# ASIC Design

*Keep sharing , Keep Exploring*

# SystemVerilog: The let construct

By sharvil111 on December 26, 2015
Hi all,

Today I am gonna discuss about the *let* construct in SystemVerilog.

Many times, when we encounter some repetition of code, in 90% of cases, an obvious alternative would be to create **tasks/functions**.

Lets start with an example to compare two variables of *int* type:

```
function void compare(int a,int b);
$display("The result is : %0s",(a==b) ? "Pass" : "Fail");
endfunction

// Usage
compare(5,4); // The result is : Fail
compare(9,9); // The result is : Pass
```

The function works fine as long as *a* and *b* are *int* type variables. But sometimes things get really more fuzzy. When using *string* or some other datatype argument, this function might not work. At that time, either explicit static cast or declaring a second function must be done.

In general, for a testbench coding, you may want to replace the part of code with some substitute. Like when you want to declare a single function in many classes, or some string concatenation, or a generalized function for comparison of all the datatypes, etc. At that time, **macros** come into picture:

```
`define compare(a,b)\
$display("The result is : %0s",(a==b) ? "Pass" : "Fail");\

// Usage
`compare(5,4); // The result is : Fail
`compare("string1","string2"); // The result is : Fail
`compare("string1","string1"); // The result is : Pass

`define suffix(name)\
{name,"_cp"}\

// Usage
`suffix("fun") // fun_cp
```

Here we have generalized/customized the datatype of input arguments. Any datatype can be passed as actual argument. Now we have a general function to compare *string*,*int*s, etc. But what if we want to have different macros with same name?

Here are some of the **disadvantages** of using macros:

1. Extensive use of macros, makes the code **difficult to understand**, the code becomes **unreadable**. Hence, **difficult to debug**.

2. Macros are the **pre-processor directives**, expanded at compile time. So, they **make compilation slower**.

3. There cannot be two or more macros with same name and different behavior from different parts of code. That is, their **scope is global**.

Reading from SystemVerilog LRM section 22.5.1 :

> **Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.**

There is no such thing like local macro, which is local to a particular module,program,interface or package. This seems to be a limitation in some specific cases.

Here comes the use of ***let* construct** in SystemVerilog to tackle, global scope of macro. Using the let construct, we can implement same sort of customization that were done by text macros.


Reading from SystemVerilog LRM 1800-2012 section 11.13:

> **let declarations can be used for customization and can replace the text macros in many cases. The let construct is safer because it has a local scope, while the scope of compiler directives is global within the compilation unit.**

In *let* construct, the formal arguments may optionally be typed and also may have optional default values just like macros.

The let body gets expanded with the actual arguments by replacing the formal arguments with the actual arguments.

From a source on internet, following are reasons to introduce this construct:

1. To provide a **safer alternative** to compiler directives.

2. To provide a **template** for immediate assertions.

3. To enable processing argument lists of **arbitrary length**. Though I've not seen any practical example of this use.

The internal variables used in a *let* that are not formal arguments are resolved according to the scoping rules from the scope in which the let is declared.

In the scope of declaration, a let body shall be defined before it is used. No hierarchical references to let declarations are allowed.

The simple tasks like compare, appending names, etc. can be performed by *let* as follows:

```
// in one module/class
let compare(a,b) = (a==b) ? "Pass" : "Fail";

// Usage
$display("Result is : %0s",compare(5,4)); // Result is : Fail
$display("Result is : %0s",compare("string1","string2")); // Result is : Fail

// in some different module/class
let compare(a,b) = (a==b) ? "Pass_test" : "Fail_test";

// Usage
$display("Result is : %0s",compare(5,4)); // Result is : Fail_test
$display("Result is : %0s",compare("string1","string1")); // Result is : Pass_test

// in one module/class
let suffix_l(name) = {name,"_cl"};

// Usage
suffix_l("fun"); // fun_cl

// in some other module/class
let suffix_l(name) = {name,"_mod"};

// Usage
suffix_l("fun"); // fun_mod
```

Here, the let statement have its **local scope**, hence we can have different statements in different scopes. let can be defined in any of the following:

— A module
— An interface
— A program
— A checker
— A clocking block
— A package
— A compilation-unit scope
— A generate block
— A sequential or parallel block
— A subroutine

One of the **disadvantage** of *let* is that it cannot be used as a **multiline construct**. While macros uses the backslash(\) operator for multiple line logic implementation. Because of this, *let* is a **rarely used** construct. But its still very helpful in case of single line logic implementation.

SystemVerilog IEEE 1800-2012 (https://blogs.mentor.com/verificationhorizons/blog/2013/02/25/get-your-ieee-1800/) section 11.13 shows example of let construct in different scopes. Please refer to Let construct (http://www.eda.org/sv-ac/hm/att-3929/letconstruct1728_070503.pdf) pdf for detailed information on *let* construct.