Can My Synthesis Compiler Do That?

What ASIC and FPGA Synthesis Compilers Support in the SystemVerilog-2012 Standard

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

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

ABSTRACT

There seems to be an industry-wide misconception that the Verilog language is used for design and synthesis, and SystemVerilog is only used for verification. The authors have often heard comments from engineers such as, "Our synthesis compiler does not support SystemVerilog", "Our [design] management is afraid of SystemVerilog" and, from a synthesis company's support engineer, "Can our compiler do that?"

The truth is, when the SystemVerilog standard was first conceived in 2002, one of the primary goals was to enable creating synthesizable models of complex hardware designs more accurately and with fewer lines of code. A great deal of SystemVerilog was intended to be used for design and synthesis. That goal was achieved, and all major ASIC and FPGA synthesis compilers support a large portion of SystemVerilog. This paper examines several synthesizable SystemVerilog constructs, 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 designs 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 complex ASIC and FPGA implementations.

Table of Contents

1. Introduction 2
2. Verilog versus SystemVerilog 2
3. Procedural blocks 3
4. Simplified net and variable declarations 4
5. Vectors with subfields 5
6. Enumerated types 6
7. Copying arrays 7
8. Structures 8
9. Packages 10
10. Decision operators and statements 11
    10.1 Wildcard decisions (wildcard equality) 12
    10.2 Set membership operator (inside) 12
    10.3 Case...inside decisions 12
    10.4 Unique, unique0 and priority decision checks 13
11. Streaming operators 14
12. Casting 15
13. Tasks and functions 17
14. Interfaces 18
15. Vector fill tokens 19
16. Expression size functions 20
17. Synthesis compiler support for SystemVerilog 21
18. Additional synthesizable SystemVerilog constructs 21
19. Summary 22
20. Conclusion 23
21. Acknowledgements 23
22. References 23
Appendix A: Table of synthesis compiler support 24

1.0 Introduction

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. ***This is not true!*** Much of SystemVerilog is synthesizable with today's ASIC and FPGA synthesis compilers. This paper poses several questions, and shows why it is important to use SystemVerilog for synthesis.

1. Can my synthesis compiler actually tell me when I have the wrong type of logic in my RTL models?
2. Can my synthesis compiler infer nets and variables without my having to worry about where to use `reg` and where to use `wire`?
3. Can my synthesis compiler work with bytes of a vector without using complex and cryptic vector part selects?
4. Can my synthesis compiler prevent me from assigning stupid values to variables?
5. Can my synthesis compiler copy an entire look-up table or register file in a single statement?
6. Can my synthesis compiler use 4 lines of code to replace 216 lines of code?
7. Can my synthesis compiler handle defining functions, constants, and other declarations in just one place, and using those definitions in multiple modules?
8. Can my synthesis compiler help me make better decisions?
9. Can my synthesis compiler reverse all the bits or bytes of my parameterized-width data word in one operation?
10. Can my synthesis compiler eliminate false warning messages about intentional size and type conversions, and only warn about unintentional conversions?
11. Can my synthesis compiler prevent me from writing tasks (subroutines) that simulate, but won't synthesize?
12. Can my synthesis compiler support modeling standard bus protocol signals at a higher level of abstraction that avoids code duplication?
13. Can my synthesis compiler fill vectors of any size and with parameterized widths with all 1s?
14. Can my synthesis compiler automatically calculate the size of my address bus, based on the depth of my memory storage?

The goal for this paper is to help engineers and engineering managers understand the advantages of using SystemVerilog for RTL design and synthesis, rather than limit (or handcuff) themselves to the thirteen-year-old Verilog-2001 language. Towards this end, this paper presents only a few of the SystemVerilog constructs that are synthesizable, and that offer significant advantages over traditional Verilog. This paper does not discuss every synthesizable SystemVerilog feature, and does not cover the synthesizable constructs in traditional Verilog.

2. Verilog versus SystemVerilog

Verilog was never just a design and synthesis language. Verilog has always been a dual-purpose language to be used to both model hardware functionality and to describe verification testbenches. Verilog synthesis compilers only support a subset of the Verilog language, the portion that can be represented in actual silicon. It is a misconception that Verilog is strictly for design and synthesis.

The original Verilog language was a simple language first introduced in the early 1980s. Design size and complexity have grown at a tremendous rate since the Verilog language was created, and, although Verilog was a very capable language for modeling the size, complexity and speed of silicon chips being designed in the 1980s and 1990s, the complexity of designs in the 2000s require a much more capable modeling and verification language.

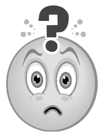
The confusing name change... In 2002, a standards committee was formed to extend the capabilities of Verilog. (One of the authors of this paper was a founding member of this committee.) The new features being added were so substantial that this standards committee decided

©2014, Sutherland HDL, Inc.

to give Verilog a new name, ***SystemVerilog***. (The standards committee also considered “Verilog++” as a new name, but felt “SystemVerilog” would have more market appeal.) In 2009, the IEEE officially terminated support for the earlier standards that used the Verilog name, and now only uses the SystemVerilog name. ***Whether you knew it or not, since 2009, you have not been using Verilog for your design projects...you have been designing with—and synthesizing—SystemVerilog!***

This paper assumes that readers are already familiar with RTL modeling and synthesis with traditional Verilog, and focuses on synthesizable constructs that SystemVerilog adds to traditional Verilog (the IEEE 1364-2001 Verilog standard).

3. Procedural blocks



Question 1 — Can my synthesis compiler actually tell me when I have the wrong type of logic in my RTL models?

With SystemVerilog it can!

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 with the **`always`** keyword. Instead, these tools can only interpret the code within the procedural block, and then “*infer*” — which is just a nice way to say “*guess*” — the engineer’s intention.

Easy-to-make coding errors in combinational logic can generate latches with no indication that latch behavior exists until the synthesis result reports are examined. Design flaws could occur, due to an engineer missing a latch inference message in lengthy synthesis reports. Example 3.a illustrates a Verilog-style always procedure intended to represent combinational logic, but does it really?

```
module adder_subtractor
(input  wire mode,
 input  wire [7:0] a, b,
 output reg  [7:0] adder_out, subtractor_out
);
  always @(mode) begin
    if (mode) adder_out = a + b;
    else subtractor_out = a - b;
  end
endmodule //adder_subtractor
```

Example 3.a: Verilog combinational always block (with bug)

Despite the comment indicating the designer’s intent was for combinational logic, there are multiple problems with the code within this always block.

1. The sensitivity list is incomplete. This will simulate with partial latched behavior, but synthesize as combinational logic.
2. The assignment to `adder_out` is not complete because `adder_out` is only updated when the true branch of the **`if`** statement is executed. This will both simulate and synthesize with latched behavior.
3. The assignment to `subtractor_out` is not complete because `subtractor_out` is only updated when the false (**`else`**) branch of the **`if`** statement is executed. This will both simulate and synthesize with latched behavior.

SystemVerilog adds three new types of RTL procedural blocks that document intent and also provide some synthesis rule checking. These are: **`always_comb`**, **`always_latch`**, and **`always_ff`**. These procedural blocks offer several important advantages for modeling synthesizable designs:

- Simulators, lint checkers, and synthesis compilers can issue warnings if the code modeled within these new procedural blocks does not match the designated type.
- Correct sensitivity lists can be either inferred or checked because the functional intent of the procedural blocks is known.
- Certain synthesis requirements can be syntactically enforced that cannot be enforced in a general purpose **`always`** procedure, because the general purpose procedure is also used for verification code and models not intended to be synthesized. One important synthesis rule that is enforced is that any variable assigned in a **`always_comb`**, **`always_latch`**, or **`always_ff`** procedure cannot be assigned a value anywhere else.

The **`always_comb`** procedural block indicates that the designer’s intent is to represent combinational logic. The following example illustrates a correct usage of a SystemVerilog **`always_comb`** procedural block (the **`logic`** type used in this example is discussed in Section 4):

```
module mux8
(input  logic sel,
 input  logic [7:0] a, b,
 output logic [7:0] y
);
  always_comb begin
    if (sel) y = a;
    else     y = b;
  end
endmodule: mux8
```

Example 3.b: SystemVerilog always_comb procedural block

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

©2014, Sutherland HDL, Inc.

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. (Even Verilog's **always_@*** will sometimes infer an incomplete sensitivity list.) This is not a syntax error, and results in RTL simulation behaving differently than the post-synthesis gate-level implementation of the design. Inadvertently leaving out a signal from the sensitivity list is an easy mistake to make in large, complex decoding logic.

Another important benefit is that, because software tools know the intent is to represent combinational logic, tools can verify that this intent is being met. Example 3.c shows the same code as in Example 3.a, with the same coding errors, but this time modeled SystemVerilog stye, using **always_comb**. (Note that the incomplete sensitivity list error in Example 3.a cannot be made with **always_comb**.)

```
module adder_subtractor
(input wire mode,
 input wire [7:0] a, b,
 output reg [7:0] adder_out, subtractor_out
);
  always_comb begin
    if (mode) adder_out      = a + b;
    else      subtractor_out = a - b;
  end
endmodule: adder_subtractor
```

Example 3.c: always_comb procedure with latch behavior

When traditional Verilog **always** procedures are synthesized, synthesis compilers cannot tell if inferred latches are intentional or unintentional. Synthesis compilers can only report how many latches were inferred, and not whether the latch was due to a coding error. With SystemVerilog **always_comb** procedures, synthesis compilers can report exactly where any unintentional latches occurred. When the code for Example 3.c was read into one synthesis compiler, the following elaboration warning was issued:

```
Warning: example03c.sv:6: Netlist for always_comb
block contains a latch. (ELAB-974)
```

The **always_latch** procedural block is 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 **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 intent is to have synchronous or asynchronous reset behavior. The clock and reset information must be specified as part of the sensitivity list. Software tools can 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.

```
module reg8
(input  logic       sel,
 input  logic [7:0] a, b,
 output logic [7:0] out
);
  always_ff @(sel, a, b)
    if (sel) out = a;
    else     out = b;
endmodule: reg8
```

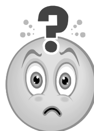
Example 3.d: always_ff procedural block with combinational sensitivity list (no clock)

The warning that is issued by one synthesis compiler is:

```
Warning:  example03d.sv:6: Netlist for always_ff
block does not contain a flip-flop. (ELAB-976)
```

SystemVerilog Advantage 1 — The benefits of using 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.

