What’s Next for SystemVerilog in the Upcoming IEEE 1800 standard

Dave Rich
Dave.Rich@siemens.com

***Abstract*-The last revision of the SystemVerilog LRM was completed in 2016 and published as IEEE 1800-2017. In the time since, tool vendors have continued to extend and make clarifying changes to their implementations. That leaves users and other tool vendors with unclear specifications on how to interpret SystemVerilog code. The next revision of the standard (P1800-2023) intends to address many of these issues to keep the language current. This paper reviews some of the high-level goals for the next revision as well as highlights a few key enhancements.**

I. Introduction

Between the Accellera Systems Initiative and the IEEE, there have been seven revisions of the SystemVerilog Language Reference Manual (LRM) over the past 20 years [1]. Five of those revisions were in the first 10 years. Many users avoid adopting SystemVerilog because feature support from different tools and vendors of the rapidly changing LRM had been so inconsistent. To this day, people continue using Verilog-1995 [2] syntax and avoid using features added by Verilog-2001 [3] (e.g., ANSI-style ports and the power operator). So, brakes were put on the SystemVerilog LRM process, giving vendors a chance to catch up and giving users the stability they wanted. The last time any of the IEEE SystemVerilog technical committees met to add changes to the LRM was at the end of 2016. The result of this work was the publication of the IEEE 1800-2017 [4] standard.

However, technology never stands still. Over the last several years, vendors have made extensions to their tools based on demands from customers, and the users are left with a hodgepodge of features with no or incomplete documentation. For example, some tools allow many more built-in functions (like $sformatf) to be used in a constant expression even though the LRM explicitly does not allow it. You won’t know which ones your tool supports until you try them. Other users simply won’t wait for any extensions and begin working around language limitations by creating extra code packages or incorporating other languages into their flow (Chisel, Perl, Python, Ruby,…). Don’t expect SystemVerilog to be the number one language choice for every design and verification project out there. Still, every language must evolve to improve and stay relevant. And developers want to protect their investment in the verification IP they develop for as long as possible.

Under the auspices of the IEEE, a project “P1800” to craft the next revision of the SystemVerilog was created in late 2019 [4]. “P” is for “Proposed standard” and the current plan is to finalize the work and make it available to the public in 2024. After a two-year delay due to the worldwide pandemic, a “Working Group” made of end users and tool vendor companies formed to submit issues on behalf of themselves or indirectly through affiliated companies. This IEEE working group uses an issue tracking system called *Mantis* [5]. The group enters issues into the Mantis database that are categorized into a few major types:

- Enhancements—New features that extend the capabilities of the current language. These enhancements may already be implemented in some tools or be completely new to the language. In many cases they have been “borrowed” from other languages, so their use models are well known.
- Errata—Obvious mistakes in the current LRM. They could be as small as typographical editing errors or as large as contradictions in the text between LRM sections because a previous update failed to capture all places that needed to be changed. Sometimes the current wording of the LRM is obviously incorrect and tools have implemented what the LRM should have said in the first place.
- Clarifications—Missing or misleading text for a particular use case where the LRM is ambiguous. This normally does not involve a change to a tool’s behavior unless tools implemented something very different from what was originally intended.

As of the publication date of this paper, 26 enhancements, 109 errata, and 61 clarifications have been approved by the committee for the next revision of the standard. These categories are not always distinct. Many enhancements uncover existing errata issues or require other clarifications. Sometimes an issue filed as an enhancement becomes just a clarification. The sections that follow highlight just a few significant issues from each category.

II. Enhancements

A. Extending coverpoints (Mantis 4703)

One of the most often requested enhancements, and my primary motivation for getting this project started is extending covergroups. Covergroups without inheritance is considered by many an unfinished feature. When extending a class, you can add new class members and override existing ones in the base class; the same holds true for constraints in classes. When embedding covergroups in classes, they are effectively additional class members. It seems obvious that the same principles should have been applied to covergroups. This comes up frequently when creating verification IP base class libraries and these libraries need to be extended to handle customizations for different usage configurations.

Two of the most critical features enabled by this enhancement are:

- Adding a new coverpoint in an extended class and then crossing it with an existing coverpoint in the base class. Currently, you must replicate the coverpoint in the base class to the extended class and cross it with the new coverpoint.
- Replacing the bin structure of an existing coverpoint or removing it entirely.

