an inverse case statement is decoding one hot states, as shown in the following example:

```
// full_case applied to one-hot state machine

logic [3:0] state, next_state;

always_comb begin      // next state logic decode
  next_state = '0;     // latch prevention
  case (1'b1)  // synopsys full_case parallel_case
    state[0]: next_state[1] = 1'b1;
    state[1]: next_state[2] = 1'b1;
    state[2]: next_state[3] = 1'b1;
    state[3]: next_state[0] = 1'b1;
  endcase
end
```

The inverse **case** statement synthesizes to one bit comparators, making the decoding very fast.

Caution: The **full_case** and **parallel_case** synthesis directives are not right for all case statements. Other than with inverse case statements, using **full_case** and/or **parallel_case** can lead to synthesized designs that do not match the RTL simulations of the design. Cliff Cummings, a prominent SystemVerilog trainer, has said, "*These directives are the most dangerous when they actually work!*". The reason these directives can be dangerous is that the synthesis optimizations change the gate-level implementation to be different than what had been verified in RTL simulation.

A major problem with the **full_case** and **parallel_case** synthesis directives is that the commands are hidden in comments — simulation is not affected by the directives, and therefore there is no verification that the optimization effects of the directives are appropriate in the design.

SystemVerilog solves this problem by bringing the functionality of **full_case** and **parallel_case** into the RTL modeling world, using the keywords **priority**, **unique0** and **unique** decision modifiers. The following table shows the mapping between the new SystemVerilog decision modifiers and the old synthesis directives:

Table 1: Decision modifiers vs. synthesis directives

SystemVerilog Decision Modifier	Synthesis Directive (Pragma)
priority	**full_case**
unique0	**parallel_case**
unique	**full_case**, **parallel_case**

These modifiers help to prevent the gotchas common to the synthesis **full_case** and **parallel_case** pragmas by bringing the pragma functionality into the RTL simulation realm. The modifiers affect both synthesis and simulation. For synthesis, the modifiers enable the appropriate **full_case** and/or **parallel_case** synthesis optimizations. For simulation, the decision modifiers have built-in checking to help detect when these optimizations would not be desirable. This checking can help detect two common coding errors that can occur with decision statements:

- Not all case expression values that occur have been specified in the case items (incomplete decisions).
- Redundant (duplicate) decision branches have been specified.

The **priority**, **unique** and **unique0** decision modifiers have built-in checking for these potential design errors. With a **unique case** decision, simulation and synthesis tools will print a violation report if overlapping *case items* exist, indicating that the *case items* cannot be evaluated in parallel. Simulators will also print a run-time violation report if the **case** statement is entered and there are no matching *case items*.

```
enum logic [3:0] {READY=3'b001, SET=3'b010, GO=3'b100} state, next_state;

always_comb
  unique case (1'b1) // inverse case statement
    state[0]: next_state = SET;
    state[1]: next_state = GO;
    state[2]: next_state = READY;
  endcase
```

The example above will print a violation report if multiple conditions match, such as if `state` has a value of `3'b111`, or if there are no matches, such as if `state` has a value of `3'b000`. These violation reports are important! They indicate that `state` values are occurring that the gate-level implementation will not decode if the `case` statement is synthesized with **`full_case`**, **`parallel_case`** optimization.

A **`priority case`** decision provides a run-time violation report if the **`case`** statement is entered and there is no matching condition.

```
enum logic [3:0] {READY=3'b001, SET=3'b010, GO=3'b100} state, next_state;

always_comb
  priority case (1'b1) // inverse case statement
    state[0]: next_state = SET;
    state[1]: next_state = GO;
    state[2]: next_state = READY;
  endcase
```

The example above will not give a violation report if multiple conditions match, such as if `state` has a value of `3'b111`, because the designer has indicated that the first matching branch should have priority. However, a violation report will occur if there are no matches, which can occur if `state` has a value of `3'b000`. This violation report indicates that the code is modeling a latch condition. If no latches were intended, the violation report allows the design problem to be fixed before going to synthesis.

It is important to note that the use of **`priority`**, **`unique`** and **`unique0`** decision modifiers will flag some of the conditions that can cause unwanted latches, but not all conditions. Consider what happens in the following code if `state` never has a value of `3'b000` during simulation:

```
enum logic [3:0] {READY=3'b001, SET=3'b010, GO=3'b100} state, next1, next2;

always_comb
  unique case (1'b1) // inverse case statement
    state[0]: next1 = SET;
    state[1]: next2 = GO;      // assign to different variable
    state[2]: next1 = READY;
  endcase
```

In this example, if one, and only one, bit of `state` is set when the case statement is executed, no violation reports are generated, and yet latches will be inferred because only some of the outputs are assigned for each of the conditions.

SystemVerilog Advantage 9 — The **`priority`**, **`unique0`** and **`unique`** decision modifiers have built-in checking that will catch many of the potential hazards when using the synthesis **`full_case`** and **`parallel_case`** optimizations.

SystemVerilog Advantage 10 — The **`priority`**, **`unique0`** and **`unique`** decision modifiers can also be used with **`if`**...**`else`** decisions. This gives the same synthesis optimizations that could only

be done with **case** statements when using the **full_case** and **parallel_case** synthesis directives. The **priority**, **unique0** and **unique** keywords also provide simulation checking, to help ensure the optimized **if**...**else** decisions will work as intended.

***Recommendation* —** Use the appropriate SystemVerilog **priority**, **unique0** or **unique** decision modifier instead of the synthesis **full_case** or **parallel_case** directives. Note, however, that it is not always desirable to have the synthesis gate minimizations that can result from either the synthesis directives or the corresponding decision modifiers. These decision modifiers should be used with caution. Also note that these decision modifiers do not prevent latches, and do not flag all conditions that might infer latches.

Note: The **unique0** case was not supported by VCS, DC or Synplify-Pro at the time this paper was written.

5.5 Loop statements

The original Verilog language has three types of synthesizable loop constructs: **for**, **repeat**, and **while**. Each of these loops requires adhering to specific coding restrictions in order to be synthesized. These restrictions are outside the scope of this paper. SystemVerilog enhances the Verilog **for** loop capabilities, and adds two additional types of loops that are useful for modeling hardware designs at the RTL level.

