DesignCon 2011

What, If Anything, In SystemVerilog Will Help Me With FPGA-based Designs?

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

Abstract

SystemVerilog has gained rapid acceptance as a powerful ASIC and custom IC design and verification language. But what about FPGA designers? Is there anything in SystemVerilog that will help make designing an FPGA easier and quicker, or is it best to stick with VHDL or classic Verilog for FPGA designs? There is no one right answer to this question — what is right for one company or project might not be best for another company or project. To answer the question for your projects, you need accurate information. This paper examines:

1) A few of the features in SystemVerilog that might be of benefit in FPGA design.

2) Which of these features are supported by major commercial FPGA design tools.

3) Which of these features are supported by major FPGA vendors.

4) When, if ever, is the time for FPGA designers to use these SystemVerilog features.

The primary objective is help you determine if SystemVerilog is the right design language for your projects today, or sometime in the future.

Author Biography

Stuart Sutherland is a well-known Verilog and SystemVerilog expert, with more than 23 years of experience using these languages for design and verification. His company, Sutherland HDL, specializes in training engineers to become true wizards using SystemVerilog. Stuart is an active member of the IEEE SystemVerilog standards committee, and has been a technical editor for every version of the IEEE Verilog and SystemVerilog Language Reference Manuals since the standards began in 1993. Prior to founding Sutherland HDL, Mr. Sutherland worked as an engineer on high-speed graphics systems used in military flight simulators. In 1988, he became a corporate applications engineer for Gateway Design Automation, the founding company of Verilog, and has been deeply involved in the use of Verilog and SystemVerilog ever since. Mr. Sutherland has authored several books and conference papers on Verilog and SystemVerilog. He holds a Bachelors Degree in Computer Science with an emphasis in Electronic Engineering Technology and a Masters Degree in Education with an emphasis on eLearning.

Verilog versus SystemVerilog

Are you using SystemVerilog today? The answer might surprise you.

Technically, there is no such thing as "*Verilog*" anymore. The IEEE dropped the name *Verilog* in 2009, and changed the name of the standard to "*SystemVerilog*".

A brief history lesson might help understand the reason for the a name change. Verilog began as a proprietary language in the early 1980s, before there were any industry standards for hardware design languages. Verilog quickly became popular with ASIC designers for several reasons, which won't be discussed here. In 1992, Verilog was made an open language, and shortly afterwards the IEEE began the process of making Verilog a true industry standard, which became the IEEE 1364-1995 Verilog standard. In 2001, the IEEE introduced a number of enhancements to Verilog to keep pace with the ever-increasing complexity of hardware designs.

In keeping with Moore's law, however, the size and complexity of hardware designs and verification testbenches quickly outgrew the capabilities of Verilog (and VHDL, another IEEE standard hardware design language). It wasn't that Verilog (and VHDL) couldn't model and verify these complex designs, it was that the amount of code that was required was becoming unmanageable and inefficient. The IEEE Verilog standards committee went to work defining a new generation of Verilog with literally hundreds of enhancements, some small and some huge, to enable modeling and verifying much larger and complex designs. The original intent was that these enhancements would all be part of a new Verilog-2005 IEEE standard. However, the magnitude of the new features was going to require an extensive overhaul of existing Verilog simulators, synthesis compilers, and other engineering tools based on Verilog-2001. The IEEE made the decision to split the new language features into a separate document, under a separate standards number, so that Electronic Design Automation (EDA) companies could continue to be Verilog compliant while implementing the huge number of new language features. Thus, the IEEE 1364-2005[3] defined the base Verilog language, and a separate document, "IEEE 1800-2005"[2] and dubbed "SystemVerilog", contained the definition of the many new features being added to Verilog.

The last piece in this brief history was the merging of the SystemVerilog extensions into the actual Verilog document. This merged document was ratified and released by the IEEE in 2009. For various reasons — some political and some due to IEEE red tape — the merged document kept the later standards number and name, IEEE 1800 SystemVerilog. In 2009, the IEEE officially ended support of the original IEEE 1364 Verilog standard and the IEEE 1800-2009 SystemVerilog standard[1] superceded and replaced Verilog.

In short, SystemVerilog is Verilog, but it is Verilog on steroids. If you are doing anything with what was once called the Verilog Hardware Description Language, you are actually using SystemVerilog, though perhaps not all of the features of the language. It is those features that is the focus of this paper.