The enhancement would allow syntax such as:

```
class pixel;
  bit [7:0] level;
  enum {OFF,ON,BLINK,REVERSE} mode;
  covergroup g1;
    a: coverpoint level;
    b: coverpoint mode;
  endgroup
  function new();
    g1 = new;
  endfunction
endclass

class colorpixel extends pixel;
  enum {red,blue,green} color;
  covergroup extends g1;
    b: coverpoint mode {    // the coverpoint 'b' from the base class is changed
      ignore_bins ignore = {REVERSE};
    }
    cross level, color;     // 'level' comes from the base class
  endgroup
endclass
```

B. Unpacked array mapping function (Mantis 7610)

SystemVerilog allows you to work with arrays (aggregates) as a whole. However, it limits you to comparison and assignment operators, as well as working only on unpacked arrays having equivalent element types. Borrowing from other languages such as Python, the array map function allows you to cast each element to a new type, as well as perform any operation on each element.

Like the other array manipulation methods (see section 7.12 of [4]) , the map() method iterates over each element using a `with()` expression and provides implicit variables *item* and *item.index*. It returns a new array with each element having the type of that expression. Here are a few examples:

```
int A[3] = {1,2,3};
byte B[3];
int C[3];
// assigns and casts array of int to an array of byte
B = A.map() with ( byte'(item) );
```

```
// increments each element of the array (use b instead of item)
B =  B.map(b) with ( b + 8'b1 ); // B becomes {2,3,4}
// Add two arrays
C = A.map(a) with (a + B[a.index] );  // C becomes {3,4,5}
```

C. `ifdef Boolean combination of identifiers (Mantis 1084)

This request precedes the publication of the first SystemVerilog LRM [3]. Enhancements related to the preprocessor have been very difficult to address because of many ambiguities in this area of the existing LRM (See Mantis 1014 in the Clarifications section).

Verilog has a simple `` `ifdef `` scheme where you specify a single macro name to test if it is defined or not. There is no way to combine testing multiple definitions with nesting multiple `` `ifdef ``s or creating intermediate define macros. Assuming there are two potential macro definitions A and B, in the existing Verilog LRM you would have to write

```
// AND
`ifdef A
   `ifdef B
     `define A_and_B
   `endif
`endif
`ifdef A_and_B
   // code for AND condition
`endif
// OR
`ifdef A
   `define A_or_B
`endif
`ifdef B
   `define A_or_B
`endif
`ifdef A_or_B
   // code for OR condition
`endif
```

With this enhancement, you can use a simple Boolean expression enclosed in parentheses

```
`ifdef (A && B)
   // code for AND condition
`endif
`ifdef (A || B)
   // code for OR condition
`endif
```

Note the value of the definition is not being tested, only the state of its definition. In C/C++, this would be equivalent to:

```
#if defined(A) && defined(B)
```

D. Add support for multiline strings (Mantis 7308)

The basic premise of this request is allowing multi-line and quoted string within a string literal.

```
string x = """
This is one continuous string.
Single ' and double " can
be placed throughout, and
only a triple quote will end it.
"""
```

This feature makes it easier to write informative or self-documenting messages so they can be easily read and maintained within the source code.

The enhancement request seemed to be simple on the surface wound up entangled with many older issues surrounding it.

- Multi-line string literals in a text macro (Mantis 1397)
- Special characters in strings (Mantis 1507)
- “Printable” characters (Mantis 7562)

E. Real number modeling (Mantis 7295 and 7669)

SystemVerilog was formed around the concept of being a *digital* logic design and verification language. That means most constructs and operators expect expressions to break down into discrete integral or Boolean values. Constraints, assertions, and covergroups rely heavily on equality expressions. Real numbers introduce irregularities with expressions like (0.1 + 0.2) == 0.3 can never evaluate true because these floating-point values cannot be accurately represented in a binary number system.

Adding real numbers to random variables and constraints involves lifting the existing restrictions from integral to allow for real types. Distributions of floating-point ranges can only deal with weights over the entire rage using **`[1.0:3.0]:/weight`**. It makes no sense to allow the other distribution **`:=`** syntax which would have applied the weight to every value in the range. There are an infinite number of discrete values in the range between two real numbers; thus, the weight of the range would be infinite.

