Is the simulator behavior wrong for my SystemVerilog code?

Weihua Han
1301 South Mopac Expressway
Austin, TX 78746
whan@synopsys.com

***Abstract*-Sometimes SystemVerilog users spend a lot of time debugging the unexpected simulation results and finally it turns out that the SystemVerilog Language Reference Manual(LRM) has different specifications as what the users had thoughts. In this paper we discuss some of these questions raised frequently by SystemVerilog users. Answers to these questions will help SystemVerilog users understand the language specification correctly and precisely. It may save designer's a lot of time from debugging the unexpected results.**

I. Introduction

In 2005 SystemVerilog was approved as the standard (IEEE-1800) for hardware description language and hardware verification language. Since then we have seen that SystemVerilog has been adopted by almost all the projects our customers worked on. But we also keep getting questions from our customers, asking "is the simulator wrong with my following SystemVerilog code?". Some of those questions were raised again and again from different users after debugging their simulation results for a while and could not decide if their code exposes a simulator bug, or it is a code issue. So for the designers who want to use SystemVerilog in their design and verification environment, it is very important to understand the language specification correctly and precisely. It may save a lot designers' time from debugging unexpected results.

In this paper, we will discuss some of those questions, each with some example snippets or problem descriptions. Then we will show what the corresponding IEEE-1800 LRM specifications are for these scenarios.

Following are the topics discussed in this paper:

- Wildcard package import and export
- Operator expression short circuiting
- Random stability
- Always_comb block inferred sensitivity list
- Longest static prefixes
- Wildcard equality operator
- 'b1 and '1
- Determining module port kind, data type, and direction
- Object constructor function call order
- Accessing array with an invalid index

II. Questions and Related SystemVerilog Specifications

A. Package import and export

For example:

```
package pkg2;
    import pkg1::*;
    export pkg1::*;
    ….
endpackage
module m1;
   import pkg2::*;
endmodule
```

A common expectation with above code is that all definitions in pkg1 are visible/available in module m1. This is not true. In SystemVerilog LRM 2005 version, it does not clearly specify what is the expected behavior on the chained package import. In SystemVerilog LRM 2009 and newer revisions, to avoid possible ambiguity, package export is specified: "By default, declarations imported into a package are not visible by way of subsequent imports of that package. Package export declarations allow a package to specify that imported declarations are to be made visible in subsequent imports."

And SystemVerilog LRM (we are using IEEE-1800-2012 as the reference in the rest of this paper) also clarifies the correct behavior on wildcard import and export: "An export of the form package_name::* exports all declarations that were actually imported from package_name within the context of the exporting package. All names from package_name , whether imported directly or through wildcard imports, are made available. Symbols that are candidates for import but not actually imported are not made available." Below is the example from LRM:

```
package p1;
  int x, y;
endpackage
package p2;
  import p1::x;
  export p1::*; // exports p1::x as the name "x"; p1::x and p2::x are the same declaration
endpackage
package p3;
  import p1::*;
  import p2::*;
  export p2::*;
  int q = x;
  // p1::x and q are made available from p3. Although p1::yis a candidate for import, it is not actually
  //imported since it is not referenced. Since p1::y is not imported, it is not made available by the export.
endpackage
```

Based on above specifications, with the original user's code, only the definitions referred in pkg2 will be visible/available in module m1.

B. *Operator expression short circuiting*

With following code

a = obj.b && c;

the user got "Null Object Access" runtime error but when the code was re-written as

a = c && obj.b;

the simulation just went through this line without any error. What could be the possible reason?

SystemVerilog LRM section "11.3.5 Operator expression short circuiting" has following specification: "Some operators (&&, ||, ->, and ?:) shall use short-circuit evaluation; in other words, some of their operand expressions shall not be evaluated if their value is not required to determine the final value of the operation.". Specifically for operator "&&" and "||", in section "11.4.7 Logical operators", LRM says: "The && and || operators shall use short circuit evaluation as follows:

— The first operand expression shall always be evaluated.
— For &&, if the first operand value is logically false then the second operand shall not be evaluated.
— For ||, if the first operand value is logically true then the second operand shall not be evaluated."

So in above code, when "c" is false, simulator won't need to evaluate "obj.b" if the code is "*a = c && obj.b*". That is the reason the simulator does not issue "Null Object Access" error.

C. *Random stability*

For example:

```
module m;
  initial $display("%m",,$urandom);
endmodule
module top;
  m m_inst1();
  m m_inst2();
  m m_inst3();
endmodule
```

Here is the simulation result:

```
  top.m_inst1  98710838
  top.m_inst2  98710838
  top.m_inst3  98710838
```

Why does the simulator generate same value for the all three instances?