The enhancements to **for** loops are the ability to declare the loop control variable as part of the loop, and the ability to have multiple initial assignments and multiple step assignments. Two example SystemVerilog-style **for** loops are:

```
for (int i=0; i<=15; i++) ...          // i is declared as part of the loop

for (int i=0, j=15; j>0; i++, j--) ... // multiple loop control variables
```

The ability to declare the loop control variable(s) as part of the **for** loop is more than just a convenience. It can prevent inadvertent, difficult-to-debug coding errors. With traditional Verilog, the loop control variable must be declared outside of, and prior to, the loop. Typically, the variables are declared at the module level. Complex modules often have more than one **always** procedural block within the module that have **for** loops. If each loop uses the same variable name for its control, **i**, for instance, the loops can collide, with multiple loops modifying the same variable at the same time. The result can be bizarre simulation behavior that is challenging to debug.

SystemVerilog adds a **do**...**while** loop, with similar syntax and usage as its C language counterpart. The **do**...**while** loop is synthesizable, with the same coding restrictions as Verilog **while** loops. The advantage of **do**...**while** over **while** is that the loop control is tested at the bottom of the loop, rather than at the top. A bottom-testing loop guarantees that the loop will execute at least once, which helps ensure that any variables assigned within the loop will be initialized.

SystemVerilog makes it easier to control the execution of loops. In complex logic, it is often necessary to abort a pass through a loop when certain conditions exist, or to abort the loop completely. Traditional Verilog could do this by using the **disable** statement, but the syntax is awkward and non-intuitive. SystemVerilog adds the C-like **break** and **continue** statements, which make it simpler to control loop execution. Using **break** and **continue**, instead of **disable**, also makes code more self-documenting.

Note: SystemVerilog also adds a **foreach** loop specifically for iterating through arrays and vectors. The **foreach** loop was not supported by either DC or Synplify-Pro at the time this paper was written. See section 12.2 for the advantages of **foreach** loops.

5.6 Tasks and functions

The traditional Verilog language has task and function subroutines. The primary differences between a Verilog task and function are:

- A task can have input, output and inout arguments; a function can only have input arguments.
- A task can have delays (execution blocks until simulation time advances); a function must execute in zero time.
- A task assigns results to output or inout arguments; a function returns a result.
- A task is used like a statement in procedural code; a function is used like an expression (the function return is the value of the expression).

Traditional Verilog tasks and functions are synthesizable, as long as certain coding restrictions are followed. These restrictions are outside the scope of this paper. SystemVerilog enhances Verilog tasks and functions in several ways. The synthesizable extensions are explained in the following subsections.

5.6.1 Functions with output and inout formal arguments

With traditional Verilog, functions could only have input arguments. SystemVerilog allows functions to also have output and inout arguments, in the same way as tasks. This enhancement becomes very important when coupled with void functions.

5.6.2 Void functions

SystemVerilog adds the ability to declare a function as **`void`**, indicating the function does not have a return value (but can assign to output arguments). A void function is used in the same way as a task, but with the requirement that it must execute in zero time (the function cannot contain any construct that might block until simulation time advances).

```
function void ripple_add (input [31:0] a, b, output [31:0] sum, output co);
  ... // adder algorithm must execute in zero time
endfunction

always_comb
  ripple_add(in1, in2, result, carry); // function is called as a statement
```

SystemVerilog Advantage 11 — Using void functions in RTL code can help ensure that subroutines will synthesize. This is because functions cannot have delays, which is also a general synthesis restriction.

Recommendation — Use void functions instead of tasks in RTL models that will be synthesized. This prevents a common problem when using tasks, where the model works in simulation, but then the task won't synthesize. This problem is much less likely to occur with void functions.

5.6.3 Formal arguments default to input

SystemVerilog simplifies defining a task or function, by assuming the argument direction is input, instead of requiring the input keyword. This can make function declarations that only have inputs more concise and more C-like.

5.6.4 Arrays, structures, unions and user-defined types as formal arguments

SystemVerilog adds several synthesizable user-defined types, as has been discussed in section 2.6. Tasks and functions were extended to support the ability of working with these types. SystemVerilog also allows arrays to be copied in, or copied out, of a task or function.

5.6.5 Pass by name in task/function calls

Traditional Verilog required that calls to a task or function pass values in or out, using the order in which the formal arguments were defined in the task or function. An inadvertent coding error, where two arguments passed in the wrong order, could lead to subtle design errors that were difficult to debug. SystemVerilog allows a task or function call to pass values to the task or function using the name of the formal arguments. The syntax is the same as with module instances and using connect-by-name. Using pass-by-name makes the task or function call more self-documenting, and can prevent coding mistakes.

```
function void ripple_add (input [31:0] a, b, output [31:0] sum, output co);
  ...
endfunction

always_comb
  ripple_add(.sum(result), .co(carry), .a(in1), .b(in2) );
```

5.6.6 Function return values can be specified, using return

Traditional Verilog returned a value from a function by assigning to the function name, similar to the Pascal language. SystemVerilog adds the more conventional, C-like way of using a **return** keyword to specify a function return.

```
function [31:0] adder ([31:0] a, b);
  adder = a + b; // Verilog style
endfunction

function [31:0] adder ([31:0] a, b);
  return a + b; // SystemVerilog style
endfunction
```

5.6.7 Parameterized task/function arguments using static classes

Parameterized modules are a powerful and widely used capability in Verilog. Parameters can be redefined for each instance of the module, making the module easily configurable and reusable. In traditional Verilog, the formal argument size and type could not be parameterized in the same way that modules could be. This reduced the ability to write reusable, configurable models. SystemVerilog provides a way to work around this limitation. The technique is to use static tasks or functions in a parameterized class. Each time the task or function is called, the class parameters can be redefined, as shown in the following example:

```
virtual class Functions #(parameter SIZE=32);
  static function [SIZE-1:0] adder (input [SIZE-1:0] a, b);
    return a+b; // defaults to a 32-bit adder
  endfunction
endclass

module top (input  logic [63:0] a, b,
            output logic [63:0] y) ;

  always_comb
    y = Functions #(.SIZE(64))::adder(a,b);  // reconfigure to 64-bit adder
endmodule
```

With parameterized tasks and functions, it is possible to create and maintain only one version of the task or function, instead of having to define several versions with different data types, vector widths, or other characteristics. DC places two restrictions on parameterized classes that are not SystemVerilog restrictions: The class must be declared as a **virtual** class, and the class must be defined in the **$unit** declaration space (a very bad place for declarations — see section 4.2).