Adding real numbers to covergroup requires a little more consideration. Coverage bins by their very nature represent discrete sets of values. A new option `real_interval` helps break up floating-point ranges into nonoverlapping bin sets.

```
coverpoint r {
  type_option.real_interval= 0.01;
  bins b[] = {[0.75:0.85]}; // 10 bins
                            // b[0] 0.75 to less than 0.76
                            // b[1] 0.76 to less than 0.77
                            // . . .
                            // b[9] 0.84 to less than or equal to 0.85
}
```

F. Chaining of method calls (Mantis 2735)

This feature allows you to use the result of a function to serve as an intermediate variable for selecting a member of the result. It is mainly used when the result is a handle to a class.

It may come as a surprise that this was not already included in the LRM as many tools have been supporting some form of it for a while. Now it is properly defined. One of the major issues with this feature is keeping backward compatibility with Verilog. It allows hierarchical reference to static variables declared inside a function from outside that function. To distinguish a hierarchical reference from an intermediate variable, you must use parentheses “()” in the function call even when there are no arguments.

```
class A;
  int member=123;
endclass
module top;
  A a;
  function A F(int arg=0);
    int member; // static variable uninitialized value 0
    a = new();
    return a;
  endfunction
  initial begin
    $display(F.member);   // 0 - No "()", Verilog hierarchical reference
    $display(F().member); // 123 - With "()", implicit variable
   end
endmodule
```

The first `$display` statement prints the value of a hierarchical reference to the static variable `top.F.member`, whose value is 0. The second `$display` statement calls the function `F`, whose return value gets used to select the class A variable `member`, whose value is 123.

G. Adding static ref arguments (Mantis 2583)

A **`ref`** argument of a time-consuming task tracks values updating between the actual arguments passed *into* a task and formal arguments defined *inside* the task during the lifetime its call. In contrast, actual **`input`** argument passes its value only at the point in time when the task gets called. Implementing a **ref** argument places more restrictions than passing an arguments value because the code inside the task has no knowledge of the actual argument's declaration. One restriction is the actual and formal argument's type must match; it is not enough just to be assignment compatible. Another restriction is the task must assume the actual argument's lifetime is automatic. The consequence of that means there are several other restrictions on what you cannot do with variables declared with automatic lifetimes. An argument passed by reference cannot

- Be the target of a nonblocking assignment.
- Have a reference on the LHS or RHS of `force` statement.
- Be referenced from within a `fork join_any/join_none` process.

This enhancement adds a **`static`** qualifier to a formal **`ref`** argument so the task or function can be assured the actual argument has a **`static`** lifetime (it would be an error to pass a variable with an automatic lifetime as an actual argument).

```
module top;
  function void monitor(ref static logic arg);
    fork // the reference to arg only becomes legal with a static qualifier
      forever @(arg) $display("arg changed at time %t", arg, $realtime);
    join_none
  endfunction

  logic C;
  initial monitor(C);
endmodule
```

III. Errata

A. @(clocking_block_name) is unequal to its associated clocking event (mantis 7172)

The current LRM has this brief example in section 14.10 (Clocking block events)

```
clocking dram @(posedge phi1); inout data;

output negedge #1 address; endclocking
```

The clocking event of the `dram` clocking block can be used to wait for that particular event:

```
 @(dram);
```

The preceding statement is equivalent to `@(posedge phi1).`

However, this cannot be true since section 14.13 says the dram event gets triggered in the observed region and the `posedge phi1` gets triggered in the active region. So, the example was expanded with a more detailed explanation:

```
always @(posedge phi1) $display("clocking event");
always @(dram) $display("clocking block event");
```

> The first **`always`** procedure in the preceding example executes in the same Active or Reactive event region as the positive edge of `phi1`. In contrast, the second **`always`** procedure executes after the Observed region following that Active or Reactive event region. This behavior can be used to avoid race conditions when sampling input data, see 14.13 for more details.

B. non-integral function actual argument allowed in a constraint expression (mantis 2841)

The current LRM says this about constraints in section 18.3:

> "Constraints can be any SystemVerilog expression with variables and constants of integral type (e.g., bit, reg, logic, integer, enum, packed struct)."

Section 18.5.13 (Constraint guards) already contradicts this with object handles, but there are many scenarios where there is not a problem using non-random non-integral expressions. For example, a real typed threshold variable requires the user to create and initialize a separate integral variable. Only this can be used in the constraint.

```
class A;
  rand bit [8:0] randvar;
  real threshold;
  int intvar;
  function void pre_randomize();
    intvar = int'(threshold);
  endfunction
  constraint c{ randvar < intvar; }
endclass
```

Mantis 2841 allows the above to be written as:

```
class A;
  rand bit [8:0] randvar;
  real threshold;
    constraint c{ randvar < int'(threshold); }
endclass
```

IV. Clarifications

A. Unclear which compiler directives must be alone on a line (Mantis 1014)

Like Mantis 1084, this is a clarification request carried over from the Verilog 1364 standardization effort. Compiler directives have always been a weak point and considered underspecified in the LRM. Another enhancement request for multi-line strings brought the priority of this issue to the forefront. One of the most common places this comes up is with conditional compilation

```
`ifdef NAME text `endif
```

Support for this structure of directives was unclear in the existing LRM and has been clarified. Most tools have already been supporting this.