The SystemVerilog-2005 standard added several dozen major constructs and hundreds of smaller, yet significant, extensions to the Verilog-2005 standard. These extensions fall into two categories: constructs for modeling hardware and constructs for verifying hardware. This paper examines the hardware modeling extensions to Verilog. An emphasis is placed on constructs that might benefit FPGA designers, along with an explanation as to what those benefits might be.

Variable data types

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

Bit-vector variables

Verilog has the **`reg`** variable type, which can be defined to represent any vector size. SystemVerilog extends the Verilog bit-vector type with two new keywords:

`logic` — a 4-state variable with a user-defined vector width. **`logic`** is a synonym for the Verilog **`reg`** type. The two keywords are interchangeable.

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

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

Traditional Verilog had the **`integer`** variable type, which represented 32-bit signed integer storage. SystemVerilog extends this with several new types:

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

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

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

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

These new variable types are synthesizable, and follow the same rules defined in the Verilog 1364.1-2002 standard[4] for synthesizing **`integer`** variables.

What is the benefit in FPGA design projects? The **`logic`** keyword does not add any new functionality over traditional Verilog, but it can help make code more self-documenting. The old **`reg`** keyword was easy to confuse with the word "*register*", which has special meaning in hardware. The **`reg`** keyword does not, in any way, infer a hardware register, but engineers not familiar with Verilog synthesis have been know to see the **`reg`** keyword and mistakenly assume it represented a hardware register. Using the **`logic`** keyword eliminates the risk of this false assumption.

The **`int`** variable type does not offer any real benefit over the traditional Verilog **`integer`** type, but it is a shorter keyword and more like the C language. Many engineers prefer to use **`int`** instead of **`integer`** as a **`for`**-loop iterator type.

The **`bit`**, **`byte`**, and other 2-state data types can be useful in Object Oriented testbenches that use constrained random value generation. In hardware models, however, these 2-state data types should be avoided. 2-state variables can mask design problems that would otherwise have shown up as a logic X. Furthermore, 2-state types becomes a simple interconnecting net after synthesis, which can transfer 4-state values. This can lead to differences in RTL versus post-synthesis gate-level simulation results. This difference can also impact formal verification of pre-synthesis versus post-synthesis models.

Net Data Types

SystemVerilog extends the traditional Verilog net data types with a new net type called **`uwire`**. The new **`uwire`** data type ("*uwire*" is short for "*unresolved wire*") only permits a single source to drive a net. (This new net type is actually defined in the IEEE 1364-2005 Verilog standard[3], but is often considered a SystemVerilog enhancement to traditional Verilog.)

What is the benefit in FPGA design projects? The **`uwire`** net type can prevent modeling errors where single-source functionality is intended, but the same net name was inadvertently used more than once.

Note: Many synthesis compilers translate uwire into wire in the synthesized model. Unfortunately, this means the post-synthesis model no longer enforces single-driver semantics. Modifications to the post-synthesis netlist could result in unintentional multi-driver logic. This paper recommends that synthesis compilers maintain the **`uwire`** declaration in the resultant synthesized netlist so that the single-source semantics are preserved in post-synthesis simulation.

User-defined types

The old Verilog language only had pre-defined built-in data types, such a **`reg`** and **`wire`**. It made the language simple to learn and use, but also required more coding to model large designs with complex data. SystemVerilog allows designers to create new, user-defined data types. Both variables and nets can be declared as user-defined types. If a net type keyword is not specified, then user-defined types are assumed to be variables. SystemVerilog has several user-defined type constructs for use in design and verification. Only type definitions (**`typedef`**), enumerated types, structures, and unions are synthesizable, and are discussed in this section of the paper. The non-synthesizable user-defined types are not discussed.

Type definitions (typedef)

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

```
typedef int unsigned uint_t;
typedef enum bit {FALSE, TRUE} bool_t;
```

User-defined types can be declared in a package and used in any number of modules. Module ports can also be declared as a user-defined type.

Enumerated types

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

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

Enumerated types have a base data type, which by default is **`int`** (a 2-state, 32-bit type). In the example above, `State` is an **`int`** type, and `WAITE`, `LOAD` and `READY` have 32-bit **`int`** values.

