Being Assertive With Your X (SystemVerilog Assertions for Dummies)

Don Mills

LCDM Engineering (Logic, Coding, & Design Methodology)

mills@lcdm-eng.com

ABSTRACT

This paper will show how to use SystemVerilog Assertions to monitor for X conditions when using synthesizable Verilog or SystemVerilog code. The paper begins by showing techniques used with Verilog today, and then shows how SystemVerilog will implement the same assertion with its built-in assertion statement. The intended audience is the design engineer with little or no experience with assertions.

Table of Contents

1.0	What are Assertions? Why Use Assertions for 'X' Checking?	4
	The Origin of 'X'	
	2.1 Uninitialized Variable	
	Code Example 2.1	
	2.2 Data Path	
	Code Example 2.2.	
	2.3 Multiple Drivers on a Wire	
	Code Example 2.3a	
	Code Example 2.3b	
	2.4 Direct 'X' Assignments	7
	Code Example 2.4a	
	Code Example 2.4b	
	2.5 Undriven Wires and Floating Module Input Ports	8
	Code Example 2.5	
	2.6 Gate Level 'X's	
3.0	The Big Lie	9
	3.1 If/Else Statements	
	Code Example 3.1a	9
	Code Example 3.1b	10
	3.2 Case/Casez/Casex Statements	
	Code Example 3.2a	10
	Code Example 3.2b	11
	Code Example 3.2c	12
	3.3 AND/NAND'ing with a Logic Low Level	122
	Code Example 3.3	12
	Table 3.3	12
	3.4 OR/NOR'ing with a Logic High Level	133
	Code Example 3.4	133
	Table 3.4	133
4.0	The Verilog Solution	133
	4.1 IF 1 ELSE IF 0 ELSE	13
	Code Example 4.1	14
	4.2 CASE default	14
	Code Example 4.2	15
	4.3 Module Port Monitoring	155
	Code Example 4.3	155
	4.4 Instantiation of Assert Modules	155
	Code Example 4.4	
	4.5 The VHDL Solution	166
5.0	PSL, OVL and PLI	16
	SystemVerilog Assertions	
-	6.1 Two Types of SystemVerilog Assertions	

Table of Contents cont'd

6.2 Immediate Assertions for SystemVerilog Code	
6.2.1 Immediate Assertions for SystemVerilog If/else	188
Code Example 6.2.1a	188
Code Example 6.2.1b	19
6.2.2 Immediate Assertions for SystemVerilog Case Statements	199
Code Example 6.2.2	199
6.2.3 Immediate Assertions for SystemVerilog Logic Gating	19
Code Example 6.2.3	20
6.3 Immediate Assertions for SystemVerilog Ports	2020
Code Example 6.3	
6.3.1 Gate Level 'X' Checking	211
6.3.3 External Immediate Assertions for Verilog Ports	211
Code Example 6.3.3	22
6.4 SystemVerilog System Functions and Tasks for Assertions	
6.4.1 \$assertoff and \$asserton	
Code Example 6.4.1	23
6.4.2 \$isunknown	233
6.4.3 \$onehot and \$onehot0	233
Code Example 6.4.3	24
7.0 Conclusions and Recommendations	24
7.1 Interface/Difference between SystemVerilog Assertions and	PSL/Sugar Assertions255
7.2 Immediate Assertions for Verilog Code	_
8.0 Acknowledgements	255
9.0 References	
10.0 About the Author	
IV.V ANUUL IIIC AUIIIVI	43

1.0 What are Assertions? Why Use Assertions for 'X' Checking?

Assertions are a mechanism or tool used by HDL's (VHDL and Verilog) to detect a design's expected or intended behavior. There are two types of assertions (both implemented by SystemVerilog): immediate assertions and concurrent assertions. Immediate assertions check the state of a condition at a given point in time. Concurrent assertions provide a means whereby a condition can be checked over time. The focus of this paper is to show how to use immediate assertions for 'X' detection during RTL and GATE level simulation. Assertions can provide a "bed of nails" type checking for 'X's. Using assertions to check for 'X's is like inverse assertions--using assertions for finding unintended behavior. The premise is that once a design is reset to a known state, it should remain in a known state throughout simulation.

This paper begins by discussing how an 'X' state can enter a simulation environment after the design had been brought to a known state. Once an 'X' state is in a design, there are many RTL coding styles that will hide the 'X', thus possibly preventing it from becoming visible during simulation. Subsequent sections will show the Verilog coding styles that hide 'X's, and the painful Verilog methods used to detect these conditions. There were many external methods developed to apply assertions to a design which are better (easier) solutions than using Verilog. These new solutions are briefly discussed. These new external solutions became the building blocks that were incorporated into SystemVerilog. Finally, a detailed discussion of SystemVerilog Assertions will be presented. The focus will be on how to use the built-in assertions of SystemVerilog as a simple, yet complete method of ensuring that a design remains void of 'X's once the design is brought to a known state.