SystemVerilog LRM specifies “Random Stability” in section 18.14. “The RNG (Random Number Generator) is localized to threads and objects. Because the sequence of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called random stability.
Random stability applies to the following:

— The system randomization calls, $urandom() and $urandom_range()
— The object and process random seeding method, srandom()
— The object randomization method, randomize()”

In section 18.14.1, it also specifies following in terms of how a module instance and its static processes (an initial block is a static process) get RNG “Initialization RNG. Each module instance, interface instance, program instance, and package has an initialization RNG. Each initialization RNG is seeded with the default seed. The default seed is an implementation-dependent value. An initialization RNG shall be used in the creation of static processes and static initializers”, “When a static process is created, its RNG is seeded with the next value from the initialization RNG of the module instance, interface instance, program instance, or package containing the thread declaration.”. With these specifications, the initialization RNGs of the 3 module m instances are seeded with the same default seed and the RNGs of the 3 initial blocks are seeded with the next value of those initialization RNGs. Therefore the 3 initial block RNGs have the same state when the $urandom is called, therefore the same random result is generated.

Users can manually specify different seeds to $urandom calls to generate different random results. For example, if using the system time as the seeds, users can use SystemVerilog DPI call to get system time:

```
//DPI C function
#include "time.h"
int unsigned mytime() {
  int unsigned seed;
      struct timespec ts;
  do {
    clock_gettime(CLOCK_REALTIME,&ts);
    seed = ts.tv_nsec;
  } while(seed == 0);  //random seed cannot be zero.
  return seed;
}
//SystemVerilog code
import "DPI-C" function int unsigned mytime();
module m1;
  initial $display("%m",,$urandom(mytime()));
endmodule
```

Note that with above solution, users always get different random results with different runs. If the users would like to keep the “random stability” among different runs, following code shows another possible solution:

```
module seeding_m;
    class c1_c;
       randc int unsigned seed;
       constraint no_zero {
          seed != 0;
       }
    endclass
    c1_c c1=new;
    function int unsigned seeding();
        c1.randomize();
        return c1.seed;
    endfunction
endmodule
module m1;
   initial $display("%m",$urandom(seeding_m.seeding()));
endmodule
```

D. Always_comb block inferred sensitivity list

For example:

```
module m1;
    logic a,b;
    function void f1();
        $display($stime,,"f1 call",,b);
    endfunction
    always_comb begin
       $display($stime,,"comb: a is:",,a);
       f1();
    end
    initial begin
        #1;  a = 1'b1;
        #10;b = 1'b1;
        #10; $finish;
    end
endmodule
```

Here is the simulation result:

```
         0 comb: a is: x
         0 f1 call x
         1 comb: a is: 1
         1 f1 call x
        11 comb: a is: 1
        11 f1 call 1
```

What is the reason that at time 11, the always_comb block seems triggered?

Besides the regular Verilog always block, SystemVerilog introduces several other kinds of always procedures, including always_comb, always_ff and always_latch. Always_comb is a very convenient way for modeling combinatioinal logical behavior. Users do not need to provide the sensitive list when using always_comb block. Instead, SystemVerilog LRM specifies always_comb sensitivities in section 9.2.2.1 as: “The implicit sensitivity list of an always_comb includes the expansions of the longest static prefix of each variable or select expression that is read within the block or within any function called within the block…”. The LRM explains it again in section 9.2.2.2.2: “The SystemVerilog always_comb procedure differs from always @* (see 9.4.2.2) in the following ways:

— always_comb automatically executes once at time zero, whereas always @* waits until a change occurs on a signal in the inferred sensitivity list.

— always_comb is sensitive to changes within the contents of a function, whereas always @* is only sensitive to changes to the arguments of a function.”

In the above example, variable “b” is read in function f1, so when b is updated, the always_comb block is triggered. And at time zero, although none of the variables gets updated, the always_comb block still executes once, based on the LRM specification.

E. Longest static prefixes

Is following code valid?

```
module m1;
  logic a, aa[2];
  logic b,c;
  always_comb aa[0]=b;  //always_comb block1
  begin:gen_blk1
    int ii=1;
    always_comb aa[ii]=c;   //always_comb block2
  end
  always @(*) a = b;
  always @(*) a = c;
endmodule
```

In SystemVerilog, a variable can be driven by one or more procedure statements in multiple procedure blocks, so in above code driving variable “a” from two always @(*) is valid. The last write determines the value of variable “a”. But it won’t be true with always_comb blocks. SystemVerilog LRM section 9.2.2.2 says “the variables written on the left-hand side of assignments shall not be written to by any other process.”. The LRM furthermore specifies on unpacked array assignment as “However, multiple assignments made to independent elements of a variable are allowed as long as their longest static prefixes do not overlap (see 11.5.3). For example, an unpacked structure or array can have one bit assigned by an always_comb procedure and another bit assigned continuously or by another always_comb procedure, etc.”.