***Note*:** At the time this paper was written, tasks and functions in a parameterized class were not supported in Synplify-Pro.

5.6.8 Unsupported task/function features

There are two important enhancements to traditional Verilog tasks and functions that are not currently synthesizable: input ports with default values, and **`ref`** arguments. These features are discussed in sections 12.3 and 12.4, as part of the authors' synthesis wish list.

6. Module ports (internal to a module)

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 logic [63:0] bus64_t;

  typedef enum logic {FALSE, TRUE} bool_t;

  typedef struct {
    logic [31:0] i0, i1;
    logic [ 7:0] opcode;
  } instruction_t;
endpackage

module ALU (output user_types::bus64_t        result,
            output user_types::bool_t         ok,
            input  user_types::instruction_t  IW,
            input  var logic                  clock);
  ...
endmodule
```

Note: Synthesis will change the port names and/or types of ports that comprise structures, unions, arrays and user-defined types. Structure and array ports are expanded to separate ports. Union ports are changed to the name of the union, without reference to the member name(s) within the union. This difference means that instances of the module in other models must be modified to match the different port connections. Following is the output from DC for the example above. The line breaks have been modified to fit the page width, and the full list of ports and declarations has been abbreviated.

```
module ALU ( result, ok, .IW({\IW[i0][31] , \IW[i0][30] ,  ...  \IW[i0][0] ,
            \IW[i1][31] , \IW[i1][30] ,  ...  \IW[i1][0] , \IW[opcode][7] ,
            \IW[opcode][6] ,   ...   \IW[opcode][0] }), clock );
  output [63:0] result;
  input \IW[i0][31] , \IW[i0][30] ,  ...  \IW[i0][0] , \IW[i1][31] ,
        \IW[i1][30] ,  ...  \IW[i1][0] , \IW[opcode][7] , \IW[opcode][6] ,
        ...  \IW[opcode][0] , clock;
  output ok;
  ...
endmodule
```

With DC, this same example can be synthesized with a `change_names -rules verilog` setting. The output with this option is more straightforward, but it is still different than the pre-synthesis port list.

```
module ALU ( result, ok, IW, clock );
  output [63:0] result;
  input [71:0] IW;
  input clock;
  output ok;
  ...
endmodule
```

7. Netlists

Netlist is used to connect modules together. Netlists are synthesizable, as long as specific restrictions are adhered to. These restrictions are outside the scope of this paper. SystemVerilog adds some important enhancements to traditional Verilog netlists. This section discusses the synthesizable enhancements.

Traditional Verilog's explicit named port connection syntax for instantiating a module can be verbose, and often requires 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 when connecting signals to an instance of a module or interface using explicitly named ports, referred to as *dot-name* and *dot-star* port connections.

The "*dot-name*" shortcut simplifies netlists when port names and signal names are the same. Only the port needs to be explicitly named; the name of the signal being connected can be omitted. 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. Note that to synthesize the dot-star shortcut 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.

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

The dot-name and dot-star shortcuts do more than just reduce redundant names. These shortcuts also add some important rule checking that can help prevent coding mistakes in a netlist.

- ***The net being connected must be explicitly declared.*** Dot-name and dot-star will not imply a net, which can prevent netlist errors that sometime occur with traditional Verilog's implicit nets.
- ***The size of the port and the net must be the same.*** Traditional Verilog allows mismatches in connection sizes, which is often the result of an erroneous net declaration or a missing net declaration.
- ***Unconnected ports must be explicitly declared (dot-star only).*** Traditional Verilog will infer that any port not explicitly listed in a module instance was intended to be left unconnected. Often, however, unconnected ports were unintentional and are a coding mistake. The dot-star shortcut requires that any unconnected ports be explicitly listed. Note that the dot-name syntax does not have this additional check, much to the authors' dismay.

SystemVerilog Advantage 12 — The dot-name and dot-star shortcuts in netlists can significantly simplify large netlists, and, at the same time, prevent design errors.

8. Interfaces

A general coding guideline in software is to use subroutines whenever the same code is required in more than one place, rather than duplicating that code. With this in mind, consider the following traditional Verilog code snippet.

```
module top;
  wire  sig1, sig2, sig3, sig4;
  wire  clock, reset;

  mod_a u1 (.sig1(sig1),
            .sig2(sig2),
            .sig3(sig3),
            .sig4(sig4),
            .clk(clock),
            .rstN(reset) );

  mod_b u2 (.sig1(sig1),
            .sig2(sig2),
            .sig3(sig3),
            .sig4(sig4),
            .clk(clock),
            .rstN(reset) );
endmodule

module mod_a (input  sig1, sig2,
              input  clk, rstN,
              output sig3, sig4);
  ...
endmodule

module mod_b (input  sig3, sig4,
              input  clk, rstN,
              output sig1, sig2);
  ...
endmodule
```

In this basic example, signals traversing between `mod_a` and `mod_b` are listed seven times. If a design change occurs, and an additional signal needs to be added between the two module instances, that code will need to be modified in seven different places, and probably in different source code files. Similarly, seven changes would be required if the specification of a port size changed.

A SystemVerilog interface can be used to encapsulate signal declaration information in one place. The example below uses an interface to encapsulate the signals from the above example into a single location. Now, if a signal needs to be added between the two modules, or a change to a vector width is needed, only a single source code location needs to be modified.