2.0 The Origin of 'X'

One of the most devastating situations that can occur within a design is to have an undetected erroneous condition in the design. The existence of 'X' states in a design after the design has been initialized can be the source of much anguish. The paper "The Dangers of Living with an 'X', (Bugs Hidden in Your Verilog)", presented by Mike Turpin at Boston SNUG 2003, won Best Technical Paper [1]. It focused on the issues of simulation, synthesis, and formal verification issues when a design has 'X's. The paper describes a number of guidelines on "How to Avoid Dangerous 'X's". The good 'X's described in the paper are the ones that do not propagate and might provide some synthesis logic reduction capabilities.

So where, and how, do 'X's get into a design? The remainder of this section details several common practices which set up an environment where 'X's could occur.

2.1 Uninitialized Variable

The default value of a synthesizable variable data type in Verilog is 'X'. Just to review, the variable data types are the data types of any signals assigned inside a procedure block (always or initial). These data types are reg, integer, real, time for Verilog. The synthesizable variable data types are reg and integer. SystemVerilog has added new 4- and 2-state variable data types. The new 4-state variable data type is logic. This new data type is the default data type for SystemVerilog. Thus any declared variable of type reg, integer, or logic, that is read, but not assigned a value, will have 'X' as its value. This is generally considered a design error. There is at least one exception, as noted in the next section.

Code Example 2.1

2.2 Data Path

Sometimes a design will not apply a reset to a long change of shift register flip-flops. The flip-flops, other than the first flop, are often called follower flip-flops. The initialization of these flops comes from the first stage being reset or loading known data. One method of initializing all the flip-flops during reset is to hold the first flip-flop in reset, then apply enough clocks for the follower flops to become reset. Often though, follower flops are used in data path design styles, which are self-initializing based on the data input to the design. If the design is not correctly managed, 'X's can enter the design and lock it up.

```
module shift_reg (clk, rst_N, data_in, sr_out);
  input clk, rst_N, data_in;
  output sr_out;
 parameter sr_size = 5;
                          // variable declaration
  reg sr0 out, i;
  reg sr_array [1:sr_size];
  always@(posedge clk, negedge rst N)
    if (!rst N)
      sr0 out <= 1'b1;</pre>
    else
      sr0_out <= data_in;</pre>
  always@(posedge clk) begin
    sr_array[1] <= sr0_out;</pre>
    // with system verilog array assignments, the following
    // for loop will not be needed.
    for (i = 2; i <= sr_size; i=i+1)
      sr_array[i] <= sr_array[i - 1];</pre>
  end
  assign sr_out = sr_array[sr_size];
endmodule
```

Code Example 2.2

2.3 Multiple Drivers on a Wire

A correct design should only have one source driving each signal, unless the signal is driven by tri-state drivers. If the signal is a variable data type (primarily **reg** or **integer**) and it has multiple concurrent sources, then the last assignment made during the simulation cycle will be the final result. This is considered a Verilog race condition, in that the final result is indeterminate. Synthesis will give a warning in this situation indicating an unknown wire-or'ed logic and then will make an attempt to build something.

When the signal with multiple sources is a net data type of type **wire**, the built-in Verilog wire resolution tables will be used to determine the final value of the signal. When coded (designed) correctly, the only situation where a signal of type **wire** would have multiple drivers would be when the drivers are tri-state drivers. Any other types of multiple driven wire signals would likely result in 'X's during simulation. However, this is a situation where during simulation, if both drivers always drove the same values, the erroneous conditions would never show up, but could very likely become a design flaw once the chip is produced.

The code examples shown here are two contrived code examples showing in simple terms common mistakes where a wire would have multiple non-tri-state sources. The first example has the same output driven by two continuous assign statements.

```
module multidrive_wire(in1, in2, out);
  input in1, in2;
  output out;

assign out = in1;
  assign out = in2;
endmodule
```

Code Example 2.3a

This second example has a common signal driven by the output of multiple instantiations.

```
module mod1(in1, out1);
  input in1;
  output out1;

  assign out = in1;
endmodule

module mod2(in2, out2);
  input in2;
  output out2;

assign out = in2;
endmodule

module top(in1, in2, out);
  input in1, in2;
  output out;
```

```
mod1 m1 (.in1, .out1(out));
mod2 m2 (.in2, .out2(out));
endmodule

module top_test;
reg in1, in2;

top t1 (.in1, .in2, .out1(out));

initial begin
   in1 = 0; in2 = 0; // out goes to 0
   #10 in1 = 1; // out goes to X
   #10 in2 = 1; // out goes to 1
end
endmodule
```