Designers can specify an explicit base type, allowing enumerated types to more specifically model hardware. An example is:

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

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

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

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

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

```
always @(posedge clock or negedge resetN)
 if (resetN == 0)State <= State.first; // reset to first enum value
 else State <= NextState;

always @(State or move_ahead)
 if (move_ahead)
  NextState <= State.next; // increment to next enum value
```

What is the benefit in FPGA design projects? Enumerated types can reduce or eliminate many common coding errors, including errors that can very difficult to detect and to debug. The following old-style Verilog example contains several simple errors that are all syntactically legal but have functional bugs. Hopefully no engineer would make all of these mistakes in one piece of code, but even one of these errors can be a tough bug to find in a large design.

```
localparam [2:0] WAIT=3'b001, LOAD=3'b010, DONE=3'b100;
localparam [2:0] RED=3'b001, GREEN=3'b010, BLUE=3'b100;

reg [2:0] State1, nState1;
reg [1:0] State2, nState2;          // BUG - variable is wrong size

always @(posedge clk or negedge rstN)
  if (!rstN) State1 <= 0;           // BUG - reset to illegal value
  else       State1 <= nState2;     // BUG - wrong nState variable
always @(State1)
  case (State1) // next state logic
    WAIT : nState1 = State1 + 1;
    LOAD : nState1 = State1 + 1;    // BUG - results in illegal value
    DONE : nState1 = State1 + 1;    // BUG - results in illegal value
```

The next example is almost identical to the one above, but modeled with SystemVerilog enumerated types. Every one of the syntactically legal bugs above become syntax errors, instead of functional bugs that must be detected and debugged during simulation.

```
typedef enum logic [2:0] {WAIT=3’b001, LOAD=3’b010, DONE=3’b100}
  fsm1_states;
typedef enum logic [1:0] {RED=3'b001, GREEN=3'b010, BLUE=3'b100}
  fsm2_states;               // ILLEGAL - size mismatch not allowed

fsm1_states State1, nState1;
fsm2_states State2, nState2;

always @(posedge clk or negedge rstN)
  if (!rstN) State1 <= 0;        // ILLEGAL - must use an enum label
  else       State1 <= nState2;  // ILLEGAL - must use same enum
always @(State1)
  case (State1) // next state logic
    WAIT : nState1 = State1 + 1;
    LOAD : nState1 = State1 + 1; // ILLEGAL - must use State1.next
    DONE : nState1 = State1 + 1; // ILLEGAL - must use State1.next
```

Structures

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

```
struct {  // anonymous structure variable
  logic [7:0] tag;
  int         a, b;
} packet_s;
typedef struct {           // typed structure
  logic [ 1:0] parity;
  logic [63:0] data_word;
} data_word_t;
data_word_t data;
```

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

```
data_word_t  data = ’{3,0};
```

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

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

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

What is the benefit in FPGA design projects? Structures can substantially reduce the amount of code that is required to represent complex data in a design. When a structure is defined as a user-defined type within in a package, the same definition can be used in multiple modules, and passed through ports from one module to another. This both simplifies working with several signals in each module, and eliminates the risk of declarations being different in each module.

Unions

SystemVerilog unions allow a single storage space to represent multiple storage formats. SystemVerilog has three types of unions: a *simple union*, a *packed union*, and a *tagged union*. Only packed unions are currently synthesizable. Packed unions require that all members within the union be packed types of the same number of bits. Packed types are vectors and packed structures. Because all members within a packed union are the same size, it is legal to write to one member (format) of the union, and read back the data from a different member.

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

What is the benefit in FPGA design projects? There are limited places where unions might be beneficial, but in those circumstances unions can reduce the amount of logic gates generated by synthesis compilers. As shown in the example above, a union can represent a hardware register that can be used for different purposes at different times. That same code would have required modeling two separate registers in traditional Verilog, and then carefully configuring synthesis to create a shared register. Using a union removes the burden from the engineer of having to configure synthesis compilers for shared registers.

Parameterized types

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

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

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

Parameterized data types are synthesizable.

What is the benefit in FPGA design projects? As with the other user-defined types, parameterized types allows modeling more functionality in few lines of code.