```
interface intf_1;
  wire  sig1, sig2, sig3, sig4;
endinterface: intf_1

module top;
  wire  clock, reset;
  intf_1 i1();
  mod_a  u1 (.a1(i1), .clk(clock), .rstN(reset) );
  mod_b  u2 (.b1(i1), .clk(clock), .rstN(reset) );
endmodule: top
```

```
module mod_a (interface a1,
              input     clk, rstN);
  ...
endmodule: mod_a

module mod_b (intf_1 b1,
              input  clk, rstN);
  ...
endmodule: mod_b
```

In the mod_a example above, interface port a1 is declared as an **interface** port type, instead of the traditional **input**, **output** or **inout** port direction. This is referred to a "*generic interface port*". It is legal to connect an instance of any interface definition to a generic interface port. In the mod_b example, interface port b1 is declared as an **intf_1** port type. This is referred to a "*type-specific interface port*". It is only legal to connect an instance of an intf_1 interface to this type-specific interface port.

> ***Recommendation —*** Only use type-specific interface ports in design models. Designs are written to expect specific signals within the interface port. A type-specific interface port ensures the intended interface, with its internal signals, will be connected to that port.

Note: Generic interface ports are synthesizable, but a module with a generic interface port cannot be the top module for elaboration.

The primary purpose of interfaces is to bundle together signals that are related in some way, such as all the signals that make up an AMBA bus, or all the signals that make up a USB bus. Although syntactically possible, interfaces are not intended to bundle all the ports of a module together into a single port. (Verification is a different story — interfaces are often used between the testbench and the design under test, where the interface might bundle signals together that are not functionally related.)

The intf_1 interface shown in the example above is synthesizable, because all the signals in the interface are net types. For synthesis, if any of the signals in the interface are variable types, then a **modport** is required to specify port direction, as shown below.

```
interface intf_2;
   wire       sig1, sig3;
   var logic  sig2, sig4;
   modport a_ports (input  sig1,
                    output sig2);
   modport b_ports (input  sig3,
                    output sig4);
endinterface: intf_2
```

Interfaces can be parameterized in the same way as modules. Parameter redefinition of each instance of an interface makes it possible to configure interfaces to match modules that have parameterized port sizes.

Arrays of interfaces can be used, as shown in the following two examples. The first example uses an array in a straight forward way. The second example maps the interface to modules, using a generate block.

```
module top1;
  intf_3 i1 [1:0]();
  mod_a  u1 (.a1(i1[0]), .clk(clock), .rstN(reset) );
  mod_b  u2 (.b1(i1[0]), .clk(clock), .rstN(reset) );

  mod_a u3 (.a1(i1[1]), .clk(clock), .rstN(reset) );
  mod_b u4 (.b1(i1[1]), .clk(clock), .rstN(reset) );
endmodule: top1
```

```
module top2;
  intf_3 i1 [1:0]();

  genvar i;
  generate
    for (i=0; i<=1; i++)
      begin:m
        mod_a u1 (.a1(i1[i]), .clk(clock), .rstN(reset) );
        mod_b u2 (.b1(i1[i]), .clk(clock), .rstN(reset) );
      end
  endgenerate
endmodule: top2
```

Interfaces can contain functionality in the same way as modules, such as always blocks, continuous assignments, tasks and functions (tasks and functions must be automatic).

```
interface intf_4;
  parameter SIZE = 16;

  wire      [SIZE-1:0] sig1, sig3;
  var logic [SIZE-1:0] sig2, sig4;

  ... // modport declarations

  function automatic logic [7:0] ecc_f (input i);
   ...
  endfunction

  assign sig4 = ecc_f(sig1);

endinterface: intf_4
```

To be synthesizable, any functional code within an interface must adhere to synthesizable RTL coding rules. This might require bringing signals such as clocks and resets into the interface, which could limit the reusability of that interface. The authors recommend limiting any functionality within an interface to functions, simple combinational logic, and assertions.

> **SystemVerilog Advantage 13 —** Interfaces, when used properly, can substantially reduce redundant declarations within a design. This leads to code that is easier to maintain and easier to reuse in other projects.
>
> ***Recommendation —*** Use interfaces to bundle related signals together. Avoid bundling non-related signals, such as clocks and resets.

9. Miscellaneous synthesizable SystemVerilog constructs

SystemVerilog adds several more synthesizable constructs to the original Verilog language.

9.1 Ending names

SystemVerilog allows a name to be specified at the end of any named group of code. The ending name must match the block name; mismatches are reported as an error. Named ends help document code, significantly aiding in code debug, maintenance, and reuse.

```
module FSM (...);
  ...
  always_ff @(posedge clock) begin: Sequencer
    case (SquatState)
      2'b01: begin: rx_valid_state
               Rxready <= '1;
               for (int j=0; j<NumRx; j++) begin: loop1
                 for (int i=0; i<NumRx; i++) begin: loop2
                   if (Rxvalid[i]) begin: match
                     ... // receive data
                   end: match
                 end: loop2
               end: loop1
             end: rx_valid_state
      ... // decode other states
    endcase
  end: Sequencer

  task get_data(...);
    ...
  endtask: get_data
endmodule: FSM
```

9.2 ‘begin_keywords and ‘end_keywords

Each generation of the Verilog and then SystemVerilog standard has reserved additional keywords for use by the language. This can make it difficult to compile legacy code with newer versions of software tools that use the latest version of the standard. For example, the word “*priority*” was not in the original Verilog language. It was therefore legal to use `priority` as a module name, function name, or a net/variable name. SystemVerilog-2005 reserved **`priority`** as a keyword (see section 5.4), and it is a synthesizable construct. Compiling an older model with the current version of VCS, DC or Synplify-Pro would result in an error, unless the software tool is told that the code was written for an earlier version of the standard.

Verilog and SystemVerilog software tools have creative ways to inform the tool about which version of the standard to use as the source code is compiled. Most tools use file extensions as a gross indicator of the language version. A file ending with **`.v`** indicates the file contents contain some version of Verilog, and a file ending with **`.sv`** indicates the contents are in some version of SystemVerilog. Tools also provide command line options or other commands to specify which language version should be used. For example, VCS has **`+v2k`** (for Verilog-2001) and **`-sverilog`** (for SystemVerilog-2005). DC has the settings **`set hdlin_sverilog_std 2005`** and **`set hdlin_sverilog_std 2009`**.

SystemVerilog provides a standard way to specify the language version of Verilog or SystemVerilog source, using the compiler directive pair: **`begin_keywords “`***<IEEE_std_number>***`”`** and **`endkeywords`**. For example:

```
`begin_keywords "1364-1995"
module test;
  wire priority;   // "priority" is a legal name in Verilog (not a keyword)
  ...
endmodule
`end_keywords