Code Example 2.3b

2.4 Direct 'X' Assignments

At first glance, one might wonder why any design intended to be implemented would have direct 'X' assignments. In Turpin's paper, direct assignments were strongly recommended when using **case** statements, so as to prevent the **case** statement from blocking or hiding an 'X' in the case select. In the example below, if the **sel** is an 'X', then the corresponding output would hold its previous value until a known condition is available on **sel**.

```
module mod_case_24a(sel, a, b, c, d, out);
  input [1:0] sel;
  input a, b, c, d;
  output out;

reg out;

always_comb //similar to always@* but better
  case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b01 : out = c;
    2'b01 : out = d;
    endcase
endmodule
```

Code Example 2.4a

Turpin's recommendation is to completely cover all the binary decodes of the case select, and still include a default to pass through an 'X' condition if the case select goes to 'X'. The synthesis result of Code Example 2.4a and Code Example 2.4b will be the same.

```
module mod_case_24b(sel, a, b, c, d, out);
input [1:0] sel;
input a, b, c, d;
```

```
output out;

reg out;

always_comb //similar to always@* but better
  case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b01 : out = c;
    2'b01 : out = c;
    endcase
endmodule
```

Code Example 2.4b

2.5 Undriven Wires and Floating Module Input Ports

Undriven wires, whether by declaration or as a floating input port, will default to a 'Z' state value. When these undriven signals are then used as inputs to logic (gates or flip-flops), the resulting output will go to 'X'. A good lint tool will generally catch these situations.

```
module floating_wires(a, b, c, out);
  input a, b, c;
  output out;

wire a, b, c, d;

assign out = (a && c) || (b && d);
endmodule

module floating_wires_top(a, b, c, d, out);
  input a, b,c, d;
  output out;

floating_wires fw (.b(b), .c(c), .out(out));
endmodule
```

Code Example 2.5

2.6 Gate Level 'X's

At the gate level, 'X's are generated by violating a component's timing requirements. Additionally, 'X's will be output from User Defined Primitives (UDP's) when a given input condition is not declared in the UDP's lookup table. There are also some crossover conditions from the RTL 'X' generation, such as multiple drivers on the same net, but hopefully those conditions would be resolved before the design went through synthesis and on to gates.

3.0 The Big Lie

If 'X's are not tested or looked for, then they are not really there, right? One of the biggest concerns of any designer is error conditions which go undetected. The frightening part of RTL 'X's in a design is that they can be masked so that they don't trickle to a primary output. This concern was one of the major justifications for the guidelines presented by Turpin in his paper at Boston SNUG 03. A number of examples of how RTL code can mask 'X's are presented in this section.

3.1 If/Else Statements

A basic **if/else** can mask an 'X' value in the **if** condition. If the **if** condition is true, then the **if** statements are executed. For all other conditions, the **else** (if there is one) statements will be executed. Thus, an **if** condition that has an 'X' in it is not considered true, and will cause the **else** branch to be executed.

```
module if_else_31a (ifcond, a, b, if_out);
  input ifcond, a, b;
  output if_out;

reg if_out;

always_comb
  if (ifcond)
    if_out = a;
  else
    if_out = b;

endmodule
```

Code Example 3.1a

The following code will test for true first, then false, and will then pass through an 'X' if the **if** condition is neither true nor false. This coding style could be used widely, but generally is not.

```
module if_else_31b (ifcond, a, b, if_out);
  input ifcond, a, b;
  output if_out;

reg if_out;
```

```
always_comb
  if (ifcond)
    if_out = a;
  else if (!ifcond)
    if_out = b;
  else
    if_out = 'x;
endmodule
```

Code Example 3.1b

3.2 Case/Casez/Casex Statements

For basic **case** statements, if the case select does not match one of the case items, the output will retain the previous value. In example 3.2a, the **sel** is fully decoded from a binary perspective, but if the **sel** goes to 'X' (or 'Z'), the unknown output will be masked and the output will hold its previous value.

```
module mod_case_32a(sel, a, b, c, d, out);
  input [1:0] sel;
  input a, b, c, d;
  output out;

reg out;

always_comb //similar to always@* but better
  case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b10 : out = c;
    2'b11 : out = d;
  endcase
endmodule
```

Code Example 3.2a

A **casez** treats 'Z's in the case item or case selects as "don't cares". Turpin's paper recommended against using **casez**'s altogether. However, for certain situations, usage is desirable. If you need the "don't care" capabilities of Verilog **casez/casex**, the previously recommended approach was to use **casez** over **casex** [2]. Now, using either **casez** or **casex** in conjunction with the SystemVerilog assertions discussed in Section 6.0 eliminates all former design concerns regarding 'X's or 'Z's in the case selects.