B. Definition of term "blocking statement" (Mantis 225)

One of the oldest issues to be clarified, the term “blocking” is well understood by experienced users but was never properly defined. This is particularly a problem because the word “block” is a homonym with multiple unrelated meanings. Within this standard, a “block” could be used as a noun to describe a set of connected conceptual elements, or as a verb, a construct that suspends execution of a process.

The term “blocking statement” in most cases refers to a construct having the potential to suspend a process. But this is further complicated using the terms “non-blocking assignment” which is never a blocking statement, and a “blocking assignment” which is only sometimes a blocking statement.

The three terms, “blocking statement”, “non-blocking assignment”, and “blocking assignment” have been clarified in the upcoming standard, as shown in by the illustrative examples below:

```
A <= #1 2; //  non-blocking assignment, not a blocking statement
A = 3;     //  blocking assignment, not a blocking statement
A = #4 5;  //  blocking assignment, a blocking statement
```

C. Packed array shortcut – (Mantis 325)

This issue originally started out as an enhancement request.

Verilog declares array ranges with pairs of index values. For packed arrays or integral vectors, the left index value represents the most significant bit(MSB), and the right index is the least significant(LSB). Integral vectors are normally declared using a LSB 0 bit numbering scheme when the index of the LSB is 0 (or little bit endian), and the index of the MSB is the width of the vector minus 1. Thus, the bit index position i represents the 2^i value of the vector.

```
bit [7:0] v = 8’b10000000; // 8-bit wide vector, v[7] = 1, 2^7 = 128
```

Unpacked arrays are also declared with pairs of index values. However, since most references to unpacked arrays only select one element at time, or select the entire array as a whole, element 0’s position is not as important. It does matter when loading the array from an external file or streaming to another array with a different index ordering.

SystemVerilog has a shortcut for declaring unpacked arrays ranges with a single number, as in C/C++. The following two declarations are equivalent:

```
int A[0:99];
int A[100]; // This shortcut implies that the left index is 0
```

Note that `int` is a shortcut for `bit signed [31:0]` with the index 0 bit being the LSB. There are also several implicit packed declarations that use the index 0 for the LSB numbering (packed structs and unions, concatenation expression part selects). This enhancement request asked for the same shortcut syntax for declaring packed arrays with a single number. For example:

```
bit [8] v = 8’hab;    // Proposed shorthand for bit [7:0] v = 8’hab;
```

But there was uncertainty as to whether the shortcut should be little bit endian to match packed arrays or big bit endian to match the index ordering of the unpacked array shortcut. In the end, the committee decided the potential confusion this enhancement would create was not justified and in fact the LRM now provides further clarity stating the array declaration shortcut only applies to unpacked arrays.

IV. Summary

This paper has presented just a few of the many changes in the upcoming IEEE SystemVerilog standard. I have pointed out the process the Working Group goes through to address each individual issue. Many issues the group goes through and not discussed here require no changes because they were found to be already addressed by other issues, unnecessary because they were easily covered by other features, or out of scope for the current updated standard proposal. Note the P1800-2023 is not a formally approved IEEE standard. All the features mentioned here are subject to final IEEE balloting approval.

V. References

[1] Goering, Richard, "Successor to Verilog approved as Accellera standard," 4 June 2002. [Online]. Available: https://www.eetimes.com/successor-to-verilog-approved-as-accellera-standard/. [Accessed 26 October 2022].

[2] "IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language," in IEEE Std 1364-1995 , pp.1-688, 14 Oct. 1996.

[3] "IEEE Standard Verilog Hardware Description Language," in *IEEE Std 1364-2001*, pp.1-792, 28 Sept. 2001.

[4] "IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language," in *IEEE Std 1800-2017*, pp.1-1315, 22 Feb. 2018.

[5] Accellera, "Accellera Mantis Database," [Online]. Available: https://accellera.mantishub.io/my_view_page.php. [Accessed 25 October 2022].