4. Simplified net and variable declarations



Question 2 — Can my synthesis compiler infer nets and variables without my having to worry about where to use **reg** and where to use **wire**?

With SystemVerilog it can!

Traditional Verilog has two primary types of data, *nets* and *variables*. There are a number of specific net data type keywords and variable keywords, but the two used almost exclusively for ASIC and FPGA design are the **wire** net type and **reg** variable type.

Verilog has strict rules on where net types, such as **wire**, *must* be used, and where they *may* be used, and where variable types, such as **reg**, *must* be used, and where they *may* be used. Learning these rules has always been a challenge, and, even after learning the rules, getting them right in RTL code remains a challenge for many engineers. (eventually many engineers simply learn to recognize the error messages when they use **wire** or **reg** incorrectly.)

The **reg** keyword can also be a source of confusion for engineers first learning Verilog. The word "*reg*" would seem to imply "*register*". A common new-user misconception is that whereever a **reg** variable is used in

©2014, Sutherland HDL, Inc.

an RTL model, synthesis will infer hardware registers. The confusing truth is that '*reg*" does not mean "*register*" and the **reg** data type does not infer hardware registers.

SystemVerilog solves both of these Verilog shortcomings with a new keyword, **logic**. The full meaning of this new type is beyond the scope of this paper. Its usage is what is important. When module ports or signals internal to a module are declared as a **logic** type, simulation and synthesis compilers will automatically infer a net or variable based on context. Correctly choosing **wire** or **reg** becomes the job of the compiler instead of the design engineer!

SystemVerilog Advantage 2 — You no longer need to worry about when to declare module ports or local signals as **wire** or **reg**. With SystemVerilog, you can declare all module ports and local signals as **logic**, and software tools will correctly infer nets or variables for you.

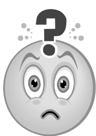
Example 4.a illustrates this advantage using a model of a simple RTL adder.

```
module adder16
(input  logic [15:0] a, b, // infers wire nets
 input  logic        ci,   // infers wire net
 output logic [15:0] sum,  // infers variables
 output logic        co    // infers variable
);
  always_comb
    {co,sum} = a + b + ci;
endmodule: adder16
```

Example 4.a: SystemVerilog automatic net and variable types

Note: At the time this paper was written, some synthesis compilers incorrectly inferred a variable type for input and inout ports instead of a wire type. This incorrect inference prevents an inout port from having multiple drivers.

5. Vectors with subfields



Question 3 — Can my synthesis compiler work with bytes of a vector without using complex and cryptic vector part selects?

With SystemVerilog it can!

In traditional Verilog, 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. Figure 1 shows the declaration of a 32-bit vector with bit 0 as the least-significant bit.

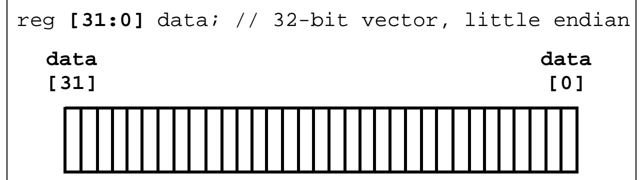


Figure 1: Verilog style 32-bit vector (no subfields)

This traditional Verilog style vector declaration accurately models hardware vectors, and works well, if the RTL design code primarily uses the entire vector or individual bits of the vector. It is more difficult in traditional Verilog, however, if the RTL design code frequently references subfields out of the vector, such as to read or write specific bytes of the vector.

Example 5.a shows the traditional Verilog way to read a specific byte out of a vector. *This example has a functional bug* — one that is easy to make with Verilog. The example will synthesize without any indication of a problem, but will not function as intended. Can you spot the bug?

```
module add_third_byte
(input  wire [31:0] a, b,
 output reg  [ 7:0] sum,
 output reg         co
);
  always @(a, b)
    {co,sum} = a[24:17] + b[24:17];
endmodule //add_third_byte
```

Example 5.a: Byte select using Verilog (with a bug)

The not-so-obvious bug is that the design specification calls for selecting byte 3 from the 32-bit vector, but the bits of the part select incorrectly select an 8-bit group that is partially from the third byte and partially from the fourth byte.

SystemVerilog adds the ability to declare vectors that have numbered subfields of a specific bit width. Figure 2 show a declaration of a 32-bit vector with eight 8-bit subfields.

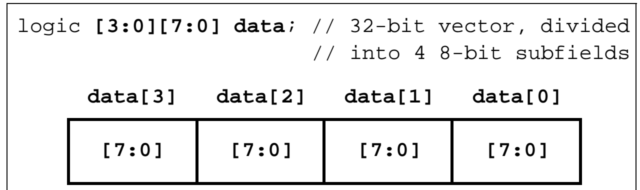


Figure 2: SystemVerilog style 32-bit vector with subfields

The four 8-bit subfields are numbered from 0 to 3, with subfield 0 being the right-most byte of the vector. The faulty Verilog code in Example 5.a can be modeled more easily — and without the subtle bug — in SystemVerilog, as shown in Example 5.b.

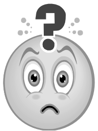
©2014, Sutherland HDL, Inc.

```
module add_third_byte
(input  logic [3:0][7:0] a, b, // 32-bit vectors
 output logic [7:0]      sum,
 output logic            co
);
  always_comb
    {co,sum} = a[2] + b[2];  // subfield selects
endmodule: add_third_byte
```

Example 5.b: Byte select using SystemVerilog

SystemVerilog Advantage 3 — Designs that frequently manipulate parts of a vector are simpler to model, more self-documenting, and correct by construction.

6. Enumerated types



Question 4 — Can my synthesis compiler prevent me from assigning stupid values to variables?

With SystemVerilog it can!

Traditional Verilog nets and variables are loosely typed. Loosely typed nets and variables have built-in conversion rules that allow a value that is out-of-range, or of a different type, to be assigned. There is no checking of the value being assigned; the value is simply converted to the data type of the net or variable.

Example 6.a shows code for a small state machine modeled with traditional Verilog. The code is syntactically correct, and will synthesize, but, due to Verilog's loosely typed behavior, there are six functional bugs that will end up in the gate-level implementation of the state-machine. (Some synthesis compilers might issue a warning for some of the bugs, but the warnings are not fatal, and synthesis will implement the bugs in the gate-level design.)

```
module controller
(input  wire      clock, resetN,
 output reg [2:0] control);

  // Names for state machine states (one-hot)
  parameter [2:0] WAIT = 3'b001,
                  LOAD = 3'b010,
                  DONE = 3'b001;       // BUG 1

  // Names for control output values
  parameter [1:0] READY = 3'b101,
                  SET   = 3'b010,
                  GO    = 3'b110;      // BUG 2

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

  // State Sequencer
  always @(posedge clock or negedge resetN)
    if (!resetN) state <= 0;          // BUG 3
    else         state <= n_state;

  // Next State Decoder (sequentially cycle
  // through the three states)
  always @(state)
    case (state)
      WAIT: n_state = state+1;
      LOAD: n_state = state+1;        // BUG 4
      DONE: n_state = state+1;        // BUG 5
    endcase

  // Output Decoder
  always @(state)
    case (state)
      WAIT: control = READY;
      LOAD: control = SET;
      DONE: control = DONE;           // BUG 6
    endcase
endmodule //controller
```

Example 6.a: Verilog state machine with 6 functional bugs

SystemVerilog adds *enumerated type* nets and variables that are more strongly typed. Enumerated types allow nets and variables to be defined with a specific set of named values, referred to as labels. Designers can specify explicit values for each label.

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 variable from the same enumerated definition.