`begin_keywords "1800-2005"
module decoder (...);
```

```
  always_comb
    priority case (...);  // "priority" is a keyword in SystemVerilog
  ...
endmodule
`end_keywords
```

(These keyword directives were actually added in the Verilog-2005 standard, not SystemVerilog, but since Verilog-2005 was released at the same time as SystemVerilog, it is generally considered a SystemVerilog enhancement to traditional Verilog.)

The **`begin_keywords** directive serves as code documentation, as well as a control to software tools. The directive shows exactly what version of Verilog or SystemVerilog was being used when the design code was written. The effect of **`begin_keywords** remains in effect until its corresponding **`end_keywords** is encountered. The effect of **`begin_keywords** is stacked, allowing nested calls to the directive as source code is read in.

> **SystemVerilog Advantage 14 —** The **`begin_keywords** directive documents the language version in use when the code was written, and helps to ensure forward compatibility with future versions of the SystemVerilog standard (or related standards such as a SystemVerilog-AMS).
>
> ***Recommendation* —** Use **`begin_keywords** at the beginning of every source code file, and a matching **`end_keywords** at the end of every source code file.

Note: At the time this paper was written, Synplify-Pro did not support these important directives.

9.3 Vector fill tokens

In traditional Verilog, there was no simple way to fill a vector with all 1's. The only options were to define a literal value of all ones with a hard coded vector width, or use operations, such as replicate or invert. In the following example, if parameter `N` is redefined to something larger than 64 bits, the design will no longer function as intended.

```
parameter N = 64;
reg [N-1:0] d, q;

always_ff @(posedge clk or negedge setN)
  if (!setN)
    q <= 64'hFFFFFFFFFFFFFFFF;   // sets at most 64 bits of q to 1
  else
    q <= d;
```

SystemVerilog adds a simple construct to specify that all bits of an expression should be set to 0, 1, X or Z. The vector size of the literal value is automatically determined based on its context.

- **'0** fills all bits on the left-hand side with 0
- **'1** fills all bits on the left-hand side with 1
- **'z** fills all bits on the left-hand side with z
- **'x** fills all bits on the left-hand side with x

```
always_ff @(posedge clk or negedge setN)
  if (!setN)
    q <= '1;   // set all bits of q to 1, regardless of the size of q
  else
    q <= d;
```

9.4 Constant variables (const)

SystemVerilog allows any variable to be declared with a constant value, by adding **const** to the declaration of the variable. One usage of **const** constants is in automatic functions.

```
function automatic int do_magic (a, b);
  const int magic_num = 86;
  ...
endfunction: do_magic
```

9.5 timeunit and timeprecision

Verilog's time units and time precision have no meaning in synthesis, but are essential in simulation. Synthesis compilers ignore time specifications, but they are mentioned in this paper because of their importance in simulation. Traditional Verilog used a **`timescale** compiler directive to specify the time information. A hazard with **`timescale** is that it is not bound by modules or by files. The directive is in effect from the time the compiler encounters it until a replacement directive is encountered. This can create a dependency in the order in which source code is read by the compiler.

SystemVerilog has an important extension for specifying time units and precision. The information can now be specified using the keywords **timeunit** and **timeprecision** inside a module or interface, which limits the information to the scope of that module or interface. Synthesis ignores these keywords, which allows them to be used in code that will be both simulated and synthesized

```
module alu (...);
  timeunit 1ns;
  timeprecision 1ns;
  ...
endmodule: alu
```

The units and precision can also be specified as a single statement, using **timeunit 1ns/1ns;**

> **SystemVerilog Advantage 15 —** These **timeunit** and **timeprecision** keywords eliminate the hazards of the **`timescale** directive, and should be used at the beginning of every module or interface, even when the code within does not specify any delays.

9.6 Expression size functions ($clog2, $bits)

SystemVerilog provides two special system functions that can be useful in synthesizable RTL code, **$clog2** and **$bits**. These functions can help prevent errors in declaration sizes, and help engineers to write models that are scalable and reusable as specifications change in current or future projects.

The **$clog2** function returns the ceiling of the log base 2 (the log rounded up to an integer value) of a vector. The function can be used in RTL models to declare vector sizes based on the value of a **parameter** or **localparam** constant. In the following example, the vector size of `xfer_size` will automatically scale to the number of bits needed to represent the value of `MAX_PAYLOAD`.

```
package my_types;
  localparam MAX_PAYLOAD = 64;

  typedef struct {
    logic [63:0] start_address;
    logic [$clog2(MAX_PAYLOAD)-1:0] xfer_size; // vector width scales
    logic [ 7:0] payload [0:MAX_PAYLOAD-1];
} packet_t;
endpackage: my_types
```

The **$clog2** function was added in the Verilog-2005 standard, but since Verilog-2005 was released at the same time as SystemVerilog, it is generally considered a SystemVerilog enhancement.

The **$bits** system function that returns the number of bits comprised in a net or variable name, or an expression. The basic usage is:

- **$bits(***data_type***)**
- **$bits(***expression***)**

  where *data_type* can be any language-defined data type or a user-defined type, and *expression* can be any value including operations or unpacked arrays.

For synthesis, **$bits** can be used in port declarations, variable declarations, and constant definitions (**parameter**, **localparam**, **const**). The following example uses **$bits** to declare the size of a register. The register will automatically scale to the size of the `payload` in the preceding package example.

```
module my_chip
 (input  logic                          clk,
  input  my_types::packet_t             in,
  output logic [$bits(in.payload)-1:0] out ); // self-scaling vector size

  always_ff @(posedge clk)
    out <= { << { in.payload }};   // pack the payload array into out reg
endmodule: my_chip
```

9.7 Assertions

SystemVerilog adds two types of assertions to the original Verilog language: immediate assertions and concurrent assertions. The authors have often heard comments that assertions are for verification engineers. ***That is a gross misconception!*** Assertions can and should be written by designers. Refer to other papers written by the authors for why and how designers can benefit from SystemVerilog Assertions (see Mills and Sutherland [17], Mills [18], Mills [19] and Sutherland [20]).

The example below illustrates an assertion that design engineers can — and should — add to synthesizable RTL models. This simple one line of code will trap any problems with the `sel` input to the module at the time and place the problem occurs, rather than hoping that verification will detect a functional problem downstream in both logic and time. Sutherland [20] discusses why this assertion, inserted by the design engineer, has several advantages over other coding styles to trap possible design problems.