In Example 3.2b, when the case select has a 'Z' in either the **lsb** or the **msb** or both, the bit position(s) become(s) a "don't care". With "don't cares", the case statement could have multiple matches, but the first matching case item will be selected. If the case select has an 'X', the result would be the same as example 3.2a., i.e., the 'X' will be masked and the output will hold its previous value. For practical purposes, the **casez** should only be used when the case select is known and does not have 'X' or 'Z' as a value. The **casez** "don't care" is a great language

feature for coding and should not be restricted, but rather it should be used with assertions to ensure that the usage meets the required guidelines for design continuity.

```
module mod_casez_32b(sel, a, b, c, d, out);
  input [1:0] sel;
  input a, b, c, d;
  output out;

reg out;

always_comb //similar to always@* but better
  casez (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b10 : out = c;
    2'b11 : out = d;
  endcase
endmodule
```

Code Example 3.2b

One side note about **casez**. This author was teaching a class on Verilog for a company and was asked to emphasize the company's coding guidelines as part of the course. Their recommendations for **casez/casex** were to never use the **casez** "because we don't case on tristates". The 'Z' in a **casez** is a "don't care", not a tri-state. The coding guidelines from this company were restrictive based on incorrect understanding of the language. Make sure that the coding guidelines you are using are up-to-date and are based on correct language implementations.

A **casex** will function in much the same way as a **casez**, but with 'X' or 'Z' treated as a "don't care" when either appears in the case items or the case select.

```
module mod_casez_32c(sel, a, b, c, d, out);
  input [1:0] sel;
  input a, b, c, d;
  output out;

reg out;
```

```
always@* //always_comb //similar to always@* but better
  casex (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b10 : out = c;
    2'b11 : out = d;
  endcase
endmodule
```

Code Example 3.2c

The paper by Mills and Cummings [2] recommended not using **casex** because it did not provide any additional functionality over the **casez**, and because 'X's were more likely to appear in a design than 'Z's. With the use of assertions to check for 'X's (and 'Z's for that matter), this recommendation could be rescinded. However, this author's opinion is that since **casex** still does not provide any additional features over **casez**, and that **casez** should still be the preferred coding construct when a set of case items is used with "don't cares" as part of its values.

3.3 AND/NAND'ing with a Logic Low Level

An **and** gate will mask a 'Z' or 'X' when **and**'ed with a low.

```
module and_mask(in1, in2, out);
  input in1, in2;
  output out;
  assign out = in1 & in2;
endmodule
```

Code Example 3.3

in1 0 0 1	in2 0 1 0	out 0 0 0
1	1	1
0	z x	0
1	Z	x
1	X	x

Table 3.3

3.4 OR/NOR'ing with a Logic High Level.

An **or** gate will mask a 'Z' or 'X' when **or**'ed with a high.

```
module or_mask(in1, in2, out);
  input in1, in2;
  output out;
  assign out = in1 | in2;
endmodule
                        Code Example 3.4
in1
     in2
            out.
0
      0
             0
0
      1
             1
      0
1
             1
1
      1
             1
0
      Z
             X
0
      Х
             Х
1
             1
      7.
```

1

Table 3.4

4.0 The Verilog Solution

Х

The procedure is to check for the 'X' condition, print an error message upon finding an 'X' condition, using pragma pairs to hide the condition checking and the print statement from synthesis. Synopsys ignores **\$displays** but does issue a warning when it encounters a **\$display**. The use of pragmas eliminates these synthesis warnings. This approach is rarely used, possibly because of the additional synthesis warnings combined with the extra typing and condition checking. A `ifdef could be included to add the flexibility of turning the manual assertions off as desired.