Unfortunately for above code the user may still see that the simulator complains on the illegal combination of drivers of “aa” from the two always_comb blocks. The reason comes from how “longest static prefixes” is defined. In SystemVerilog LRM section 11.5.3 has following specification on longest static prefix:

The definition of a static prefix is recursive and is defined as follows:

— An identifier is a static prefix.
— A hierarchical reference to an object is a static prefix.
— A package reference to net or variable is a static prefix.
— A field select is a static prefix if the field select prefix is a static prefix.
— An indexing select is a static prefix if the indexing select prefix is a static prefix and the select expression is a constant expression.

Here is the LRM example in the same section.

```
Examples:
localparam p = 7;
reg [7:0] m [5:1][5:1];
integer i;
m[1][i] // longest static prefix is m[1]
m[p][1] // longest static prefix is m[p][1]
m[i][1] // longest static prefix is m
```

Based on above specifications, in the user’s code, the longest static prefix is “aa[0]” in always_comb block1 and the longest static prefix is “aa” in always_comb block2. They are overlapped.

To avoid this error user can modify the code as:

```
always_comb aa[0]=b;  //always_comb block1
begin:gen_blk1
  localparam ii=1;
  always_comb aa[ii]=c;   //always_comb block2
end
```

Localparam is a constant expression so the longest static prefix is now “aa[1]” in always_comb block2, it has no overlap with ‘aa[0]” anymore.

F. Wildcard equality operator

For example:

```
$display("(2'b1x ==? 2'b10) is: %b",(2'b1x ==? 2'b10));
$display("(2'b10 ==? 2'b1x) is: %b",(2'b10 ==? 2'b1x));
```

The output is:

```
(2'b1x ==? 2'b10) is: x
(2'b10 ==? 2'b1x) is: 1
```

Why does the simulator give different results?

SystemVerilog LRM section 11.4.6 specifies the behavior of wildcard equality operator as “The wildcard equality operator (==?) and inequality operator (!=?) treat x and z values in a given bit position of their right operand as a wildcard. x and z values in the left operand are not treated as wildcards.”. So in a wildcard equality operator, only the x and z in the right operand are treated as wildcard. If x and z are in the left operand, they are not treated as wildcard. This is the reason the simulator gives different result in above example.

SystemVerilog defines different kinds of equality (inequality) operators. For logical equality (inequality) operators: == and !=, if there is x or z bits in either operand, the relation is ambiguous, then the result shall be a 1-bit unknown value (x). For case equality (inequality) operators: === and !==, x or z in any bits in both operands shall be included in the comparison and shall match for the result to be considered equal, so the result of these operators shall always be a known value, either 1 or 0. For wildcard equality(inequality) operators: ==? and !=?, if x or z bits are in left operand, the result can be x if those bits are not being compared with wildcards in right operand.

G. 'b1 and '1

For example:

```
reg[126:0] x;
initial begin
   x = {127{1'b0}};
   if('b0 === x) $display("'b0 passed");
   else  $display("'b0 failed");
   x = {127{1'b1}};
   if('b1 === x) $display("'b1 passed");
   else  $display("'b1 failed");
   x = {127{1'bx}};
   if('bx === x) $display("'bx passed");
   else  $display("'bx failed");
   x = {127{1'bz}};
   if('bz === x) $display("'bz passed");
   else  $display("'bz failed");
end
```

The simulation result is:

```
'b0 passed
'b1 failed
'bx passed
'bz passed
```

Why does ‘b1 fail but ‘bx/’bz pass?

SystemVerilog LRM section 5.7.1 has specifications on integer literal constants. It says “the numbers specified with the base format shall be treated as signed integers if the s designator is included or as unsigned integers if the base format only is used.”, “The number of bits that make up an unsized number (which is a simple decimal number or a number with a base specifier but no size specification) shall be at least 32. Unsized unsigned literal constants where the high-order bit is unknown (X or x) or three-state (Z or z) shall be extended to the size of the expression containing the literal constant.”. Based on above specification “'b1” is an unsized unsigned integer and its value is 1 which does not equal {127{1’b1}}. For “'bx === {127{1'bx }}”, the simulator needs to extend 'bx to 127 bits (127'bx) so the final result of this case equality operation is “TRUE”.

Here is the LRM example:

```
logic [84:0] e, f, g;
e = 'h5; // yields {82{1'b0},3'b101}
f = 'hx;  // yields {85{1'hx}}
g = 'hz; // yields {85{1'hz}}
```

In the same section, LRM specifies another kind of integer literal: “An unsized single-bit value can be specified by preceding the single-bit value with an apostrophe ('), but without the base specifier. All bits of the unsized value shall be set to the value of the specified bit.”, i.e. '1 to define a value which has all bits ‘1’. In this case, if x is {127{1 'b1}}, ('1 === x) will be “TRUE”.