```
module mux41 (input        a, b, c, d,
                input  [1:0] sel,
                output logic out);

  always_comb begin
    assert (!$isunknown(sel)) else $error("%m : case_Sel = X");
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
    endcase
  end
endmodule: mux41
```

DC and Synplify-Pro ignore assertions in design code, which allows assertions to be used in code that will be both simulated and synthesized.

10. Other synthesizable constructs

Traditional Verilog contained a number of other constructs that are synthesizable. These constructs are also part of SystemVerilog, but were not extended by SystemVerilog. For completeness, these other synthesizable constructs are listed here, but are not described in any detail.

- Attributes (treated as comments)
- Generate statements
- Variable part selects
- Primitives (built-in only, User-defined Primitives are not synthesizable)

11. Difference between Design Compiler and Synplify-Pro

The Synopsys Design Compiler and Synplify-Pro synthesis compilers are closely aligned in the SystemVerilog language constructs supported for synthesis. There are, however, a few differences, which can be important for designers who are using both tools. The table below summarizes these differences. Only SystemVerilog constructs supported by one tool, and not the other, are listed in this table. Constructs that are supported by both tools are not listed.

Table 2: Differences in SystemVerilog support in DC vs. Synplify-Pro

SystemVerilog Language Construct	Design Compiler	Synplify-Pro
`‘begin_keyword`, **`‘end_keyword`** compatibility directives	yes	no
Package import before module port list	yes	no
Parameterized tasks and functions, using parameterized static classes	yes	no
Enumerated type methods (**`.next`**, **`.prev`**, etc.)	yes	no
`‘__FILE__` and **`‘__LINE__`** debug macros	yes	no
`priority` and **`unique`** modifier to **`if`**...**`else`**	yes	ignored
Cross module references (XMRs)[1]	no	yes
`real` data type	no	yes
Increment or decrement operator on right-hand side of assignment statement	no	yes
Nets declared from **`typedef struct`** definitions	no	yes
Extern module declarations	no	yes
`$onehot`, **`$onehot0`**, **`$countones`**	no	yes
Interface modport expressions	no	yes
Immediate assertions	ignored	yes
`let` macros	ignored	yes
Checkers	no	ignored
`expect` statements	no	ignored

[1] The HDL Compiler (DC) reference manual says that cross-module references are supported "if the hierarchical name remains inside the module that contains the name, and each item on the hierarchical path is part of the module containing the reference." This restriction means that references to interface port contents are legal, but references to the contents of some other module are illegal.

12. Wish List and Recommendations

The SystemVerilog constructs that are supported by DC and Synplify-Pro can significantly improve RTL design. SystemVerilog has other constructs that *could* be synthesizable, and can also make designing for simulation and synthesis easier and more productive. This section lists several of these constructs, and why it would be beneficial for synthesis to support them.

12.1 uwire single source nets

The **`uwire`** type does not permit multiple drivers on a net. ***You should be using the* `uwire` *net type — it can prevent subtle, dangerous design bugs!*** By default, module input ports with no explicit type specified will default to the **`wire`** net type. Netlists that connect modules together are also typically modeled using **`wire`** net types. The **`wire`** type will silently resolve multiple drivers, but in most designs, nearly all module inputs and interconnecting netlists are intended to have a single source. The **`wire`** type does not check nor enforce only having a single driver. An inadvertent typographical error or re-use of a signal name can result in unintended multiple drivers that can go undetected in simulation. Greene, Salz and Booth [21] reported how a typo in a large design resulted in two drivers on a control signal, but the design appeared to work correctly in exhaustive simulations. The bug was not caught until the verification was run using the proprietary VCS -xprop simulation option, which caused an X to occur in simulation, but only after several thousands of clock cycles had been simulated with one specific test case.

The **`` `default_nettype``** compiler directive can be used to make **`uwire`** the default type for module input ports and undeclared nets in a netlist.

```
`default_nettype uwire
module program_counter (
  input               clock, resetN, loadN,
  input  logic [15:0] new_count,
  output logic [15:0] count );
  ...
endmodule: program_counter
`default_nettype wire
```

This example takes advantage of the SystemVerilog rule that **`input`** (and **`inout`**) ports left undeclared or declared as **`logic`** will default to a net type of the type specified by the **`` `default_nettype``** compiler directive, or **`wire`** if no default net type has been specified. The example above sets the default net type to **`uwire`** at the beginning of the module, and changes it back to **`wire`** at the end of the module, so as to not impact any subsequent code that might have been modeled, assuming the normal default net type.

> **SystemVerilog Advantage 16 —** The **`uwire`** net type enforces the design intent of only having single-source logic. This can prevent subtle, difficult to detect and to debug design errors. (The **`uwire`** net type was actually added in the Verilog-2005 standard, not the SystemVerilog standard.)
>
> ***Recommendation —*** Once supported by your synthesis compiler, make **`uwire`** the default net type at the beginning of each module, and use **`uwire`** instead of **`wire`** when explicit nets are declared. Explicitly use the **`wire`** or **`tri`** net type only when it is intended to have multiple drivers.

Note: At the this paper was written, this important net type was also not supported by VCS.

12.2 Foreach loops

The **foreach** loop is used to iterate through array elements. A **foreach** will automatically declare its loop control variables, automatically determine the starting and ending indices of the array, and automatically determine the direction of the indexing (count up, or count down). The following examples contrast iterating through a two-dimensional array using traditional Verilog, SystemVerilog with array query methods (see section 2.5.5) and the SystemVerilog **foreach** loop.

The array declaration used in these three examples is:

```
logic [31:0] LUT [0:7][0:255];
```

Example 1: Iterating through a 2-dimensional array with traditional Verilog. This style hard codes the nested **for** loop boundaries. If the design specification for the array were to change in mid project (as if that ever happens) and the **for** loops were not updated, or were updated incorrectly, a functional bug would be introduced that could be difficult to detect and debug.

```
for (int i=0; i<=7; i=i+1)
  for (int j=0; i<=255; j=j+1)
    ... // do something with LUT[i][j]
```

Example 2: Iterating through a 2-dimensional array using SystemVerilog array query methods. This style is scalable and reusable, but can be more difficult to write, and more difficult to debug a coding error.

```
for (i=$right(LUT,1); i<=$left(LUT,1); i=i+$increment(LUT,1))
  for (j=$right(LUT,2); i<=$left(LUT,2); j=j+$increment(LUT,2))
    ... // do something with LUT[i][j]
```

Example 3: Iterating through a 2-dimensional array with the SystemVerilog **foreach** loop. Note that the `i` and `j` variables are not declared — the **foreach** loop automatically declares these variables internally.