4.1 IF 1 ELSE IF 0 ELSE

Synthesis will assume the **if** condition is either a high or a low, and therefore will only need a single **if** check. For simulation, the first **if** will test the **if** condition for the high state, then a second is used to check for the **if** condition at the low state. A final **else** is added for simulation to print an assert statement, indicating that the **if** condition was neither high nor low, and to output an 'X' for the **if** statement.

```
module if_else_41 (ifcond, a, b, if_out);
  input ifcond, a, b;
  output if_out;

reg if_out;  // variable declaration
  always_comb
  if (ifcond)
```

```
if out = a;
   else
      //pragma translate off
      if (!ifcond)
      //pragma translate on
      if_out = b;
    //pragma translate off
    else
      // could test ... if (^ifcond == 1'bx)
      // but that would be redundant
     begin
      if_out = 'x;
      $display("assert: ifcond in mod ifcond is X
                at time %d", $time );
    //pragma translate on
endmodule
```

Code Example 4.1

4.2 CASE default

If all the case conditions are covered, synthesis will treat the case as a full_case. By assigning 'out' to 'X' in the case default, the case statement will pass an 'X' on if the **sel** is either an 'X' or a 'Z'. There are many variations of case statements covered in other papers [2] which would use different derivatives of the basic assert shown below.

```
module mod_case_42(sel, a, b, c, d, out);
  input a, b, c, d;
  input [1:0] sel;
  output out;
  reg out;
  always_comb
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
      //pragma translate off
      default :
                  begin
        out = 'x;
        $display("assert: ifcond in mod ifcond is X
                  at time %d", $time );
        end
      //prama translate on
```

endcase endmodule

Code Example 4.2

4.3 Module Port Monitoring

Sections 4.1 and 4.2 show simple Verilog assertions for **if** and **case** statements used within procedural blocks. To check for combinational logic 'X' conditions, the input and output ports to the module should be monitored. To be completely thorough, monitoring all inputs to all levels of combinational logic would be required.

Code Example 4.3

4.4 Instantiation of Assert Modules

The approach shown in sections 4.1 through 4.3 work with RTL code, but once the design is synthesized to gates, the assertions are lost. Only the approach shown in section 4.3 can be easily modified so that it will work with both an RTL- and a GATE-level design. The assertion is instantiated in the test bench and then uses hierarchical referencing to the ports of the instantiated modules in the design. The assertion will be in the test bench and will be available for both RTL and GATE testing.

```
module and_mask(in1, in2, and_out);
  input in1, in2;
  output and_out;
  assign and_out = in1 & in2;
endmodule

module or_mask(in1, in2, or_out);
  input in1, in2;
  output or_out;
  assign or_out = in1 | in2;
endmodule
```

```
module top(in1, in2, and_out, or_out);
  input in1, in2;
  output and_out, or_out;
  and_mask m1 (.in1, .in2, .and_out);
  or mask m2 (.in2, .in2, .or_out);
endmodule
module top_test;
  reg in1, in2;
  top t1 (.in1, .in2, .and_out, .or_out);
  initial begin
    in1 = 0; in2 = 0; // out goes to 0
    #10 in1 = 1;  // out goes to X
#10 in2 = 1;  // out goes to 1
  end
  //verilog assert statements
  always@(t1.ml.in1, t1.ml.in2, t1.ml.out)
    if (^{t1.m1.in1, t1.m1.in2, t1.m1.and_out} == 1'bX)
      $display("assert: X in and_mask ports
                   at time %d", $time );
  always@(t1.m2.in1, t1.m2.in2, t1.m2.out)
    if (^{t1.m2.in1, t1.m2.in2, t1.m2.or_out} == 1'bX)
      $display("assert: X in or_mask ports
                   at time %d", $time );
endmodule
```

Code Example 4.4

4.5 The VHDL Solution

VHDL was written with assertions as part of the language. Many of the approaches shown above can be done with VHDL assertions. One issue with VHDL assertions at the test bench level is that the assertions cannot see internal signals of the design unless the signals are declared as global. There are many new features added to SystemVerilog assertions that go way beyond the capabilities of VHDL assertions. Only a few of the SystemVerilog immediate assertions will be discussed in this paper. Discussion of other features of immediate assertions and also dynamic assertions will be left for future papers.

5.0 PSL, OVL and PLI

Due to the lack of formal assertions in the Verilog language and the need to provide capabilities beyond simple print statement assertions, a number of assertion extensions to Verilog have been developed. Third party tools such as Vera and "that other verification language" interface to simulations through the PLI to monitor and apply assertions to a design. The PLI can be used to

manually generate assertions, however, it is recommended that manual PLI assertions be limited to only a few per design because the of the complexity of managing these PLI assertions[3].

Accellera has and is working on a number of projects to standardize assertion syntax. Three of these projects are OVL, PSL and SystemVerilog. Open Verification Library (OVL) is intended to be a set of Verilog and VHDL assertions that can be placed in HDL code to monitor specific properties of a design. The Property Specification Language (PSL) is an assertion property language based on the IBM Sugar property language. Where OVL is primarily intended to be used during the RTL coding phase, PSL is designed to focus on the system level of a design. The mix of these two approaches is applied and incorporated into SystemVerilog. Many of the Accellera committee members working on OVL and PSL joined the Accellera SystemVerilog committees to define the test capabilities of SystemVerilog and to maintain the standardization they had been working to achieve.