The stronger rules of enumerated types provide significant advantages over traditional Verilog. Example 6.b is the same state machine as the one previously shown in Example 6.a, but this time modeled using SystemVerilog enumerated types. Example 6.b has the same six bugs, but, with enumerated types, all six bugs become syntax errors. Neither simulators nor synthesis compilers will not allow this faulty state machine to be implemented in gates until these bugs are fixed.

```
module controller
(input  logic       clock, resetN,
 output logic [2:0] control_out
);

// Enumerated variables for fsm states
// One-hot encoding for the labels
enum logic [2:0] {WAIT  = 3'b001,
```

```
                     LOAD  = 3'b010,
                     DONE  = 3'b001} // SYNTAX ERROR
                     state, n_state;

// Enumerated variables for control output values
enum logic [1:0] {READY = 3'b101,
                  SET   = 3'b010,
                  GO    = 3'b110} // SYNTAX ERROR
                  control;

  assign control_out = control;

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

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

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

Example 6.b: SystemVerilog style state machine with 6 bugs

SystemVerilog Advantage 4 — Enumerated types can prevent coding errors that could be difficult to detect and debug!

7. Copying arrays

Question 5 — Can my synthesis compiler copy an entire look-up table or register file in a single statement?

With SystemVerilog it can!

Traditional Verilog supports one-dimensional and multi-dimensional arrays, which, in RTL models, can be used to represent look-up tables and register files. However, traditional Verilog only permits access to a single element of an array at a time. To copy data from one array to another array requires writing loops that index through each element of each array.

Example 7.a illustrates copying the contents of a 3-dimensional register file into a second register file. Each register file stores 16 8-bit values for red, green, and blue, giving a total of 48 bytes.

The restriction in Verilog that arrays can only be copied one element at a time also makes it difficult to pass an array through module ports. Passing signals through ports is a form of a continuous assignment. Since arrays can only be assigned one element at a time, to pass the array through the module port it is necessary to either:

- Create a separate port for each element of the array.
- Create a vector port that is wide enough to contain all elements of the array as if the array were concatenated together.

Creating separate ports for each array element is simple, but only practical for very small arrays. Defining a vector port wide enough to represent the entire array reduces the number of ports, but requires extra programming. An output port require using loops to iterate through the array and assigning each element of the array to a part-select of the port vector, often referred to as "packing" the array into the vector. An input port requires using loops to iterate the array and unpacking the vector by assigning vector part selects to the corresponding array elements.

Example 7.a. shows that, even for a simple design with a small array, the code to copy arrays and pass an array through a port is complex and convoluted.

```
module rgb_frame_collector
(input  wire [7:0]  red, green, blue,
 input  wire        clk, rstn,
 output reg         frame_ready,
 // NEXT LINE IS A VECTOR PORT TO PASS AN ARRAY
 output reg [383:0] rgb_frame_out
);
  // internal arrays (16x3 arrays of bytes)
  reg [7:0] rgb_frame1 [0:15][0:2];
  reg [7:0] rgb_frame2 [0:15][0:2];

  // internal temporary variables
  reg [3:0] frame_addr;
  integer   i, j;

  // load arrays
  always @(posedge clk, negedge rstn)
    if (!rstn) begin
      frame_addr <= 0;
      frame_ready = 0;
      // RESET ARRAYS USING LOOPS:
      for (i=0; i<=15; i=i+1) begin
        for (j=0; j<=2; j=j+1) begin
          rgb_frame1[i][j] <= 0;
          rgb_frame2[i][j] <= 0;
        end
      end
    end
    else begin
      // as red/green/blue bytes come in, store
      // them in the rgb_frame1 array
```


©2014, Sutherland HDL, Inc.

```
      frame_ready <= 0;
      rgb_frame1[frame_addr][0] <= red;
      rgb_frame1[frame_addr][1] <= green;
      rgb_frame1[frame_addr][2] <= blue;
      frame_addr <= frame_addr + 1;

      // COPY ARRAYS USING LOOPS
      // frame1 is is full when frame_addr wraps
      // to 0; copy frame1 array to frame2 array
      if (~|frame_addr) begin
        for (i=0; i<16; i=i+1) begin
          for (j=0; j<3; j=j+1) begin
            rgb_frame2[i][j] <= rgb_frame1[i][j];
          end
        end
        frame_ready <= 1;
      end
    end

  // PACK ARRAY INTO VECTOR PORT
  always @(*) begin
    for (i=0; i<=15; i=i+1) begin
      for (j=0; j<=2; j=j+1) begin
        rgb_frame_out[383-(i*24+(j*8))-:8] =
          rgb_frame2[i][j];
      end
    end
  end
endmodule //rgb_frame_collector
```

Example 7.a: Verilog style register file copy — using loops

The complex part selects used in the last few lines of Example 7.a above are difficult to write, and an easy place to make a subtle coding error that is time consuming to debug.

SystemVerilog allows entire arrays to be copied as a single assignment statement. Array copy assignments require that both sides of the assignment statement have the same number of dimensions and the same number of elements in each dimension. The number of bits of each element must also be the same size and of compatible data types.

SystemVerilog also supports assigning a list of values into an array as a single assignment, which is a variation of an array copy. The list of values is enclosed in `'{ }`. The list can contain separate values for each array element, or a default value for the entire array.

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 contents of an entire look-up table or register file can be passed from one module to another module using a single port.

Example 7.b illustrates copying the 3-dimensional RGB register file using a single assignment statement, instead of several lines of code and nested loops. Array copy and array ports makes this example shorter and much less complicated that the traditional Verilog code in Example 7.a.

```
module rgb_frame_collector
(input  logic [7:0] red, green, blue,
 input  logic       clk, rstn,
 output logic       frame_ready,
 // NEXT LINE IS AN ARRAY PORT
 output logic [7:0] rgb_frame_out [0:15][0:2]
);
  // internal array (16x3 arrays of bytes)
  logic [7:0] rgb_frame1 [0:15][0:2];

  // internal temporary variables
  logic [3:0] frame_addr;

  // load arrays
  always_ff @(posedge clk, negedge rstn)
    if (!rstn) begin
      frame_addr  <= 0;
      frame_ready <= 0;
      // RESET ARRAYS USING ARRAY ASSIGNMENT
      rgb_frame1    <= '{default:0};
      rgb_frame_out <= '{default:0};
    end
    else begin
      // as red/green/blue bytes come in, store
      // them in the rgb_frame1 array
      frame_ready <= 0;
      rgb_frame1[frame_addr] <='{red,green,blue};
      frame_addr <= frame_addr + 1;

      // COPY ARRAYS USING ARRAY ASSIGNMENT
      // frame1 is is full when frame_addr wraps
      // to 0; copy frame1 array directly to
      // rgb_frame_out port
      if (~|frame_addr) begin
        rgb_frame_out  <= rgb_frame1;
        frame_ready <= 1;
      end
    end
endmodule: rgb_frame_collector
```

Example 7.b: SystemVerilog style register file copy — 1 line

SystemVerilog Advantage 5 — Array copy assignments can significantly reduce the complexity of design code for moving blocks of data from one array to another. Array assignments also do not require changing any code if the sizes of the arrays change.

8. Structures

Question 6 — Can my synthesis compiler use 4 lines of code to replace 216 lines of code?

With SystemVerilog it can!

Traditional Verilog models functionality at the individual signal level. While this accurately represents the signals in

©2014, Sutherland HDL, Inc.

the silicon, it can require many lines of code to work with each individual signal.

Example 8.a models a packet processor that operates on a UNI data cell. The module receives a UNI cell on an input port, operates on the payload of the cell, and passes the resultant data through a UNI cell output port.

A UNI cell is comprised of 54 signals:

- GFC - Generic Flow Control, 4 bits
- VPI - Virtual Path Identifier, 8 bits
- VCI - Virtual Channel Identifier, 16 bits
- PTI - Payload Type Indication, 3 bits
- CLP - Cell Loss Priority, 1 bit
- HEC - Header Error Control, 8 bits
- Payload - Payload, array of 48 bytes

With traditional Verilog, the UNI cell input requires 54 input ports; a separate port for each signal within the packet. The UNI cell output requires another 54 ports. The UNI register that stores the operation result must be modeled as 54 registers, one for each signal in the UNI cell. Resetting the register requires 54 assignment statements, and registering the operation results requires another 54 assignment statements. This makes a total of 216 statements to model the ports and internal register assignments. Example 8.a only shows a portion of those 216 lines of code.

```
module uni_atm
(input         clk, rstN,
 input  wire [ 3:0] GFC_in,
 input  wire [ 7:0] VPI_in,
 input  wire [15:0] VCI_in,
 input  wire [ 2:0] PTI_in,
 input  wire        CLP_in,
 input  wire [ 7:0] HEC_in,
 input  wire [ 7:0] Payload00_in,
 input  wire [ 7:0] Payload01_in,
 input  wire [ 7:0] Payload02_in,
 // Payload  3 through 44 not shown //
 input  wire [ 7:0] Payload45_in,
 input  wire [ 7:0] Payload46_in,
 input  wire [ 7:0] Payload47_in,

 output reg  [ 3:0] GFC_out,
 output reg  [ 7:0] VPI_out,
 output reg  [15:0] VCI_out,
 output reg  [ 2:0] PTI_out,
 output reg         CLP_out,
 output reg  [ 7:0] HEC_out,
 output reg  [ 7:0] Payload00_out,
 output reg  [ 7:0] Payload01_out,
 output reg  [ 7:0] Payload02_out,
 // Payload bytes 3 through 44 not shown //
 output reg  [ 7:0] Payload45_out,
 output reg  [ 7:0] Payload46_out,
 output reg  [ 7:0] Payload47_out
);
  // Only the UNI cell output register is shown.
  // The code that operates on the input UNI cell
  // is not shown. For simplicity, this example
  // just registers the input UNI cell without
  // doing any processing.
  always @(posedge clk, negedge rstN)
    if (!rstN) begin
      GFC_out       <= 4'b0;
      VPI_out       <= 8'b0;
      VCI_out       <= 16'b0;
      PTI_out       <= 3'b0;
      CLP_out       <= 1'b0;
      HEC_out       <= 8'b0;
      Payload00_out <= 8'b0;
      Payload01_out <= 8'b0;
      Payload02_out <= 8'b0;
      // Payload 3 through 44 not shown //
      Payload45_out <= 8'b0;
      Payload46_out <= 8'b0;
      Payload47_out <= 8'b0;
    end
    else begin
      GFC_out       <= GFC_in;
      VPI_out       <= VPI_in;
      VCI_out       <= VCI_in;
      PTI_out       <= PTI_in;
      CLP_out       <= CLP_in;
      HEC_out       <= HEC_in;
      Payload00_out <= Payload00_in;
      Payload01_out <= Payload01_in;
      Payload02_out <= Payload02_in;
      // Payload 3 through 44 not shown //
      Payload45_out <= Payload45_in;
      Payload46_out <= Payload46_in;
      Payload47_out <= Payload47_in;
    end
endmodule //uni_atm
```

Example 8.a: Verilog style for a UNI packet register

The 54 signals that make up a UNI cell could also be represented as a single vector that is 424 bits wide. Part-selects of the vector would then be used to reference specific signals of the UNI cell. This approach simplifies the number of ports required to transfer a UNI cell through module ports, but using part selects from a large vector for each UNI signal can make the code more difficult to write and maintain. The additional code to unpack the 424-bit input vector and to repack the operation results into a 424 bit vector would provide little, if any, advantage over the separate port declarations shown in Example 8.a.

SystemVerilog adds a more abstract and powerful hardware modeling construct called a `structure`. Structures are used to bundle related signals together under a common name. A SystemVerilog structure is analogous to a structure in the C language, or a record in VHDL.

Example 8.b shows how the 54 UNI ATM cell signals can be bundled together as a SystemVerilog structure:

©2014, Sutherland HDL, Inc.