```
foreach ( LUT [i,j] )
  ... // do something with LUT[i][j]
```

After examining these three examples, which one would you want to write, debug, maintain, and reuse? The **foreach** loop is supported by VCS, and should be synthesizable in DC and Synplify-Pro.

> **SystemVerilog Advantage 17 —** The **foreach** loop can simplify RTL code for working with arrays and can reduce the risk of coding errors, because **foreach** automatically matches the array.

12.3 Task/function inputs with defaults

SystemVerilog allows the formal inputs of a task or function to be specified with default values.

```
function int incrementor(int count, step=1);
  return (count + step);
endfunction
```

Task/function inputs with a default value do not have to be passed an actual value when the task or function is called. Below are two calls to the `incrementor` function above. The first call only passes a value to the `count` formal input, and not to the `step` input. The second call overrides the default value of `step`.

```
always @(posedge master_clock)
  result = incrementor( .count(data_bus) );  // use default step value

always @(posedge slave_clock)
  result = incrementor( .count(data_bus), .step(2) ); // pass in step value
```

At the time this paper was written, DC did not support this feature.

12.4 Task/function ref arguments

The original Verilog language allows tasks and functions to reference external, module-level signals that were not passed into the task or function. External references to `master_clk` and `data` are shown in the example below. (The examples in this section do not adhere to all synthesis coding requirements, in order to focus on the specific SystemVerilog feature of external signal references).

```
module fib_gen ( input  logic        master_clk, start,
                 output logic [15:0] data );

  always_ff @(posedge start) begin
    fibonacci; // master_clk and data are not passed to the task
  end

  task automatic fibonacci (); // task does not have inputs or outputs
    logic [15:0] a=0, b=1;
    for (int i=0; i <= 31; i++) begin
      @(posedge master_clk) data = a + b;
      a = b;
      b = data;
    end
  endtask: fibonacci
endmodule: fib_gen
```

External, module-level references from within a task or function are synthesizable. This capability is essential when tasks use a clock. In order to see changes on the clock, the task must see the module-level clock signal. If the clock were passed in as an input, only the value of the clock at the time the task was called would be passed in. The local input value would not change when the external clock changed. Similarly, any value changes made within the task must be made to the module-level signals. If the task passed values back as output arguments, only the final value when the task exits would be passed out.

The problem with external references is that it makes code less reusable because the external signals are hard coded within the task or function. SystemVerilog fixes this limitation. Formal arguments of a task or function can be declared as **`ref`** instead of **`input`**, **`inout`** or **`output`**. A **`ref`** argument becomes a reference to the external actual argument in the call to the task or function.

```
module fib_gen (
  input  logic        master_clk, start,
  output logic [15:0] data );

  always_ff @(posedge start) begin
    fibonacci(master_clk, data); // master_clk and data are passed to task
  end

  task automatic fibonacci (ref clk, ref [15:0] val);
    logic [15:0] a=0, b=1;
    for (int i=0; i <= 31; i++) begin
      @(posedge clk) val = a + b;      // task uses local names
      a = b;
      b = val;
    end
  endtask: fibonacci
endmodule: fib_gen
```

Using **`ref`** arguments allows a task or function to use local names, rather than hard coded external names, making the task or function more self-documenting, as well as easier to maintain and reuse.

12.5 Set membership operator (inside) with expressions

The **inside** set membership operator compares a value to a list of other values enclosed in { }. The operator is synthesizable (see section 5.2.2), but there are three useful capabilities of this operator that DC and Synplify-Pro do not support:

The list of values can be expressions, such as other variables or nets.

```
if (data inside {a, b, c}) ...  // if data matches the value of a, b or c
```

The list of values can be stored in an array.

```
int PRIMES [8] = '{2,3,5,7,11,13,17,19};

if (data inside {PRIMES}) ...   // if data matches a value in PRIMES array
```

The operator can be used in continuous assignments.

```
assign at_boundary = (data inside {0, 1023});
```

12.6 Package chaining

Complex designs will often utilize several packages. One package can import definitions from another package. The imported items, however, are not automatically visible outside of that package. Consider the following example:

```
package base_types;
  typedef logic [31:0] bus32_t;
  typedef logic [63:0] bus64_t;
endpackage: base_types

package alu_types;
  import base_types::*;     // wildcard import items from another package

  typedef enum logic [3:0] {ADD, SUB, ...} opcode_t;

  typedef struct {
    bus32_t i0, i1;
    opcode_t opcode;
  } instr_t;
endpackage: alu_types

module alu
  import alu_types::*;
  (input  instr_t instruction,     // OK: instr_t has been imported
   output bus64_t result );        // ERROR: bus64_t has not been imported
  ...
endmodule
```

In order for module `alu` to use definitions from both packages, both packages need to be imported into `alu`. SystemVerilog also has the ability to "chain" packages, so that a module only has to import the last of the packages in a chain, which would be `alu_types` in the preceding example. Package chaining is done using a combination of a package **import** and **export** statements.

```
package alu_types;
  import base_types::*;     // wildcard import items from another package
  export base_types::*;     // export (chain) the imported items
```