Shared declaration spaces

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

Packages

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

- **`parameter`** and **`localparam`** constant definitions
- **`const`** variable definitions
- **`typedef`** user-defined types (discussed in later in this paper)
- Fully automatic **`task`** and **`function`** definitions
- **`import`** statements from other packages
- **`export`** statements of other packages

An example package is:

```
package alu_pkg;
  typedef enum {ADD, SUBTRACT, MULTIPLY, DIVIDE} optcode_t;
  typedef struct {...} packet_t;
endpackage
```

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

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

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

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

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

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

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

What is the benefit in FPGA design projects? The use of packages can significantly reduce redundant code by allowing tasks, functions, and user-defined types (discussed later in this paper) to be declared once and used in many different places. Packages can reduce the risk of inconsistency in duplicate code (e.g. a task written one way in one design block and differently in another design block), and the costly risk of duplicated code being updated in one place and not in another place as design features are added or modified. Packages also make it much easier to reuse work done for one project in future projects, thus reducing debugging and overall time-to-market of future projects. Some of the capabilities of packages could be mimicked in traditional Verilog using ‘include compiler directives, but that coding style required awkward file management and imposed difficult file order dependencies during compilation. The use of packages eliminates the tedious and error-prone limitations of old-style Verilog ‘include directives.

$unit

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

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

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

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

$unit can contain the same kinds of user definitions as a named package, and has the same synthesis restrictions. $unit is automatically visible to all design (and verification) blocks that are compiled at the same time.

What is the benefit in FPGA design projects? Sutherland HDL strongly discourages the use of $unit in any type of project. Although synthesizable, $unit can lead to spaghetti code that is difficult to debug, difficult to maintain, and difficult to reuse. Each invocation of the synthesis compiler creates a new $unit implicit package that is unique to that compilation. Declarations in the $unit created by one compilation are not visible in the $unit created by another compilation. This risk of multiple compilation units leads to file order dependencies during compilation, and often special invocation options for simulation or synthesis compilers. $unit does not provide any benefit over SystemVerilog packages. The use of packages instead of $unit is strongly recommended.

Data arrays

SystemVerilog extends the Verilog static array construct in several ways that are synthesizable. The two synthesizable array types are *packed arrays* and fixed-size *unpacked arrays* (SystemVerilog also has several types of dynamically sized arrays for use in verification code, which are not synthesizable).

Packed arrays

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

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

```
reg [31:0] bus1;       // array of 32 contiguous bits

bus1[15:7] = 8’hFF;    // select an 8-bit subfield
bus1[31]   = 1’b1;     // select a single bit
```

SystemVerilog allows bit-vectors (packed arrays) to be declared with multiple dimensions. A packed array with multiple dimensions is still a vector made up of contiguous bits. That vector, however, is now divided into subfields.

```
logic [3:0][7:0] bus2;  // array of 32 contiguous bits

bus2[1] = 8’hFF;        // select an 8-bit subfield
bus2[3][7] = 1’b1;      // select a single bit
```

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

What is the benefit in FPGA design projects? Dividing a vector into subfields that can be indexed with a single index can simplify code that frequently accesses subfields within a vector. The code is simple to read and maintain. The risk of a coding error is reduced when compared to using old-style Verilog part selects, where an engineer must mentally calculate and keep track of the bit numbers that make up a subfield. On the other hand, if a design must frequently access individual bits of a vector, then the old-style Verilog declaration is easier to use and more self-documenting. An advantage of SystemVerilog is that it is still Verilog, and both old coding styles and new coding styles can be freely intermixed.

Unpacked arrays

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

- Arrays of user-defined types
- Copying entire arrays, or slices of arrays
- Assigning literal values to entire arrays or slices of arrays
- Passing arrays through module ports and to tasks and functions
- Array traversal **`foreach`** loop

A full description of each of these enhancements is beyond the scope of this paper. All of these enhancements are synthesizable, but with restrictions to meet behavior that can be represented in physical hardware.