```
typedef struct {
  logic [ 3:0] GFC; // Generic Flow Control
  logic [ 7:0] VPI; // Virtual Path ID
  logic [15:0] VCI; // Virtual Channel ID
  logic [ 2:0] PTI; // Payload Type Indication
  logic        CLP; // Cell Loss Priority
  logic [ 7:0] HEC; // Header Error Control
  logic [ 7:0] Payload [48]; // Payload
} uni_t;
```

Example 8.b: UNI cell structure definition

The structure definition in Example 8.b is defined as a user-defined type using the **typedef** keyword. User-defined types are another important SystemVerilog extension to traditional Verilog. By defining the UNI structure as a user-defined type called uni_t, the uni_t type can be used for multiple declarations in the same module, and used in multiple modules.

Ports and variables of a uni_t user-defined type are declared the same way as with ports or variables with built-in types. The declaration of a uni_t variable is:

```
  uni_t uni_cell;
```

Individual members of a structure are accessed using a dot operator (**.**), such as:

```
uni_cell.VCI = 16'hDEAD;
```

A powerful aspect of structures is that they 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. Using **typedef** to create a user-defined type based on the structure definition ensures that two structures are of the same definition, and 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.

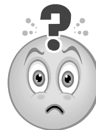
Example 8.c illustrates the same UNI cell register that was shown as traditional Verilog code in Example 8.a, but modeled using the uni_t structure definition in Example 8.b to bundle together the 54 signals that make up a UNI cell. Note how much shorter Example 8.c is than Example 8.a! The structure bundle allows reducing the 54 input ports in Example 8.a to just 1 port. Likewise, the 54 output ports are reduced to a single port, the 54 assignment statements to reset the register are reduced to a single assignment statement, and the 54 assignment statements to store a packet are reduced to a single assignment. Thus, the 216 lines of code for the ports and register in Example 8.a are reduced to 4 lines of code in Example 8.c (plus the 10 lines to declare the UNI cell structure shown in Example 8.b).

```
module uni_atm
(input        clk, rstN,
 input  uni_t uni_cell_in,
 output uni_t uni_cell_out
);
  // Only the UNI cell output register is shown.
  // The code that operates on the input UNI cell
  // is not shown. For simplicity, this example
  // just registers the input UNI cell without
  // doing any processing.
  always_ff @(posedge clk, negedge rstN)
    if (!rstN) uni_cell_out <= '{default:'0};
    else       uni_cell_out <= uni_cell_in;
endmodule: uni_atm
```

Example 8.c: SystemVerilog style for a UNI packet register

SystemVerilog Advantage 6 — By using structures to bundle related signals together, the signals can be assigned and transferred to other modules as a bundle, reducing the lines of code and ensuring consistency.

9. Packages



Question 7 — Can my synthesis compiler handle defining functions, constants, and other declarations in just one place, and using those definitions in multiple modules?

With SystemVerilog it can!

The original Verilog language does not have a shared declaration space. Each module contains all declarations used within that module. This is a major language limitation. If the same parameter, task or function definition is needed in multiple modules, designers have to connive awkward work-a-rounds, typically using a combination of **'ifdef** and **'include** compiler directives. The addition of SystemVerilog user-defined types, such as structures and enumerated types, make the lack of a shared declaration space a more 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 that 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

©2014, Sutherland HDL, Inc.

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[2:0] {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: alu_types
```

Example 9.a: SystemVerilog package

The definitions in a package can be referred to within a module in three different ways, all of which are synthesizable:

- Explicit package references
- Explicit import statements
- Wildcard import statements

Explicit references of a package item use the package name followed by a double colon (**`::`**). For example:

```
alu_types::bus64_t result;
```

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.

Example 9.b illustrates using explicit package item references and explicit package item imports:

```
module alu
(input  alu_types::instr_t instr,
 output alu_types::bus64_t result
);
  import alu_types::ADD, alu_types::SUB;

  always_comb begin
    case(instr.opcode)
      ADD: result = instr.i0 + instr.i1;
      SUB: result = instr.i0 - instr.i1;
    endcase
  end
endmodule: alu
```

Example 9.b: Explicit package reference and import

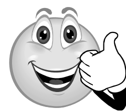
Wildcard imports use an asterisk to represent all definitions within the package. Wildcard imports make all items of the package visible within a module. For example:

```
module alu
import alu_types::*;
(input  instr_t instr,
 output bus64_t result
);

  always_comb begin
    case(instr.opcode)
      ADD: result = instr.i0 + instr.i1;
      SUB: result = instr.i0 - instr.i1;
    endcase
  end
endmodule: alu
```

Example 9.c: Wildcard package import

(Note: At the time this paper was written, one synthesis compiler did not support package imports before the module port list, as shown in the preceding code snippets. This compiler did support using explicit package references in the port list, and package import statements after the port list.)



SystemVerilog Advantage 7 — Packages provide a clean and simple way to use definitions of tasks, functions, and user-defined types throughout a design and verification project. Packages can eliminate duplicate code, the risk of mismatches in different blocks of a design, and the difficulties of maintaining duplicate code.

10. Decision operators and statements

Question 8 — Can my synthesis compiler help me make better decisions?

With SystemVerilog it can!

Two primary constructs used for RTL modeling are the **`if`**...**`else`** and **`case`** (with its wildcard variations) decision statements. 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 decision statements are coded to generate the intended hardware. Failing to follow proper coding guidelines can cause simulation versus synthesis mismatches (see Mills and Cummings [9]). SystemVerilog adds important enhancements to traditional Verilog to help reduce or eliminate these mismatches.

10.1 Wildcard decisions (wildcard equality)

An **if**...**else** decision often needs to be made based on portions of a vector. To do this in traditional Verilog requires using bit-selects or part-selects of the vector. For example:

```
module hi_lo_check
(input  wire [15:0] address,
 output reg         hi_lo_set
);
  always @(*)
    hi_lo_set = ((address[15:12]==4'hF) &&
                 (address[ 3: 0]==4'hF));
endmodule //hi_lo_check
```

Example 10.a: Verilog operation on portions of a vector

SystemVerilog has a better way. The *wildcard equality operators* compare the bits of two values, but with ability to mask out specific bits from the comparison. 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 ?.

```
module hi_lo_check
(input  logic [15:0] address,
 output logic        hi_lo_set
);
  always_comb
    hi_lo_set = (address ==? 16'hF??F);
endmodule: hi_lo_check
```

Example 10.b: SystemVerilog operation on portions of a vector using case equality operator

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

SystemVerilog Advantage 8 — Wildcard equality operators simplify decisions that are based on specific bits within a vector, and can reduce the risk of coding errors that could occur when using traditional Verilog bit-selects and part-selects.

10.2 Set membership operator (inside)

Sometimes a decision needs to be made based on a signal matching one or more of several possible values. This can be difficult in traditional Verilog if there are a large number of possible values.

```
if (data==256 || data==512 || data==1024
   || data==2048}) ...
// if data is one of the four values listed
```

SystemVerilog provides a much simpler way to do this. The **inside** set membership operator compares a value to a list of other values enclosed in **{ }**. The values can be constants or other signals. The list of values can include a range of values between **[ ]**, and can include values that are stored in an array. The inside set membership operator also allows bits in the value list to be masked out of a comparison in the same way as the case equality operators.

```
module range_comparator
(input  logic [15:0] data, a, b, c, d,
 output logic pass1, pass2, pass3, pass4, pass5
);
  always_comb begin
    pass1 = (data inside {256, 512, 1024, 2048});
    pass2 = (data inside {[0:255]});
    pass3 = (data inside {16'b0000????????1010});
    pass4 = (data inside {a, b, c, d});
    pass5 = (data inside {[a:b]});
  end
endmodule: range_comparator
```

Example 10.c: SystemVerilog inside operator

(Note: At the time this paper was written, some synthesis compilers required that the values enclosed in the { } set must be constant expressions.)

SystemVerilog Advantage 9 — The **inside** operator can significantly reduce the complexity of traditional Verilog code when decisions are based on any of several conditions being true, or a range of values being true.

10.3 Case...inside decisions

One of the ugliest gotchas of traditional Verilog is the subtle and undesired behavior of **casex** and **casez** statements. Many conference papers have focused on the problems caused by these constructs, and have recommended limited usage. The SystemVerilog **case**...**inside** statement replaces **casex** and **casez**, and eliminates these problems.