```
    typedef struct {
      bus32_t i0, i1;
      opcode_t opcode;
    } instr_t;
  endpackage: alu_types
```

An **`export`** statement can export a specific item, or use a wildcard to export all items imported from another package. Note that, when using wildcards, only the definitions actually used within the package will be exported. In the preceding example, any definitions in `base_types` that were not used within `alu_types` will not be chained.

12.7 Extern module declarations

To facilitate separate file compilation with the dot-star shortcut (see section 7), 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 read in that module or interface.

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

Note: At the time this paper was written, Synplify-Pro supported **`extern`** definitions, but DC did not.

12.8 Configurations

Configurations have been part of Verilog since the Verilog-2001 standard. Configurations define a set of rules that explicitly specify where source code can be found for each module instance in a design. At the time this paper was written, Synplify-Pro supported single configuration blocks, and DC did not support configurations at all. The authors feel both tools should support configurations to the same extent as VCS.

12.9 User-defined net types and generic net types

The SystemVerilog-2012 standard adds the ability for designers to define abstract net types that can communicate more information than just value and strength. A primary usage is to allow a netlist to connect instances of modules that have analog ports, such as UPF power ports. SystemVerilog-2012 also adds a generic net type that can assume the net type of a lower-level module port to which the generic net is connected. Synthesis needs to support these new capabilities, so that, when the synthesis tool writes out a netlist, it can contain the correct interconnections for UPF power types and **`wreal`** analog signals.

13. Summary

SystemVerilog is synthesizable! When the IEEE extended the Verilog-2001 standard, one of the primary goals was to make it possible for design engineers to:

- Be more productive by being able to model more functionality in fewer lines of code. Many of the SystemVerilog constructs that have been discussed in this paper achieve that goal. Structures, user-defined types, and array assignments are examples of how SystemVerilog can significantly reduce the amount of code needed to represent complex functionality.
- Clearly indicate design intent. Specialized procedural blocks, programming statements and decision modifiers allow engineers to state intention. Software tools can then verify if that intent is correctly represented in the RTL code.

- More accurately align simulation behavior to the way synthesis tools realize that functionality in gates. Constructs such as enumerated types, **`case`**...**`inside`** and the **`priority`**, **`unique0`** and **`unique`** ensure that RTL simulation and gate-level implementation closely correspond.
- Enable design reuse through scalable, configurable models. SystemVerilog constructs like interfaces, array assignments, context-aware literal values and many others help to achieve this goal.

These benefits of SystemVerilog enable design engineers to rapidly develop RTL code, more easily maintain that code, and minimize the classic Verilog gotchas, where the RTL code simulates differently than the synthesized gate-level design.

This paper listed several advantages for designing with, and synthesizing SystemVerilog: Substantial benefits can be gained from using SystemVerilog in synthesizable RTL code by following these guidelines:

1. Use **`logic`** for modules ports and most internal signals – forget **`wire`** and **`reg`**.
2. Use the **`uwire`** net type to check for and enforce single-driver logic.
3. Use *enumerated types* for variables with limited legal values.
4. Use *structures* to collect related variables together.
5. Use *user-defined types* to ensure consistent declarations in a design.
6. Use *packages* for declarations that are shared throughout a design.
7. Use **`always_comb`**, **`always_latch`** and **`always_ff`** procedural blocks.
8. Use **`case`**...**`inside`** instead of **`casez`** and **`casex`**.
9. Use **`priority`**, **`unique0`**, **`unique`** in place of the full_case, parallel_case pragmas.
10. Use **`priority`**, **`unique0`**, **`unique`** with **`if`**...**`else`** when appropriate
11. Use **`void`** functions instead of tasks in RTL code
12. Use *dot-name* and *dot-star* netlist shortcuts.
13. Use *interfaces* to group related bus signals, functions (if appropriate) and assertions.
14. Use **`` `begin_keywords``** to specify the language version used.
15. Use a locally declared **`timeunit`** instead of **`` `timescale``**.

14. Acknowledgements

The authors express their appreciation to Will Cummings, Bernard Miller, Tim Schneider, Sivaramalingam Palaniappan, and others at Synopsys who provided technical support and comments. We also thank Jason Rziha of Microchip and LeeAnn Sutherland of Sutherland HDL for their careful reviews of the paper drafts. The efforts of these people have helped to ensure that this paper is complete and accurate.

15. References

[1] *"1364-1995 IEEE Standard Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 1995. ISBN:.1-55937-727-5

[2] *"1364-2001 IEEE Standard Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2001. ISBN: 0-7381-2826-0.

[3] *"1364.1-2002 IEEE Standard for Verilog Register Transfer Level Synthesis"*, IEEE, Pascataway, New Jersey. Copyright 2002. ISBN: 0-7381-3501-1.

[4] *"P1364-2005 Draft Standard for Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN: 0-7381-4850-4.

[5] *"1800-2005 IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN: 0-7381-4811-3.

[6] *"1800-2009 IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language"*, IEEE, Pascataway, New Jersey. Copyright 2009. ISBN: 978-0-7381-6129-7.

[7] *"1800-2012 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language"*, IEEE, Pascataway, New Jersey. Copyright 2013. ISBN: 978-0-7381-8110-3 (PDF), 978-0-7381-8111-0 (print).

[8] "*HDL Compiler™ for SystemVerilog User Guide, Version G-2012.06-SP4*", Synopsys, December 2012, https://solvnet.synopsys.com.

[9] "*Synopsys FPGA Synthesis Reference Manual*", Synopsys, December 2012, https://solvnet.synopsys.com.

[10]"*Synopsys FPGA Synthesis User Guide*", Synopsys, December 2012, https://solvnet.synopsys.com.

[11]"*The Benefits of SystemVerilog for ASIC Design and Verification, Version 2.5*", Synopsys, January 2007, https://solvnet.synopsys.com.

[12]Sutherland, Davidmann and Flake, *"SystemVerilog for Design Engineers, second edition"*, Springer, Boston, Massachusetts. Copyright 2006. ISBN: 978-0-387-36495-7.

[13]Pieper, *"SystemVerilog from a Synthesis Perspective"*, Design and Verification Conference (DVCon), San Jose, California, March 2004.

[14]Mills and Cummings, "*RTL Coding Styles That Yield Simulation and Synthesis Mismatches*", Synopsys Users Group (SNUG) conference, San Jose, California, March 1999.

[15]Cummings, *" 'full_case parallel_case', the Evil Twins of Verilog Synthesis*", Synopsys Users Group (SNUG) conference, Boston, Massachusetts, 1999.

[16]Sutherland, *"SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes"*, Synopsys Users Group (SNUG) conference, San Jose, California, March 2005.

[17]Mills and Sutherland, *"SystemVerilog Assertions Are For Design Engineers Too!*", Synopsys Users Group (SNUG) conference, San Jose, California, March 2006.

[18]Mills, "*Being assertive with your X (SystemVerilog assertions for dummies)*", Synopsys Users Group Conference (SNUG) San Jose, California, 2004.

[19]Mills, "*Yet Another Latch and Gotchas Paper*", Synopsys Users Group (SNUG) conference, San Jose, California, March 2012.

[20]Sutherland, *"I'm Still in Love with My X!"*, Design and Verification Conference (DVCon), San Jose, California, February 2013.

[21]Greene, Salz and Booth, "X-Optimism Elimination during RTL Verification", Synopsys Users Group Conference (SNUG) San Jose, 2012.