6.0 SystemVerilog Assertions

Some of the restrictions noted in Turpin's paper are a result of the lack of good 'X' assertion checkers. Turpin's paper also recommended two new assertion type checkers which had been proposed to Accellera for addition to PSL. As noted in Section 5.0, SystemVerilog's assertions are based on PSL, and to a large degree are a subset of PSL. The advantage of SystemVerilog's assertions is that they are built into the language, and therefore run from the kernel rather than through a PLI or FLI. In other words, they will not slow down the simulation as much as third party tools will, and they will be easier to use because they are part of the language.

6.1 Two Types of SystemVerilog Assertions

SystemVerilog has two types of assertions: immediate assertions and dynamic assertions. Immediate assertions are statements in procedure blocks, tasks, and functions. They evaluate at the time the statement is executed, just like a blocking statement. A dynamic assertion is used to monitor conditions over time. The focus of this section, and the overall purpose of this paper, is to show the use of immediate assertions in a very simple manner to monitor and to ensure that you have no 'X's in your design, without giving up any coding flexibility.

6.2 Immediate Assertions for SystemVerilog Code

Immediate assertions provide a simple solution to monitoring for 'X's in a design. Synopsys developers have stated that when assertions are implemented, synthesis will ignore the assert statements. This means that pragmas should no longer be required to prevent error or warning statements when SystemVerilog code is read for synthesis.

The syntax for an immediate assertion is:

assert (expression) pass_statement [else fail_statement]

An immediate assert statement follows similar syntax to that of an **if** statement. The **else** clause is optional. The statements in the pass or fail section can be any procedural statements from print messages to counters.

SystemVerilog has built-in defined failure behavior:

```
$fatal - ends the simulation
$error - gives a runtime error, but simulation continues
$warning - gives a runtime warning, simulation continues
$info - prints the specified message

initial begin
    assert (address == `start_adr);
    else $fatal("bogas start address");
```

6.2.1 Immediate Assertions for SystemVerilog If/else

Adding an assert statement just before the **if** statement allows for testing the **if** conditions outside the **if** statement. This leaves the **if** statement clean, yet the integrity of the condition checking is maintained.

Code Example 6.2.1a

A second and perhaps less-preferred style would be to apply the **if** statement inside the assert statement. This approach might not be acceptable for synthesis (only time will tell).

```
if_out = b;
else $error("ifcond = X");
endmodule
```

6.2.2 Immediate Assertions for SystemVerilog Case Statements

Using an assert statement to monitor the case select signal simplifies the **case** statement. When the **case** statement covers all the binary conditions, the default is no longer needed since it was used to pass the 'X' state through the **case** statement. You can still use the 'X' assignment as a "don't care" for synthesis and other coding styles. The assert will ensure that the **case** select never goes to 'X', making **casex** and **casez** safe constructs.

Code Example 6.2.1b

```
module mod_case_622(sel, a, b, c, d, out);
  input a, b, c, d;
  input [1:0] sel;
  output out;
  reg out;
  always_comb begin
    assert (^sel !== 1'bx);
      else $error("case_Sel = X");
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
    endcase
  end
endmodule
```

Code Example 6.2.2

6.2.3 Immediate Assertions for SystemVerilog Logic Gating

Verilog can model combinational logic through a number of methods. These include continuous assign statements, instantiation of gate primitives, and combinational always blocks. As shown above, **and** and **or** gates can mask 'X's, thus, the inputs to combinational logic must be monitored for 'X' conditions.

```
module and_or_623a(in1, in2, and_out, or_out);
  input in1, in2;
  output and_out, or_out;

assign or_out = in1 | in2;
  assign and_out = in1 & in2;
```

```
always_comb begin
  assert (^{in1, in2} !== 1'bx);
    else $error("logic inputs = X");
endmodule
```