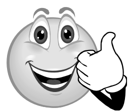
The SystemVerilog **case**...**inside** decision statement allows mask bits to be used in the *case items*. These "don't care" bits are specified using X, Z or ?, the same as with **casex**. The important difference is that **case**...**inside** uses one-way, asymmetric masking for the comparison, whereas **casex** uses two-way masking, 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. This 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 and will not cause any masking to occur.

In the following example, any X or Z bits in `opcode` will not be masked, and an invalid instruction will be trapped by the default condition:

```
module decoder
(output logic [2:0] control,
 input  logic [3:0] opcode
);
  always_comb begin
    case (opcode) inside
      4'b0???: control = 3'b000; // only test msb
      4'b10??: control = 3'b001; // test hi bits
      4'b1111: control = 3'b110; // test all bits
      default: control = 3'bxxx;
    endcase
  end
endmodule: decoder
```

Example 10.d: case...inside decision statement with wild cards

If **casex** had been used in the preceding example, any bits in opcode that were X or Z might have resulted in a branch other than the default being executed. This double-sided masking is different behavior than the synthesized gate-level implementation, which means the gate-level implementation is not what was verified at the RTL level.



SystemVerilog Advantage 10 — Using **case...inside** can prevent subtle design problems that can occur with **casex** or **casez**, and ensure that simulation closely matches how the synthesized gate-level implementation will behave.

10.4 Unique, unique0 and priority decision checks

From the earliest days of RTL synthesis, synthesis compilers have used 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 was simulated and verified.

These directives cause synthesis to perform specific optimizations to the gate-level implementation of the **case** decisions. A major flaw 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 simulation cannot verify that the optimization effects of these directives are appropriate in the design. That is to say, the *full_case*/*parallel_case* optimizations are not verified at the RTL level. There have been many conference papers documenting the hazards of the *full_case* and *parallel_case* synthesis directives (see Mills and Cummings [9], Cummings [10], Mills [12], and others).

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** as decision modifiers. In simulation, these decision modifiers add built-in verification checking to help detect when the *full_case* or *parallel_case* optimizations would not be desirable. This checking can help detect and prevent two common coding errors that can occur with decision statements:

- A case expression value occurs that is not specified in the case items (indicating that *full_case* synthesis optimization will not work correctly).
- A case expression value occurs that matches multiple case item branches. (indicating that *parallel_case* synthesis optimization will not work correctly).

For synthesis, these modifiers enable the appropriate *full_case* and/or *parallel_case* synthesis optimizations. Table 1 shows the mapping between the SystemVerilog decision modifiers and the synthesis directives:

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

Table 1: Decision modifiers vs. synthesis directives

(Note: At the time this paper was written, the **unique0** case was not supported by some synthesis compilers).

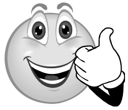
Example 10.e uses the **unique** decision modifier to illustrate the importance of these decision modifiers, and the advantage of using SystemVerilog over traditional Verilog for RTL design and synthesis. With a **unique case** decision, simulation and synthesis tools will print a violation report if overlapping *case items* exist, indicating that the *case items* should not 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 [2:0] {READY = 3'b001, // one-hot
                  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
```

Example 10.e: SystemVerilog unique case (partial code)

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 be decoded if the `case` statement is synthesized with *full_case*, *parallel_case* optimization.



SystemVerilog Advantage 11 — The **`priority`**, **`unique0`** and **`unique`** decision modifiers have built-in checking that will catch many of the potential hazards when using synthesis *full_case* and/or *parallel_case* optimizations. This checking can warn engineers when *full_case* or *parallel_case* synthesis optimizations would not be appropriate for the design.

SystemVerilog Advantage 12 — The **`priority`**, **`unique0`** and **`unique`** decision modifiers can also be used with **`if...else`** decisions. This provides the same synthesis optimizations that could only be done with **`case`** statements with traditional Verilog. The decision modifiers also provide simulation checking, to help ensure the optimized **`if...else`** decisions will work as intended.

Note: It is not always desirable to have the synthesis gate minimizations that can result from either the synthesis directives or the SystemVerilog decision modifiers. These optimizations should be used with caution. Also note that the **`priority`**, **`unique0`** and **`unique`** decision modifiers do not prevent latches, and do not flag all conditions that might infer latches. More details on the **`priority`**, **`unique0`** and **`unique`** decision modifiers can be found in conference papers by Cummings [10], Sutherland [11], Sutherland and Mills [14], and in Sutherland HDL training workshops.

11. Streaming operators

Question 9 — Can my synthesis compiler reverse all the bits or bytes of my parameterized-width data word in one operation?

With SystemVerilog it can!

Traditional Verilog does not have an operator that will reverse the bits or bytes of a vector. For a bit reversal, engineers might be tempted to try the following assignment statement, but quickly find out that this is an illegal assignment, because Verilog (and SystemVerilog) requires that part selects of vector use the same endian convention as the declaration of the vector. Example 11.a is illegal because the part-select of `data` on the right-hand side of the assignment has bit 0 as the most-significant bit, but the declaration of `data` has bit 0 as the least-significant bit.

```
parameter WIDTH=64;
reg [WIDTH-1:0] data;

always @(posedge clock)
  if (swap_bits)
    data[WIDTH-1:0] <= data[0:WIDTH-1]; //illegal
```

Example 11.a: Illegal attempt for a bit reversal

With traditional Verilog, a bit or byte reversal on a vector with a fixed width can be done using a concatenate operator, as shown on the following 8-bit vector:

```
module swap_bits
(input  wire [7:0] data_in,
 input  wire       swap_bits,
 input  wire       clock,
 output wire [7:0] data_out
);
  reg  [7:0] d;
  assign data_out = d;
  always @(posedge clock) begin
    d <= data_in;
    if (swap_bits)
      d[7:0] <= { d[0],d[1],d[2],d[3],
                  d[4],d[5],d[6],d[7] };
  end
endmodule
```

Example 11.b: Verilog style bit reversal, fixed vector size

Using a concatenate operator is cumbersome for larger size vectors, and impossible for vectors with a parameterized width. Another solution that will work with larger vector sizes and parameterized widths is to use a **`for`**-loop that iterates through each bit of a vector, as in Example 11.c. A **`for`**-loop can also be used to do a byte reversal, although the code becomes more complex (obfuscated might be a better description), as shown in the Example 11.c.

```
module swap_bits_or_bytes
#(parameter WIDTH=64)
(input  wire [WIDTH-1:0] d_in,
 input  wire             swap_bits,
 input  wire             swap_bytes,
 input  wire             clock,
 output reg  [WIDTH-1:0] d_out
);
  integer            i;

  always @(posedge clock) begin
    d_out <= d_in;
    if (swap_bits)
      for (i=0; i<WIDTH; i=i+1)
        d_out[(WIDTH-1)-i] <= d_out[i];
    else if (swap_bytes)
      for (i=0; i<WIDTH; i=i+8)
        d_out[((WIDTH-1)-i)-:8] <= d_out[i+:8];
```


©2014, Sutherland HDL, Inc.

```
  end
endmodule //swap_bits
```

Example 11.c: Verilog-style bit or byte reversal using for-loop

SystemVerilog has an easier way to swap vector bits or bytes! SystemVerilog adds magical *streaming operators*, also called *pack and unpack operators*. The streaming operators pull-out groups of bits from a vector (unpack), or push-in groups of bits into a vector (pack).

- **`{<<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 of the assignment. The bits pulled out can be in groups of any number of bits. The default is 1 bit at a time if a block size is not specified.

Example 11.d illustrates using the streaming operator for either a bit reversal or a byte reversal.

```
module swap_bits_or_bytes
#(parameter WIDTH=64)
(input  logic [WIDTH-1:0] d_in,
 input  logic             swap_bits,
 input  logic             swap_bytes,
 input  logic             clock,
 output logic [WIDTH-1:0] d_out
);
  always @(posedge clock) begin
    d_out <= d_in;
    if (swap_bits)
      d_out <= { <<  { d_out }}; // bit reverse
    else if (swap_bytes)
      d_out <= { << 8{ d_out }}; // byte reverse
    end
endmodule: swap_bits_or_bytes
```

Example 11.d: Bit or byte reversal using streaming operator

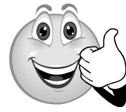
Unpacking occurs when a streaming operator is used on the left-hand side of an assignment. Blocks of bits are pulled out of the right-hand expression, and assigned to the expression(s) within the streaming operator. One synthesizable application of unpacking is to transfer the bits of a vector into elements of an array.