What is the benefit in FPGA design projects? Old-style Verilog supported single and multi-dimensional arrays, but with a major limitation. Only one element of an array could be manipulated at a time. It was not possible to pass an array, or a pointer to an array, from one module to another, or into a task or function. To copy one array into another array required writing loops that indexed through an array and copied one element at a time. Most of the array enhancements listed above are based on the ability to manipulate all of, or portions of, an array in a single line of code. This new ability is another way in which SystemVerilog can significantly reduce the number of lines of code required to model and verify complex hardware designs. The following example illustrates how just 3 lines of code (plus the data declarations and module ports) can model a register that represents a large number of hardware gates. This same functionality would have required many dozens of lines of code in traditional Verilog, including a separate port and separate assignments for each member of the `uni_t` structure and each element of the `Payload` array.

```
package design_types;
  typedef struct {
    logic [ 3:0] GFC;
    logic [ 7:0] VPI;
    logic [15:0] VCI;
    logic        CLP;
    logic [ 2:0] T;
    logic [ 7:0] HEC;
    logic [ 7:0] Payload [0:47];
  } uni_t; // UNI cell signal definitions
endpackage

module transmit_reg (output design_types::uni_t  data_reg,
                     input  design_types::uni_t  data_packet,
                     input  logic                clock, resetN);
  always @(posedge clock or negedge resetN)
    if (!resetN) data_reg <= '{default:0};
    else         data_reg <= data_packet;
endmodule
```

Module ports

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

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

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

```
package user_types;
  typedef enum bit (FALSE, TRUE} bool_t;
  typedef struct {    // declared in $unit space
    logic [31:0] i0, i1;
    logic [ 7:0] opcode;
  } instruction_t;
endpackage

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

What is the benefit in FPGA design projects? One important benefit to passing compound data values such as structures and arrays is the simplification of RTL code. A design engineer can model at a higher level of abstraction, passing large bundles of information from one module to another using simple port declarations. Synthesis compilers can then expand this abstract, compound port into the many discrete ports, automating what in traditional Verilog required the design engineer to do manually.

Another benefit is the use of variables on input ports. Old Verilog required that only net types could be used on input ports. Nets can have multiple driver (with the exception of the **`uwire`** type), which a module can "*back drive*" its input ports. Typically this is an unintentional design error. Variables have a semantic restriction that they can only be assigned a value from a single source. The input port is a source, which means any other assignment to the variable will be illegal, instead of a functional bug in the design.

Casting

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

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

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

What is the benefit in FPGA design projects? Traditional Verilog did not have a cast operator, but, as a loosely typed language, casting could be accomplished using an assignments to a temporary variable of the target data type. The drawback of this old style was not only the extra code required, but the risk of another engineer, perhaps in a future project, not realizing the purpose of the temporary variable and removing it from the code.

The following example models a floating point multiplier and an integer adder in traditional Verilog. Without the use of a temporary variable, synthesis would infer a floating point adder instead of the intended integer adder.

```
real r;
reg [ 2:0] b;
reg [31:0] a;
reg [31:0] temp;

always @(a or b or r) begin
  temp = r ** b;
  y = a + temp;
end
```

The SystemVerilog code for this same model is:

```
real r;
logic [ 2:0] b;
logic [31:0] a;

