2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Is The Simulator Behavior Wrong With My SystemVerilog Code

Weihua Han
whan@synopsys.com

SYNOPSYS®
Silicon to Software™

Introduction

- Understanding SystemVerilog specifications in Language Reference Manual correctly and precisely may save a lot of designer's time from debugging the unexpected results.
  - Several questions were raised frequently from our users
- Topics
  - Wildcard package import and export
  - Operator expression short circuiting
  - Random stability
  - Always_comb block inferred sensitivity list
  - Longest static prefixes

Introduction(cont.)

- Topics(cont.)
  - 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

Wildcard Package Import and Export

- Are all the definitions in pkg1 visible/available in module m1?
  - In SystemVerilog 2005, no clear specification on the expected behavior of chained package import
  - In SystemVerilog 2009, package export is specified.

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

Wildcard Package Import and Export

LRM 26.6 Exporting imported names from packages

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.

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.**

Wildcard Package Import and Export

- LRM example

```
package p1;
  int x, y;
endpackage
…
package p3;
  import p1::*;
  import p2::*;
  export p2::*;
  int q = x;
  // p1::x and q are made available from p3. Although p1::y is 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
```

Operator Expression Short Circuiting

- What is the reason that I got different simulation results with the code?
  - obj is really Null, but why does the second code not catch it?

a = obj.b && c
result in "Null Object Access" error

a = c && obj.b
simulation went through this line without any error

Operator Expression Short Circuiting

LRM 11.3.5 Operator expression short circuiting

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.

LRM 11.4.7 Logical operators

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.

Random Stability

- Why does the simulator generate the same value for all of the three instances?
  - I would expect a new different value generated each time $urandom is called.

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

Result:
top.m_inst1  98710838
top.m_inst2  98710838
top.m_inst3  98710838
```

Random Stability

LRM 18.14 Random stability

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()**

…

18.14.1 Random stability properties

"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. "

Random Stability

- Seeding the $urandom call n
  random results
  - Solution 1: using system t
    system time through Syst
    - User will get different ra

```
//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
```

Random Stability

- Seeding the $urandom call manually to generate different random results
  - Solution 2: using a

```
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
```

Always_comb Block Inferred Sensitivity List

- What is the reason that at time 11 the always_comb block seems triggered?
  - Compared with traditional Verilog, SystemVerilog has more kinds of always procedures
    - Always_comb, always_ff, always_latch
  - Always_comb for modeling combinational logical behavior
    - No need to provide the sensitive list

```
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
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
```

Always_comb Block Inferred Sensitivity List

9.2.2.2.1 Implicit always_comb sensitivities
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**

9.2.2.2.2 always_comb compared to always @*
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.

Longest static prefixes

- Is the example 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
```

Longest static prefixes

LRM 9.2.2.2 Combinational logic always_comb procedure

The variables written on the left-hand side of assignments shall not be written to by any other process. 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. See 6.5 for more details.

11.5.3 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.**

Longest static prefixes

- Solution

```
LRM 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
```

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

Wildcard Equality Operator

```
$display("(2'b1x ==? 2'b10) is: %b",(2'b1x ==? 2'b10));
$display("(2'b10 ==? 2'b1x) is: %b",(2'b10 ==? 2'b1x));
Result:
(2'b1x ==? 2'b10) is: x
(2'b10 ==? 2'b1x) is: 1
```

- What is the reason the simulator gives the different results?
- SystemVerilog has different kinds of equality and inequality operators
  - Logical equality and inequality: ==, !=
  - Case equality and inequality: ===, !==
  - Wildcard equality and inequality: ==?, !=?

Wildcard Equality Operator

11.4.6 Wildcard equality operators

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.**

The different types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (X or Z). The == and != operators may result in x if any of their operands contains an X or Z. The === and !== operators explicitly check for 4-state values; therefore, X and Z values shall either match or mismatch, never resulting in X. **The ==? and !=? operators may result in X if the left operand contains an x or Z that is not being compared with a wildcard in the right operand.**

'b1 and '1

- Why does 'b1 fail but 'bx/'bz pass?

```
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

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

'b1 and '1

LRM examples:

```
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}}
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

'1 === x  is TRUE when x is {127{1 'b1}}
```

5.7.1 Integer literal constants

the numbers specified with the base format shall be t
designator is included or as unsigned integers if the b
number of bits that make up an unsized number (wh
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.**

“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.

Determining Module Port Kind, Data Type, and Direction

- Questions:
  - What do “input **wire** logic aa” and “input **var** logic bb” mean?
  - With “input logic in1”, is “in1” a variable port or net port? With “output logic out1”, is “out1” a variable port or net port?

- SystemVerilog net vs variable

```
//is this code valid at all?
module m1(input logic aa);
    assign aa = 1'b1;
endmodule
```

Determining Module Port Kind, Data Type, and Direction

LRM 6.5 Nets and variables

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.

Determining Module Port Kind, Data Type, and Direction

- 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(like **wire**), or the keyword ***var*** which specifies that the port is variable.
  - data_type: such as logic, int, etc
  - port_name

Determining Module Port Kind, Data Type, and Direction

23.2.2.3 Rules for determining port kind, data type, and direction

If the 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 (see 22.8).

— 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.

Object Constructor Function Call Order

- Variable var1 in parent class gets valule 0 instead of 10(myvar1). Is the simulator wrong?

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

Object Constructor Function Call Order

LRM 8.7 Constructors

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.

Accessing Array with An Invalid Index

- By accident when I tried to write an array element, the index has one bit of x. Should the simulator error out in this case?

LRM 7.4.6 Indexing and slicing of arrays
LRM 11.5.1 Vector bit-select and part-select addressing
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).

Conclusion and References

- Understanding the LRM specification correctly and precisely is important
  - Write the first code correctly to get the expected result
  - May save a lot of debugging time
- References
  - Stuart Sutherland, Don Mills “Synthesizing SystemVerilog Busting the Myth that SystemVerilog is only for Verification”, SNUG Silicon Valley 2013
  - IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language IEEE Std 1800™-2012 (Revision of IEEE Std 1800-2009)