```
module array_loader
#(parameter WIDTH=32)
 (input  logic [WIDTH-1:0] in,
  input  logic             clock,
  output logic [7:0]       out [0:WIDTH/8-1]
 );
  always @(posedge clock)
    {>>{out}} <= in; // unpack input vector and
                     // assign to output array
endmodule: load_array
```

Example 11.e: Unpack a vector using streaming operator

In this example, if vector `in` has the value `32'hAABBCCDD`, the result in array `a` will be:

```
out[0]=AA, out[1]=BB, out[2]=CC, out[3]=DD
```



SystemVerilog Advantage 13 — 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 streaming operators provide a concise and accurate way to model functionality that would require complex, error-prone code in traditional Verilog.

Note: At the time this paper was written, some synthesis compilers did not support block sizes greater than one bit, such as were used in the byte reordering and vector unpacking examples.

12. Casting

Question 10 — Can my synthesis compiler eliminate false warning messages about intentional size and type conversions, and only warn about unintentional conversions?

With SystemVerilog it can!

Verilog and SystemVerilog are "loosely typed" languages that allow an expression of one type and vector size to be assigned to a net or variable of a different type or size. Loosely typed assignments are allowed because the language has built-in rules on how to convert, or implicitly "cast", an expression to match the type and size of the net or variable to which it is assigned.

Synthesis compilers follow the Verilog/SystemVerilog loosely-typed rules, and implement the implicit cast conversions in the gate-level design that is generated.

Many company's also use coding style checker software, commonly referred to a "lint checkers", to verify that RTL code adheres to synthesis and corporate coding styles. These lint programs are notorious for issuing a warning messages whenever a type or size implicit cast occurs. These warnings can be important, and should always be examined to ensure that the synthesized implementation matches the intended design functionality.

The problem with implicit cast warning messages is that RTL modeling uses the conversion rules of the language, probably much more than you realize. This places a burden on engineers to decide which size or type conversion

warnings are false (the conversions are correctly implementing design intent), and which warnings are real and indicate a design problem. Example 12.a illustrates this problem:

```
module set_reset_counter
 (input  wire       clk, rstN, setN,
  output reg [7:0] count
 );
  always @(posedge clk)
    if (!rstN)       count <= 1'b0;
    else if (!setN) count <= 4'hF;
    else             count <= count + 1;
endmodule //set_reset_counter
```

Example 12.a: Counter with several implicit size casts

Several implicit casts occur in Example 12.a. Most of these conversions correctly implement the design intent, but one conversion is due to a mistake in the code that would result in a synthesized design that does not work as intended.

1. The reset assignment to `count` is a 4-bit constant zero, but `count` is an 8-bit variable. Verilog/SystemVerilog will implicitly extend the 4-bit value with zeros, which will work as intended. (A more common implicit cast is the reset assignment `count <= 0;`. In this case, the unsized 0 is 32 bits wide, and the upper 24 bits are truncated, which also works as intended.)
2. The set assignment to `count` is a 4-bit constant hex F, but `count` is an 8-bit variable. Verilog will implicitly extend the 4-bit value with zeros, resulting in an 8-bit binary value of 00001111. *This will not set the counter to all ones, as intended!*
3. The `count + 1` operation in final line in Example 12.a will first implicitly convert `count` to a 32-bit value in order to add it to the literal 1, which is a 32-bit expression.
4. The literal 1 in `count + 1` is a signed expression, and will be implicitly converted to an unsigned value in order to add it to `count`, which is unsigned.
5. Finally, the upper 24 bits of the 32-bit result of `count + 1` are truncated in order to assign the result back to the 8-bit `count` variable.

Of these five implicit cast conversions, 4 of them work as intended. Any lint checker warnings for these implicit conversions can be ignored. Conversion number 2 in this list is a serious design flaw. Simulation will compile this flaw without any warning for the size mismatch, leaving it to a competent verification engineer to detect the functional bug. Synthesis will also compile without a warning for the size mismatch, and implement the functional bug in the resulting logic gates. In a larger design, there can be hundreds of false warnings regarding size and type conversions. These false warnings can hide a warning message for an incorrect or undesired size or type conversion. ***This is a serious problem!***

Example 12.b illustrates another situation where an implicit cast is desirable. This example shows a synthesizable coding style to rotate an expression a variable number of times.

```
module rotate
(input  wire [31:0] a,
 input  wire [ 5:0] b,
 output reg  [31:0] y
);
  always @*
    y = {a,a} >> b;  // rotated value is in
                     // lower 32 bits of {a,a}
endmodule //rotate
```

Example 12.b: Variable rotate operation that relies on implicit truncation

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 of the right-hand expression are automatically truncated off. This is the desired functionality, and the code simulates — and will synthesize — correctly.

Simulators and most Synthesis compilers will not generate a warning message for the implicit truncation in Example 12.b. Lint checkers, however, will issue a warning message about the implicit size truncation, even though the truncations are correct and desired. In a large design, there can be hundreds of these pesky “size mismatch” warning messages, most of which are false warnings.

SystemVerilog adds an explicit cast operator to traditional 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);`**

These cast operators follow the same rules as the implicit cast conversion rules that are built into Verilog and SystemVerilog, but an explicit cast documents that the design engineer is aware of, and wants, the conversion.

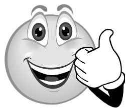
One usage of explicit casting is to eliminate those annoying “size mismatch” synthesis warning messages that can hide other important warnings. When explicit casting is used, lint checkers will not generate a conversion warning.

Example 12.c shows how size casting can be used to prevent the size mismatch warnings with a variable rotate operation, where the upper 32-bits of the intermediate result are intentionally truncated.

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

always_comb
    y = 32'({a,a} >> b);// rotated value is in
                        // lower 32 bits of {a,a}
```

Example 12.c: Variable rotate operation with casting for explicit truncation



SystemVerilog Advantage 14 — Explicit casting eliminates false type or size mismatch warning messages that can hide other important warnings. Explicit casting also serves to document that a size, type, or signing conversion is intended.

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

13. Tasks and functions

Question 11 — Can my synthesis compiler prevent me from writing tasks (subroutines) that simulate, but won't synthesize?

With SystemVerilog it can!

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. Functions have very few restrictions because they must execute in zero time. Tasks, on the other hand, have many restrictions in order to meet synthesis requirements.

These restrictions are outside the scope of this paper, but what is important to note is that it is a common problem to write a task that is legal for simulation and functionally correct, but is illegal for synthesis. Coding errors of this nature might not be found until after days or weeks of RTL design, simulation and functional verification have been invested into a project. Restructuring the task to make it legal for synthesis, and then re-verifying the RTL functionality, can add delays to a project.

SystemVerilog enhances Verilog tasks and functions in several ways that are synthesizable. Only two of those enhancements are examined in this paper: function output arguments, and void functions.

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 important when coupled with a second enhancement, void functions.

SystemVerilog adds the ability to declare a function as **`void`**, indicating the function does not return a value (but can assign to output or inout 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 stall execution until simulation time advances.)

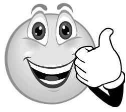
Tasks have many restrictions, which can make them difficult to synthesize. On the other hand, functions, including void functions, have very few synthesis restrictions, and, therefore, will almost always synthesize. Void functions make it possible to write task-like subroutines that will synthesize without the potential problems of tasks. Example 13.a illustrates using a void function.

```
module ripple_adder
  (input  logic [31:0] in1, in2,
   output logic [31:0] result,
   output logic        carry);

  function void ripple_add (input  [31:0] a, b,
                            output [31:0] sum,
                            output        co);
    logic [31:0] c;
    //half adder for bit zero
    sum[0] = a[0] ^ b[0];
    c[0]   = a[0] & b[0];
    //full adders for remaining bits
      for (int i = 1; i <= 31; i++) begin
        sum[i] =  a[i] ^ b[i] ^ c[i-1];
        c[i]   = (a[i] & b[i])   |
                 (a[i] & c[i-1]) |
                 (b[i] & c[i-1]);
      end
    co = c[31];
  endfunction

  always_comb
    ripple_add(in1, in2, result, carry);
      // the void function is called like a task
endmodule: ripple_adder
```

Example 13.a: Void function as a task-like subroutine



SystemVerilog Advantage 15 — Using void functions instead of tasks in RTL code can help ensure that subroutines will be synthesizable. Void functions help prevent a common problem when using tasks, where the model works correctly in simulation, but won't synthesize.

14. Interfaces

Question 12 — Can my synthesis compiler support modeling standard bus protocol signals at a higher level of abstraction that avoids code duplication?

With SystemVerilog it can!

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. Unfortunately, traditional Verilog does not support this good coding practice when it comes to modeling the communication between modules. Example 14.a illustrates connecting a master device to a slave device, and shows how traditional Verilog requires declarations to be duplicated several times.

```
module master
(inout  wire [31:0] data,
 output reg  [ 4:0] address,
 output reg         read, write,
 input  wire        clk, rstN
);
  ... // master's functionality not shown
endmodule //master

module slave
(inout wire [31:0] data,
 input wire [ 4:0] address,
 input wire        read, write,
 input wire        clk, rstN
);

  ... // slave's functionality not shown
endmodule //slave

module top
(inout  wire [31:0] data,
 output wire [ 4:0] address,
 output wire        read, write,
 input  wire        clock, reset
);
  master u1 (.data    (data),
             .address (address),
             .read    (read),
             .write   (write),
             .clk     (clock),
             .rstN    (reset) );
  slave  u2 (.data    (data),
             .address (address),
             .read    (read),
             .write   (write),
             .clk     (clock),
             .rstN    (reset) );
endmodule //top
```

Example 14.a: Verilog style inter-module communication (lots of duplicated declarations)

In this simple example, the signals `data`, `address`, `read` and `write` that traverse between `mod_a` and `mod_b` are listed seven times. If a design change occurs, and an additional signal needs to be communicated between the two modules, or if the specification of a bus width changes, the Verilog source code will need to be modified in several different places. Typically, each module will be in a different file, making it difficult (and error prone) to make changes to the communication protocol signals.

SystemVerilog provides a more abstract way for modules to communicate, called an *interface*. An interface can be used to encapsulate signal declaration information in one place. Interfaces are an ideal RTL design construct for when design blocks communicate using a standard bus protocol such as AMBA AHB or USB 3.0 or an in-house proprietary protocol.