Here is the LRM example:

```
logic [15:0] a, b, c, d;
a = '0; // sets all 16 bits to 0
b = '1; // sets all 16 bits to 1
c = 'x; // sets all 16 bits to x
d = 'z; // sets all 16 bits to z
```

H. Determining module port kind, data type, and direction

If a module port is declared as “input logic in1”, should “in1” be a variable port or net port? If a module port is declared as “output logic out1”, should “out1” be a variable port or net port?

In SystemVerilog a module port is declared with 4 properties:

- port_direction: such as input, output, inout, ref, etc.
- port_kind: can be any of the net type keywords, or the keyword *var* which specifies that the port is variable.
- data_type: such as logic, int, etc
- port_name

In Verilog and SystemVerilog, a net data and a variable data behave very differently:

- A net can be written by one or more continuous assignments, by primitive outputs, or through module ports. The resultant value of multiple drivers is determined by the resolution function of the net type. A net cannot be procedurally assigned. If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied. A force statement can override the value of a net. When released, the net returns to the resolved value.
- Variables can be written by one or more procedural statements, including procedural continuous assignments. The last write determines the value. Alternatively, variables can be written by one continuous assignment or one port.

So it is very important to understand a port is a net or a variable. SystemVerilog LRM section 23.2.2.3 specifies the rules for determining port kind, data type, and direction. In case port_kind is omitted:

- For input and inout ports, the port shall default to a net of default net type. The default net type can be changed using the `default_nettype compiler directive.
- For output ports, the default port kind depends on how the data type is specified:
  - If the data type is omitted or declared with the implicit_data_type syntax, the port kind shall default to a net of default net type.
  - If the data type is declared with the explicit data_type syntax, the port kind shall default to variable.
- A ref port is always a variable.

Based on these specifications, “input logic in1” declares in1 as a net port and “output logic out1” declares out1 as a variable port.

I. Object constructor function call order

For example:

```
class parent;
  int var1;
  function new(int var1);
    this.var1 = var1;
    $display("parent: var1 is %d",this.var1);
  endfunction
endclass
class child extends parent;
   int myvar1 = 10;
   function new();
      super.new(myvar1);
      $display("child: myvar1 is %d",myvar1);
  endfunction;
endclass
module m1;
  child obj=new;
endmodule
```

The simulation result is:

```
parent: var1 is           0
child: myvar1 is          10
```

Variable var1 in parent class gets valule 0 instead of 10(myvar1).

The result of "this.var1" in parent class depends on the execution order of "super.new(myvar1)" call and the initialization of myvar1 ("int myvar1 = 0") in child class. SystemVerilog LRM section 8.7 has following specification on class constructor method: "The new method of a derived class shall first call its base class constructor [super.new() as described in 8.15]. After the base class constructor call (if any) has completed, each property defined in the class shall be initialized to its explicit default value or its uninitialized value if no default is provided. After the properties are initialized, the remaining code in a user-defined constructor shall be evaluated.". Based on this specification, the execution order in above code is:

1. super.new(myvar1).
2. myvar1 is initialized to 10
3. $display("myvar1 is %d",myvar1);

When super.new(myvar1) is called, myvar1 is not initialized and has the default value of int type variables (it is 0).

J. Accessing array with an invalid index

Another very common question is: when accessing an array with an invalid index, should I expect a runtime error from simulator? SystemVerilog LRM 7.4.6 and 11.5.1 have following specifications: "If an index expression is out of bounds or if any bit in the index expression is x or z, then the index shall be invalid. Reading from an unpacked array of any kind with an invalid index shall return the value specified in Table 7-1. Writing to an array with an invalid index shall perform no operation, with the exceptions of writing to element [$+1] of a queue (described in 7.10.1) and creating a new element of an associative array (described in 7.8.6).". "A part-select that addresses a range of bits that are completely out of the address bounds of the vector, packed array, packed structure, parameter or concatenation, or a part-select that is x or z shall yield the value x when read and shall have no effect on the data stored when written. Part-selects that are partially out of range shall, when read, return x for the bits that are out of range and shall, when written, only affect the bits that are in range." Based on these specifications, Verilog and SystemVerilog users cannot rely on simulator to check invalid index on both packed and unpacked arrays.

III. Conclusion

Some SystemVerilog questions and the related SystemVerilog LRM specifications are discussed in this paper. Understanding these specifications correctly will help SystemVerilog users avoid unexpected simulation results.

References

[1] Stuart Sutherland, Don Mills "Synthesizing SystemVerilog Busting the Myth that SystemVerilog is only for Verification", SNUG Silicon Valley 2013

[2] IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language IEEE Std 1800™-2012 (Revision of IEEE Std 1800-2009)