always_comb
  y = a + logic [31:0]'(r ** b);
```

Operators

SystemVerilog adds a number of operators that are synthesizable:

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

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

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

  Synthesis compilers might restrict the use of assignment operations within a compound expression.

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

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

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

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

SystemVerilog adds several other operators to traditional Verilog, but these operators are not yet widely supported by synthesis compilers, and therefore not discussed in this paper.

What is the benefit in FPGA design projects? As with many other SystemVerilog enhance-ments, the primary benefits are reduced lines of code, more intuitive self-documenting code, and a reduced risk of coding errors.

Procedural blocks

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

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

The **`always_comb`** and **`always_latch`** procedural blocks automatically infer a complete and correct logic sensitivity list. The procedures also semantically enforce certain synthesis rules, one of which is that no other procedural block can write to the same variable to which the procedural block writes. **`always_ff`** also enforces a number of synthesis restrictions.

What is the benefit in FPGA design projects? The advantages of these specialized procedural blocks are very important. The old-style Verilog **`always`** procedure is meant to be a general purpose *simulation* coding block that is used in both RTL models and testbenches. It is an "anything goes" procedure with no language restrictions. Synthesis, on the other hand, must impose a number of restrictions on the general purpose **`always`** procedure. It is not uncommon in traditional Verilog to spend hours or days modeling and simulating complex hardware at the RTL level, only to find out that the code won't synthesize. By enforcing key synthesis restrictions, the SystemVerilog specialized procedural blocks will not simulate if the block will not synthesize.

Even worse than finding out that an RTL model will not synthesize is when the RTL code does synthesize, but not as intended. The burden is then placed on having very good verification code that will detect the incorrect functionality, which is often followed by long hours of debugging the cause of the problem. In traditional Verilog, synthesis compilers had no way to know what type of logic a designer intended. All the compiler could do was to analyze the body of the procedure and infer (guess) what was intended. The engineer then had to manually inspect the synthesis reports and resultant circuitry to determine if synthesis guessed correctly. The process was tedious and error prone. With the SystemVerilog specialized procedural blocks, there is no guessing. Synthesis compilers must still analyze the body of the procedure, just as with old-style Verilog. If, however, the functionality the RTL code does not match the procedure type, the synthesis compiler can and will issue a warning message. The engineer no longer has to spend precious engineering time inspecting the gate level circuitry to determine what synthesis inferred.

Note that these specialized procedural blocks do not force or guarantee that synthesis will generate the intended type logic. If the actual functionality within these specialized procedural blocks does not represent the type of the procedure, synthesis will generate gate logic that represents the actual functionality and issue a warning that the actual functionality does not match the intent indicated by the type of procedural block.

Programming statements

SystemVerilog substantially extends the programming capabilities of traditional Verilog. Many of these extensions are targeted towards testbench programming. Major EDA companies often place so much marketing emphasis on the testbench capabilities that engineers overlook the fact that SystemVerilog also contains important new programming features for modeling synthesizable hardware. One of the more notable synthesizable enhancements to Verilog programming statements is **`unique case`** and **`priority case`** decision statements. These statements address two limitations in the old Verilog HDL.

First, traditional Verilog defined that `if`...`else` and `case` statements must be evaluated in source code order. SystemVerilog does not change this rule. In hardware implementation, this would require extra, priority encoding logic. Synthesis will optimize out this extra logic if it can determine that all branches of the decisions are mutually exclusive (unique). There are many situations, however, where synthesis cannot determine mutually exclusive behavior. The design engineer must then add a "*parallel_case*" synthesis directive, called a pragma, to direct synthesis to omit the priority encoded logic. This pragma is hidden in a comment that is ignored by simulation but parsed by synthesis. For example:

```
case (state)   //synthesis parallel_case
```

Second, the Verilog language does not require that a decision statement always execute a branch of code. Should this occur, synthesis will add latches to the implementation. If the engineer knows that the values that are not decoded can never occur, the engineer must add a '*full_case*" synthesis pragma to prevent synthesis from adding the latches. Again, the pragma is specified as a comment, which is ignored by simulation.