Code Example 6.2.3

6.3 Immediate Assertions for SystemVerilog Ports

It might be argued that if ports of a module were monitored for 'X's, then other 'X' checking would be redundant. Checking only the ports will miss the signals that are undriven within the module and thus float to logic 'X' or 'Z' depending on their data type. Additionally, as noted earlier in this paper, signals that are outputs of UDPs or driven directly to 'X', are other sources of locally generated 'X's within a module. So if the combinational logic and **if** and **case** statements are all being monitored by assert statements, why monitor the ports also? This author proposes that monitoring for 'X's on the port signals for RTL combinational code is redundant. However, checking ports and internal combinational inputs for 'X's will monitor for flip-flop outputs that go to 'X'. Perhaps a better method would be to monitor ports and flip-flop outputs for 'X'.

```
module shift_reg (clk, rst_N, data_in, sr_out);
  input clk, rst_N, data_in;
  output sr_out;
 parameter sr_size = 5;
                         // variable declaration
  reg sr0 out, i;
  reg sr_array [1:sr_size];
  always@(posedge clk, negedge rst_N)
    if (!rst N)
      sr0_out <= 1'b1;</pre>
    else
      sr0_out <= data_in;</pre>
  always@(posedge clk) begin
    sr_array[1] <= sr0_out;</pre>
    // with system verilog array assignments, the following
    // for loop will not be needed.
    for (i = 2; i <= sr_size; i=i+1)
      sr array[i] <= sr array[i - 1];</pre>
  end
  assign sr_out = sr_array[sr_size];
```

```
always_comb begin
   assert (^{clk, rst_N, data_in, sr_out} !== 1'bx);
   else $error("FF have X problems");
endmodule
```

Code Example 6.3

Ports and flip-flop output signal names are the only points that are constant between an RTL design and the synthesized GATE design. 'X' assert statements created for these points during RTL can be reused during the GATE verification as well. The only stipulation is that these assertions be defined external to the design so that they are preserved during synthesis. Section 6.3.2 will show how SystemVerilog has hooks built in to make this process less painful.

6.3.1 Gate Level 'X' Checking.

When considering GATE design 'X' checking, the task can seem overwhelming. 'X's can be generated by bad UDPs used in the gate level or by good UDPs where the inputs violate some condition. Additionally, timing violations will be the most significant source of 'X's at the gate level. If the gate logic is modeled as optimistic, **and** and **or** gates could mask out the 'X'. To be absolutely complete at the gate level, every net would need to be monitored for 'X's. This, however, is very extreme and impractical. Monitoring the module ports and other critical points as determined by the designer would be a reasonable medium.

6.3.3 External Immediate Assertions for Verilog Ports

Monitoring module ports and flip-flop outputs at the test bench level will make these assertions usable with both RTL and GATE level designs.

```
module and_mask_633(in1, in2, and_out);
  input in1, in2;
  output and_out;
  assign and out = in1 & in2;
endmodule
module or_mask_633(in1, in2, or_out);
  input in1, in2;
  output or_out;
  assign or_out = in1 | in2;
endmodule
module top 633(in1, in2, and out, or out);
  input in1, in2;
  output and_out, or_out;
  and_mask_633 m1 (.in1, .in2, .and_out);
  or mask 633 m2 (.in2, .in2, .or out);
endmodule
module top_test_633;
  reg in1, in2;
```

Code Example 6.3.3

6.4 SystemVerilog System Functions and Tasks for Assertions

SystemVerilog has a number of system functions that simplify and augment assertions.

6.4.1 \$assertoff and \$asserton

The most useful system tasks are the **\$assertoff** and **\$asserton**. One of the biggest issues regarding the Verilog print statement style of assertions is that they are either always on, or through **`defines**, set to be always off. They could not be turned on or off dynamically during a simulation. When set to on, they output large numbers of false positives during reset, and other times in the simulation where assertion results would be a "don't care". By using **\$assertoff**, the assertions specified will be turned off until a **\$asserton** is executed, thus eliminating false positives during "don't care" periods in the simulation.

```
sr array[1] <= sr0 out;</pre>
    // with systemverilog array assignments, the following
    // for loop will not be needed.
    for (i = 2; i <= sr_size; i=i+1)
      sr_array[i] <= sr_array[i - 1];</pre>
  end
  assign sr out = sr array[sr size];
always_comb begin
    assert (^{clk, rst_N, data_in, sr_out} !== 1'bx);
      else $error("FF have X problems");
endmodule
module test;
  shift_reg sr(*)
  initial begin
    $assertoff(sr)
     ... do reset stuff
     $asserton(sr)
      . . .
                      Code Example 6.4.1
```

The arguments for **\$assertoff** and **\$asserton** are a list of assertions, a list of modules, or a specified number of hierarchical levels down the hierarchy.