Example 14.b uses an interface to encapsulate the signals from Example 14.a 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, instead of having to modify the port list in every module that uses the communication bus.

```
interface comm_bus;
  wire  [31:0] data;
  logic [ 4:0] address;
  logic        read, write;

  modport m_ports (inout data,
                   output address, read, write);

  modport s_ports (inout data,
                   input address, read, write);
endinterface: comm_bus

module master
(interface a1,    // generic interface port
 input clk, rstN
);
  ... // master's functionality not shown
endmodule: master

module slave
(comm_bus b1,     // type-specific interface port
 input clk, rstN
);
  ... // slaves's functionality not shown
endmodule: slave
```


©2014, Sutherland HDL, Inc.

```
module top;
  logic clock, reset;

  comm_bus i1(); // instance of the interface

  master u1 (.a1   (i1.m_ports),   // connect i/f
             .clk  (clock),
             .rstN (reset));

  slave  u2 (.b1   (i1.s_ports),   // connect i/f
             .clk  (clock),
             .rstN (reset));
endmodule: top
```

Example 14.b: SystemVerilog style inter-module communication using interface ports

In the `master` 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 as a "*generic interface port*". An instance of any interface definition can be connected to a generic interface port. In the `slave` example, interface port `b1` is declared as a **`comm_bus`** port type. This is referred to as a "*type-specific interface port*". It is only legal to connect an instance of a `comm_bus` interface to this type-specific interface port. The authors recommend only using 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: At the time this paper was written, some synthesis compilers did not support, or imposed restrictions, on generic interface ports.)

Interfaces can do more than just encapsulate the signals of a bus protocol. Interfaces can encapsulate functionality associated with the signals that make up the bus. For example, an interface might contain a function that returns the parity of its data bus. Interfaces can also encapsulate built-in verification code, such as assertions, so that all communications over the bus are automatically verified. To be synthesizable, any functional code within an interface must adhere to synthesizable RTL coding rules.

SystemVerilog Advantage 16 — Interfaces simplify complex bus definitions and interconnections, and ensure consistency throughout the design. Interfaces can substantially reduce redundant declarations within a design, which leads to code that is easier to maintain and easier to reuse.

15. Vector fill tokens

Question 13 — Can my synthesis compiler fill vectors of any size and with parameterized widths with all 1s?

With SystemVerilog it can!

With traditional Verilog, there is no simple way to fill a vector with all ones. The only options are to define a literal value of all ones, or to use operations such as replicate or invert. Any of these approaches can lead to design errors when bus widths are parameterized to allow the width to be redefined for each usage of a model.

In Example 15.a shows a set/reset flip flop where the flip flop is set by assigning a literal value of all ones.

```
module set_reset_dffN
#(parameter N = 64)
(input  wire         clk, rstN, setN,
 input  wire [N-1:0] d,
 output reg  [N-1:0] q
);
  always @(posedge clk) // synchronous set/reset
    if (!rstN)
      q <= 64'h0;       // resets all bits to 0
    else if (!setN)
      q <= 64'hFFFFFFFFFFFFFFFF; // sets at most
    else                        // 64 bits to 1
      q <= d;
endmodule //set_reset_dffN
```

Example 15.a: Verilog style of setting a vector to all ones

Example 15.a has a serious flaw! If parameter `N` is redefined to something larger than 64 bits, the reset functionality will still be correct because of Verilog implicit cast rules that will extend the 64-bit 0 with additional zeros (see Section 12). The flaw is that the set functionality will no longer function as intended. The 64-bits of 1 will be extended with zeros, resulting in a faulty design. Synthesis compilers will implement this faulty functionality in the gate-level design since that is what was modeled in the RTL code.

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.

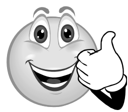
- **`'0`** fills all bits on the left-hand side with 0
- **`'1`** fills all bits on the left-hand side with 1
- **`'z`** fills all bits on the left-hand side with z
- **`'x`** fills all bits on the left-hand side with x

These context-dependent fill tokens will always function as intended, without the flaw that was in the previous Verilog example. The following example illustrates using the SystemVerilog fill tokens.

©2014, Sutherland HDL, Inc.

```
module set_reset_dffN
#(parameter N = 64)
(input  logic         clk, rstN, setN,
 input  logic [N-1:0] d,
 output logic [N-1:0] q
);
  always_ff @(posedge clk) // synch. set/reset
    if (!rstN)
      q <= '0;       // reset all bits to 0
    else if (!setN)
      q <= '1;       // set all bits to 1
    else
      q <= d;
endmodule: set_reset_dffN
```

Example 15.b: SystemVerilog style of setting a vector to all 1



SystemVerilog Advantage 17 — Vector fill values simplify code, and help ensure the code will function correctly These fill tokens can also prevent size mismatches between the fill value and assignment or expression in which the value is used.

16. Expression size functions

Question 14 — Can my synthesis compiler automatically calculate the size of my address bus, based on the depth of my memory storage?

With SystemVerilog it can!

A common situation is for the vector width of a bus to be dependent on some other declaration. For example, the width of FIFO read and write pointers are dependent on the depth (number of storage locations) in the FIFO. This is illustrated in Example 16.a, which uses traditional Verilog.

```
module fifo
#(parameter DEPTH   = 32, // depth of the FIFO
            D_WIDTH = 8,  // data bus width
            P_WIDTH = 5   // pointer widths
)                         // (depend on DEPTH)
(input  wire              i_clk, o_clk,
 input  wire [P_WIDTH-1:0] rd_ptr, wr_ptr,
 input  wire [D_WIDTH-1:0] data_in,
 output reg  [D_WIDTH-1:0] data_out
);
  reg [D_WIDTH-1:0] storage [0:DEPTH-1];
  always @(posedge i_clk)
    storage[wr_ptr] <= data_in;

  always @(posedge o_clk)
    data_out <= storage[rd_ptr];
endmodule // fifo
```

Example 16.a: Verilog style port sizes with dependencies, using hard coded calculations

With traditional Verilog, the width of the read and write pointers (the `P_WIDTH` parameter in the preceding example) must be calculated by the design engineer. If a different value for the FIFO `DEPTH` parameter is specified when the FIFO is synthesized, the designer must also calculate and specify a new value for `P_WIDTH` parameter. Failure to change `P_WIDTH`, or calculating an incorrect value, can result in a gate-level FIFO implementation that does not work as intended.

With traditional Verilog, a constant function can be used to calculate parameter values and vector widths, instead of hard coding in the value or width, as was done in Example 16.a. Note, though, that each module that needs this function must contain a local copy of the function, since traditional Verilog does not have packages for shared definitions, as discussed in Section 9. Example 16.b illustrates using a constant function to calculate the size of the read and write pointers, based on the depth of the FIFO.

```
module fifo
#(parameter DEPTH   = 32, // depth of the FIFO
            D_WIDTH = 8,  // data bus width
            P_WIDTH = log2(DEPTH) // ptr widths
)
(input  wire               i_clk, o_clk,
 input  wire [P_WIDTH-1:0] rd_ptr, wr_ptr,
 input  wire [D_WIDTH-1:0] data_in,
 output reg  [D_WIDTH-1:0] data_out
);
  reg [D_WIDTH-1:0] storage [0:DEPTH-1];

  always @(posedge i_clk)
    storage[wr_ptr] <= data_in;

  always @(posedge o_clk)
    data_out <= storage[rd_ptr];

  function integer log2 (input DEPTH);
    case (DEPTH)
        4: log2 = 2;
        8: log2 = 3;
       16: log2 = 4;
       32: log2 = 5;
       64: log2 = 6;
      128: log2 = 7;
      256: log2 = 8;
      default: begin
        $display("ERROR: Unsupported DEPTH");
        $finish;    // abort simulation
        log2 = 5;  // default for synthesis
      end
    endcase
  endfunction
endmodule // fifo
```

Example 16.b: Verilog style port sizes with dependencies, using a constant function

SystemVerilog provides two special system functions that can be useful in synthesizable RTL code, **`$bits`** and

$clog2. These functions can help prevent errors in declaration sizes, and help with writing models that are scalable and reusable as specifications change in current or future projects.

The **$bits** system function 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 built-in data type or user-defined type; *expression* can be any value, including or unpacked arrays and operation results.

For synthesis, **$bits** can be used in port declarations, net and variable declarations, and constant definitions.

The **$clog2** function returns the ceiling of the log base 2 (the log2 rounded up to an integer value) of a vector. The function can be used in RTL models to declare vector sizes based on the value of a **parameter** or **localparam** constant, or the return of **$bits**. Example 16.c shows how the traditional Verilog FIFO in Example 16.a can be written in synthesizable SystemVerilog so that P_WIDTH is automatically calculated from the value of DEPTH. If DEPTH is redefined, P_WIDTH will automatically scale to the correct width. The risk of inadvertently specifying an incorrect value for P_WIDTH has been eliminated.

```
module fifo
#(DEPTH   = 32,            // depth of the FIFO
  D_WIDTH = 8,             // data bus width
  P_WIDTH = $clog2(DEPTH)  // pointer widths
)
(input  logic               i_clk, o_clk,
 input  logic [P_WIDTH-1:0] rd_ptr, wr_ptr,
 input  logic [D_WIDTH-1:0] data_in,
 output logic [D_WIDTH-1:0] data_out
);
  logic [D_WIDTH-1:0] storage [0:DEPTH-1];

  always_ff @(posedge i_clk)
    storage[wr_ptr] <= data_in;

  always_ff @(posedge o_clk)
    data_out <= storage[rd_ptr];
endmodule: fifo
```

Example 16.c: SystemVerilog style port sizes with dependencies

The following Example 16.d shows a more elaborate usage of **$bits** and **$clog2**. This example concisely models a configurable packet register, where each packet contains a payload that is an array of data words. The payload array size is parameterized, and can be specified as a different size for each usage of the register. The register storage is a vector, where the size of the vector is based on the size of the array. The model uses the SystemVerilog streaming operator to transfer the array contents into the register (see Section 11). All declarations in this model will automatically scale to the correct widths to match the payload size. This concise model represents a significant number of logic gates, once synthesized.

```
package payload_64;
  localparam MAX_PAYLOAD = 64;

  typedef struct {
    logic [63:0] start_address;
    logic [$clog2(MAX_PAYLOAD)-1:0] xfer_size;
    logic [ 7:0] payload [0:MAX_PAYLOAD-1];
  } packet_t;
endpackage: payload_64

module payload_register
(input logic clk,
 input payload_64::packet_t           in,
 output logic [$bits(in.payload)-1:0] out
);
  always_ff @(posedge clk)
    out <= { << { in.payload }}; // pack entire
                                 // payload array
                                 // into register
endmodule: payload_register
```

Example 16.d: Using SystemVerilog $bits and $clog2

SystemVerilog Advantage 18 — The **$bits** and **$clog2** expression size functions enable writing RTL models that are scalable and "correct by construction". They eliminate the risk of an engineer inadvertently specifying incongruent vector widths.