```
case (state)   //synthesis full_case
```

These synthesis pragmas are fraught with dangers, and are the subject of numerous conference papers and various engineering conferences. At the root of the dangers is that fact that, because the pragmas are hidden in comments, simulation does not check that case statements actually behave in the way the pragmas specify.

The **`unique`** and **`priority`** case statements instruct both simulators and synthesis compilers as to the type of hardware intended. Tools can use this information to check that the code properly models the desired logic. With **`priority case`**, all simulators must issue a warning message if the case statement is evaluated and no branch is executed. With **`unique case`**, simulators must report an error if two code branches could be true at the same time or if the case statement is evaluated and no branch is executed.

What is the benefit in FPGA design projects? For synthesis, there is no specific benefit. A **`priority case`** statement turns on the `full_case` pragma, and a **`unique case`** statement turns on both the `full_case` and `parallel_case` pragmas. The benefit is in simulation, and it is an important benefit. As programming statements, simulation will issue warning messages, should the functionality of the case statement not behave as full case or parallel case. These warning messages can save untold hours of debugging unintended behavior in a post-synthesis gate-level design. Or, worse, bugs in a completed and shipped product.

Interface Ports

SystemVerilog adds interface ports to Verilog. An interface port is a compound port type that can include data type declarations (both nets and variables), user-defined methods, and procedural code. The syntax, semantics, and capabilities of interfaces is a large topic that cannot be addressed in this paper, due to space limitations. In brief, interfaces provide a means for designers to centralize the definition of a bus, as opposed to having the definition scattered in several modules throughout the design. Synthesis can then distribute the bus hardware appropriately throughout the design.

What is the benefit in FPGA design projects? The value of interface ports depends on the type of design and whether a communication bus protocol is used within the design. Sutherland HDL's experience is that interface ports have little advantage in most types of designs, in part because synthesis compilers only support a subset of interface port capabilities.

Synthesis Support

All of the constructs presented in this paper are supported by the major FPGA synthesis compilers: Synopsys Synplify-Pro, Synopsys DC, and Mentor Precision. These tools do impose restrictions on some of the constructs. There is some variance in these restrictions between these commercial tools. FPGA design engineers using any of these commercial synthesis compilers can — and should — take advantage of these SystemVerilog enhancements to traditional Verilog.

At the time this paper was written, the support for SystemVerilog in proprietary synthesis compilers provided by FPGA vendors is severely lacking. Xilinx supports a very limited subset of SystemVerilog, but has plans for increasing this support in the near future. Altera also has very limited support for SystemVerilog, and has not made plans for further support public. FPGA design engineers who are using proprietary FPGA synthesis compilers will need to wait for better support, in order to benefit from SystemVerilog's enhancements to traditional Verilog.

Summary

SystemVerilog adds a large set of enhancements to the traditional Verilog Hardware Design Language. Some of these enhancements are for verification, while others are for modeling hardware. The focus of this paper has been on the synthesizable hardware modeling features in SystemVerilog. The benefits of the synthesizable enhancements that SystemVerilog adds to the traditional Verilog HDL have been discussed. These language features can reduce the amount of code required to model large, complex designs and at the same time eliminate or detect coding errors that appear to simulate correctly, but could result in an incorrect or non-optimal synthesized hardware design. There are many advantages to using these SystemVerilog enhancements in FPGA design work.

A reason for not yet adopting these synthesizable SystemVerilog features has also been discussed. SystemVerilog is well supported by major commercial FPGA synthesis compilers, but current support in proprietary FPGA synthesis compilers is somewhat limited.

The answer to the question, *"What, if anything, in SystemVerilog will help me with FPGA-based designs?"* is not a simple one. SystemVerilog has a great deal to offer for FPGA design, but the synthesis compiler you are using might not support some of those features — at least not at the time this paper was prepared.

References and resources

[1] *"IEEE 1800-2009 standard for the SystemVerilog Hardware Description and Verification Language"*, IEEE, Pascataway, New Jersey, 2009. ISBN 978-0-738-16130-3 (PDF version).

[2] *"IEEE 1800-2005 standard for the SystemVerilog Hardware Description and Verification Language"*, IEEE, Pascataway, New Jersey, 2005. ISBN 0- 7381-4811-3.

[3] *"1364-2005 IEEE Standard Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN:0-7381-2827-9.

[4] *"1364.1-2002 IEEE Standard for Verilog Register Transfer Level Synthesis"*, IEEE, Pascataway, New Jersey. Copyright 2002. ISBN:0-7381-3502-X.

[5] *"SystemVerilog for Design, second edition"*, by Sutherland, Davidmann, and Flake. published by Springer, Boston, Massachusetts, 2006. ISBN: 978-0-387-33399-1.

[6] "*A Proposal for a Standard Synthesizable Subset for SystemVerilog-2005: What the IEEE Failed to Define*", by Stuart Sutherland. presented at DVCon, March 2006. Available at http://www.sutherland-hdl.com/papers/2006-DVCon_SystemVerilog_synthesis_subset_paper.pdf

[7] *"SystemVerilog: A Design & Synthesis Perspective"*, by Karen Pieper. Presented at the 2006 Synopsys Users Group Conference, Boston, Massachusetts.

[8] *"Towards a Practical Design Methodology with SystemVerilog Interfaces and Modports"*, by Jonathan Bromley. Presented at the 2007 Design and Verification Conference and Exhibition (DVCon), San Jose, California.

[9] *"Seamless Refinement from Transaction Level to RTL using SystemVerilog Interfaces"*, by Jonathan Bromley. Presented at the 2008 Synopsys Users Group Conference, San Jose, California.

[10] *"Building Polymorphic Modules with Synthesizable SystemVerilog Constructs"*, by B. Hook. Presented at the 2008 Synopsys Users Group Conference, San Jose, California.

[11] *"SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes"*, paper by Stuart Sutherland, Synopsys Users Group (SNUG) conference, March 2005. Available at http://www.sutherland-hdl.com/papers/2005-SNUG-paper_SystemVerilog_unique_and_priority.pdf