6.4.2 \$isunknown

The system function **\$isunknown** is shorthand for checking if a condition is unknown. The following two assertions are equivalent:

```
assert (^{clk, rst_N, data_in, sr_out} !== 1'bx);
assert !($isunknown({clk, rst_N, data_in, sr_out});
```

6.4.3 \$onehot and \$onehot0

SystemVerilog provides two additional system functions that are not really part of a paper on 'X' detection, but are very much worth noting. The **\$onehot** system function will return a one if only one bit in the argument is set to a one. The **\$onehot**0 system function will return a one if only one bit in the argument is set to a zero. Obviously these functions will be used to monitor state vectors in **onehot** state machines.

```
module onehot_sm (clk, rst_N, data_in, sm_out);
  input clk, rst_N, data_in;
  output sm_out;

parameter state1 = 3'b001,
```

```
state2 = 3'b010,
            state3 = 3'b100;
                   // variable declaration
  reg sm_out;
  reg [2:0] state, next;
  always ff@(posedge clk, negedge rst N)
    if (!rst N)
     state <= state1;</pre>
    else
      state <= next;
  always comb begin
   next = 'bx;
   case(state)
     state1 :
                           next = state2;
      state2 : if (data_in) next = state3;
              else next = state2;
      state3 :
                           next = state1;
    endcase
  end
  assign sr_out = (state == state3);
  alway comb
   assert $onehot(state);
     else $error("onehot state value failure");
endmodule
                     Code Example 6.4.3
```

The next state logic is initialized to 'X' to cover all the unused states as "don't cares" for synthesis. Turpin suggested that direct assignments to a known state for unused states might provide a better synthesis result. Choose your favorite approach. In Example 6.4.3, the **next** assigned to 'X' gives two advantages. First, any missing state transitions will force the next state to 'X' thus locking the state machine into an 'X' state. Second, when all the state transitions are covered, pre-assigning **next** to 'X' will have no effect on the RTL code, but will be interpreted by the synthesis tool the same as **full_case**, thus making all unused states as "don't cares" for logic reduction. This coding style is monitored by the assert statement so that a missing state transition and state going to 'X' is immediately identified.

7.0 Conclusions and Recommendations

There are a number of conditions in which 'X's can be introduced to a simulation. In the early days of digital design, designers hoped that simulations and wave form monitoring were sufficient to find broken portions of a design, such as 'X's in the design. Now that design complexity has reached exponential proportions, automatic monitoring and self-checking designs are the only solutions that can be applied in order to have confidence of success. Using built-in assertions to monitor for 'X' conditions will provide the foundation for a more stable design to

then verify functionality. The focus of the verification can then be on design verification instead of split between verification and design integrity.

7.1 Interface/Difference between SystemVerilog Assertions and PSL/Sugar Assertions One of the primary advantages of SystemVerilog is that it is designed and based on working technology. SystemVerilog has pulled together Verilog, SuperLog, and even some aspects of VHDL to use as a basis for modeling functionality. Vera, Sugar, PSL, and OVL, as well as SystemC, were used as a basis for the verification parts of SystemVerilog. This makes SystemVerilog superior to other approaches because the depth of verification capabilities built into the kernel of the simulator. Other tools provide similar capabilities, but they must interface to Verilog or VHDL through the tools PLI or FLI.

7.2 Immediate Assertions for Verilog Code

Immediate assertions are the ideal method for monitoring for 'X's in a design. They do not require the overhead that the Verilog method requires. Because they are built into the language, they will not impact simulation performance the way third party verification languages would. Perhaps the most critical asset of SystemVerilog assertions is the capabilities to turn on and off the assert checking thus eliminating the numerous false positives that other methods generate.

8.0 Acknowledgements

I want to acknowledge Cliff Cummings, Steve Golson, Stu Sutherland, and Tim Wilson for their advice in reviewing ideas and concepts for this paper. Most importantly, I acknowledge my sweetheart who is also my technical writing reviewer and the one who actually makes the paper readable. Finally I acknowledge Joanne Wegener for having serious patience with me as I have completed this paper.

9.0 References

- [1] Mike Turpin, "The Dangers of Living with an 'X', (Bugs Hidden in Your Verilog)", *SNUG* (*Synopsys Users Group*) 2003 User Papers, September 2003.
- [2] Don Mills and Cliff Cummings, "RTL Coding Styles that Yield Simulation and Synthesis Mismatches", *SNUG (Synopsys Users Group) 1999 Proceedings*, March 1999.
- [3] Foster, Krolnik, Lacey, <u>Assertions-Based Design</u>, First Edition, Kluwer Academic Publishers, 2003, pg. 121.

10.0 About the Author

Don Mills is an independent EDA consultant, ASIC designer, and Verilog/VHDL trainer with 19 years of experience.

Don has inflicted pain on Aart De Geuss for too many years as SNUG Technical Chair. Aart was more than happy to see him leave! (Not really.) Don has chaired three San Jose SNUG conferences: 1998-2000, the first Boston SNUG 1999, and three Europe SNUG Conferences:

2001-2003. He has authored or co-authored many papers for SNUG since 1997. Don is on various Verilog and SystemVerilog committees with IEEE and Accellera.

Don holds a BSEE from Brigham Young University.