17. Synthesis compiler support for SystemVerilog

The SystemVerilog constructs discussed in this paper are supported by most major ASIC and FPGA synthesis compilers. There are exceptions, however, where specific constructs are not supported, or are only partially supported, by specific synthesis compilers. Appendix A contains a table showing the synthesis compiler support for the constructs shown in this paper.

18. Additional synthesizable SystemVerilog constructs

There are many more RTL modeling constructs that SystemVerilog adds to traditional Verilog, and which are synthesizable by most major RTL synthesis compilers. These constructs include:

- 2-state types
- Parameterized types
- **$unit** declaration space

©2014, Sutherland HDL, Inc.

- Unions
- ++ and -- increment/decrement operators
- Multiple for-loop iterator variables
- **do**...**while** loops
- **foreach** loops
- **break** and **continue** loop controls
- Continuous assignments to variables
- Task/function formal arguments with default values
- Task/function calls with pass by name
- Function return statements
- Parameterized tasks and functions
- Dot-name and dot-star netlist connections
- Named statement group ends
- Version keyword compatibility directives
- **const** variables
- Assertions
- Local time unit and precision definitions

These additional constructs are not covered in this paper, but are discussed in another paper by the authors (see Sutherland and Mills [14]). Note that some of these constructs might not be supported in some synthesis compilers.

19. 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 with fewer lines of code. Many of the SystemVerilog constructs that have been discussed in this paper achieve that goal. Structures, user-defined types, array assignments and interfaces are examples of how SystemVerilog can significantly reduce the amount of code needed to represent complex functionality.
- Clearly indicate design intent. Specialized procedural blocks and decision modifiers allow engineers to state intention. Software tools can then verify whether 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 other SystemVerilog features 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 has shown a number of advantages for designing with, and synthesizing, SystemVerilog. These advantages show that there are substantial benefits to be gained from using SystemVerilog instead of traditional Verilog for RTL design and synthesis. The advantages mentioned in the paper are:

2. You no longer need to worry about when to declare module ports or local signals as `wire` or `reg`. With SystemVerilog, you can declare all module ports and local signals as `logic`, and software tools will correctly infer nets or variables for you.
3. Designs that frequently manipulate parts of a vector are simpler to model, more self-documenting, and correct by construction.
5. Array copy assignments can significantly reduce the complexity of design code for moving blocks of data from one array to another. Array assignments also do not require changing any code if the sizes of the arrays change.
4. Enumerated types can prevent coding errors that could be difficult to detect and debug!
6. By using structures to bundle related signals together, the signals can be assigned and transferred to other modules as a bundle, reducing the lines of code and ensuring consistency.
7. Packages provide a clean and simple way to use definitions of tasks, functions, and user-defined types throughout a design and verification project. Packages can eliminate duplicate code, the risk of mismatches in different blocks of a design, and the difficulties of maintaining duplicate code.
1. The benefits of using 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.
8. Wildcard equality operators simplify decisions that are based on specific bits within a vector, and can reduce the risk of coding errors that could occur when using traditional Verilog bit-selects and part-selects.
9. The `inside` operator can significantly reduce the complexity of traditional Verilog code when decisions are based on any of several conditions being true, or a range of values being true.
10. Using `case`...`inside` can prevent subtle design problems that can occur with `casex` or `casez`, and ensure that simulation closely matches how the synthesized gate-level implementation will behave.

©2014, Sutherland HDL, Inc.

11. The `priority`, `unique0` and `unique` decision modifiers have built-in checking that will catch many of the potential hazards when using synthesis full_case and/or parallel_case optimizations. This checking can warn engineers when full_case or parallel_case synthesis optimizations would not be appropriate for the design.
12. The `priority`, `unique0` and `unique` decision modifiers can also be used with `if...else` decisions. This provides the same synthesis optimizations that could only be done with `case` statements with traditional Verilog. The decision modifiers also provide simulation checking, to help ensure the optimized `if...else` decisions will work as intended.
13. 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 streaming operators provide a concise and accurate way to model functionality that would require complex, error-prone code in traditional Verilog.
14. Explicit casting eliminates false type or size mismatch warning messages that can hide other important warnings. Explicit casting also serves to document that a size, type, or signing conversion is intended.
15. Using void functions instead of tasks in RTL code can help ensure that subroutines will be synthesizable. Void functions help prevent a common problem when using tasks, where the model works correctly in simulation, but won't synthesize.
16. Interfaces simplify complex bus definitions and interconnections, and ensure consistency throughout the design. Interfaces can substantially reduce redundant declarations within a design, which leads to code that is easier to maintain and easier to reuse.
17. Vector fill values simplify code, and help ensure the code will function correctly These fill tokens can also prevent size mismatches between the fill value and assignment or expression in which the value is used.
18. The `$bits` and `$clog2` expression size functions enable writing RTL models that are scalable and "correct by construction". They eliminate the risk of an engineer inadvertently specifying incongruent vector widths.

20. Conclusion

Appendix A says it all. This appendix shows how well six different synthesis compilers support the SystemVerilog constructs illustrated in the examples listed in this paper. ***The nearly full support from six different synthesis compilers for these examples is proof that engineers can and should take advantage of the many benefits available from using SystemVerilog in ASIC and FPGA designs!***

21. Acknowledgements

The authors express their appreciation to Shalom Bresticker of the DVCon Technical Committee for his detailed review of the initial draft of this paper, and suggestions for improving the quality of the paper.

22. References

[1] *"1364-2001 IEEE Standard for Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2001. ISBN: 0-7381-2826-0.

[2] *"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).

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

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

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

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

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

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

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

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

[11] 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.

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

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

[14] Sutherland and Mills, *"Synthesizing SystemVerilog: Busting the Myth that SystemVerilog is only for Verification"*, Synopsys Users Group (SNUG) Silicon Valley conference, Santa Clara, California, March 2013.

©2014, Sutherland HDL, Inc.

Appendix A: Table of synthesis compiler support

The following table lists the SystemVerilog constructs discussed in this paper, and shows the support for these constructs by several major synthesis compilers. The information in this table is based the most current released versions available to the authors at the time this paper was written.

NOTE: The support for SystemVerilog constructs is based solely on the examples listed in this paper. These examples are not exhaustive and do not test every aspect of each SystemVerilog construct. The authors feel, however, that the examples show proper and common usage for these constructs. The nearly full support from six different synthesis compilers for the examples in this paper proves that engineers can and should take advantage of the many benefits available from using SystemVerilog in ASIC and FPGA designs.

SystemVerilog Construct	Synthesis Compiler A	Synthesis Compiler B	Synthesis Compiler C	Synthesis Compiler D	Synthesis Compiler E	Synthesis Compiler F
always_ff, always_comb, always_latch	**yes**	**yes**	**yes**	**yes**	partial*	partial*
logic type / inferred wire or reg	partial*	**yes**	**yes**	**yes**	**yes**	**yes**
Vectors with subfields	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
Enumerated types	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
Array assignments and copying, array ports	**yes**	partial*	**yes**	**yes**	**yes**	**yes**
Structures and structure assignments	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
Packages and package import	**yes**	**yes**	**yes**	partial*	**yes**	**yes**
==? wildcard equality operator	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
inside operator, constant expressions	**yes**	**yes**	**yes**	**yes**	**yes**	no
inside operator, variable expressions	no	no	**yes**	**yes**	**yes**	no
case…inside decisions	**yes**	**yes**	**yes**	**yes**	**yes**	no
unique and priority decisions	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
unique0 decisions	partial*	no	**yes**	no	no	no
Streaming operators (pack and unpack)	**yes**	**yes**	**yes**	**yes**	**yes**	no
Casting	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
Void functions	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
Interfaces	partial*	**yes**	**yes**	**yes**	**yes**	**yes**
Vector fill tokens	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
$bits, $clog2 expression size function	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**

*partial indicates that a compiler either: a) accepted some of the constructs listed in the table row, but not all of the constructs, or b) accepted a construct in the paper example but did not produce the correct results (e.g.: some compilers parsed always_comb and always_ff, but did not generate warnings for some of the incorrect logic within those blocks).

The synthesis compiler used to test the examples in this paper are (in a different order than the table): ***Synopsys Design Compiler*** (also called ***DC*** and ***HDL Compiler***) version 2013.03-SP4, ***Synopsys Synplify-Pro*** version 2013.03-SP1, ***Mentor Precision Synthesis RTL Plus*** version 2013a.9, ***Cadence Encounter RTL Compiler*** version 13.10, ***Xilinx Vivado*** free web version 13.4, and ***Altera Quartus II*** free web version 13.1.

©2014, Sutherland HDL, Inc.