Advanced X-Propagation Features

Revision: 28 February 2012

www.tachyon-da.com

ABSTRACT

Current Verilog semantics of X handling in the IEEE 1364 standard allow many X related bugs to remain undetected, causing many design problems to not be found until gate level simulation. These issues, sometimes referred to as 'X-bugs', arise later at netlist level. This problem vastly increases debugging complexity and cost.

Ideally, these bugs would be caught and fixed during RTL level verification eliminating the need for costly gate level debugging. CVC contains unique X-Propagation features that enable seamlessly adding X-propagation features to designs written in regular Verilog RTL. The Verilog source remains unchanged because X-propagation options are added to change simulation time semantics that expose hidden X bugs. These features save valuable time early in the design cycle. CVC helps locate X-related design flaws which may otherwise go undetected by static formal tools because of the dynamic timing provided by simulation with X-propagation. This document explains how to use CVC to find X-bugs by running CVC 's X-propagation options.

Contents

1 Verilog Standard X Handling

1.1 Standard Verilog X

The Verilog standard contains 4-state values 0/1/X/Z, with X-semantics described as *unknown* for simulation and *don't care* for synthesis. The X-semantics for the Verilog language reference manual results in many undetected RTL level Xs. The Xs later appear causing post-synthesis gate level simulations to fail. A simple example of a disappearing X illustrates how this may occur:

```
if (rst)  a = 1;
else      a = 0;       //rst is 'x this will get executed, was the intention that it be 1/0?
```

If rst is 'bx, normal Verilog semantics treat the 'bx as false. The *else* clause is then selected so the assignment is "a = 0". The X value that passed undetected may not indicate a design problem. In addition, X values may be used to achieve better synthesized results because X's as *don't cares* allow the synthesizer more flexibility, or sometimes engineers intentionally inject Xs into designs to help expose bugs.

1.2 Verilog X References

Rather than describe all the issues with Verilog X-bugs, we refer readers to an excellent paper by Mike Turpin titled "The Dangers of Living with an X (bugs hidden in your Verilog)". This paper goes into great detail about the dangers of disappearing Xs in Verilog RTL and the problems that arise due to standard Verilog X-semantics. It is a suggested read for those looking for a complete Verilog X problem explanation.

The form of pre-synthesis RTL X-propagation implemented in OSS CVC was defined by Renaesus.

2 CVC X-Propagation

2.1 Changes to Verilog RTL Semantics with X-Propagation

X-propagation changes the behavior of procedural RTL simulation in an attempt to find design errors caused by X (unknown) values that are removed by the current IEEE 1364 Verilog standard semantics and therefore need to be caught by gate level simulation. The goal of this new X-propagation feature is to remove the need to debug a design at gate level by checking for X-bugs at the RTL level when simulations run much faster and debugging is simpler. The idea behind detecting X-propagation bugs during RTL simulation is that X values during gate level simulation mean either 0 or 1. CVC 's X-propagation features conceptually simulate the RTL so that first the simulation is done with X=0 and then with X=1. The two results are then conceptually combined.

The algorithm works by changing the semantics of *if* and *case* statement when the select contains unknowns (Xs) to substitute first 0 and then 1 for X bits and then combine the results of the two conceptual simulations. *Always* block event controls (@ statements) that contain to and from X as well as potential *posedge* and *negedge* are changed to conceptually simulate as if the X value transition was either a 0 or a 1 and then conceptually combined the results. The key concept is the idea of X-propagation is to find synthesizable parts of a module's RTL that contain only procedural and non-blocking assigns. Every *if, case, always* event control statements are marked as eligible or not. Eligible generally means synthesizable, i.e. nested cases that are eligible for X-propagation increase pessimism and do not have Verilog constructs that require current Verilog X semantics. The X-propagation ifs, procedural assigns (=), non blocking assigns (<=), and non blocking with delays (<= #constant). Any *if, case, or always* item with blocking delay controls, task enables, etc, will not be marked as X-propagation eligible. Any procedural code with procedural blocking time movement will not be X-propagation eligible.

When an eligible *if* or *case* statement is executed during simulation and the select has any X bits, a special algorithm is used that combines the unknown X values to ensure all possible matches that when they are combined give the same result. The special X-propagation algorithm selects all *if* clauses and *case* items that can possibly match, combines all the right hand sides and then at the end, still within a single Verilog time unit, performs the procedural assignment or schedules the assignment for non blocking assigns. The bits are combined just like standard Verilog for the question mark colon (?:) operator: 0-0 = 0, 1-1 = 1, and all other possible bit combinations are set to X. If one variable is set in one possible selected block route and not the other, it will be set to X.

2.2 X-Propagation *If* Statement Simulation

If statements without X/Z selection expressions simulate following standard Verilog rules. If the selection expression has Xs, then if there is no *else,* all variables on the left-hand-side will be combined with their original values upon entry to the *if* statement. If the *else* clause is present, the *if* clause and *else* clause are both executed in X-propagation mode. This means that for any procedural assignment or non blocking assign, assignments in either the *then* clause or the *else* clause are combined using the normal compatible value merging rules. The combining of right hand sides assigned to the same left hand side variable is scalared so each bit is tracked separately. Therefore bit selects and part selects will

only change the selected bits.

Some simple *if* examples:

***if* example 1:**

```
//during simulation rst = 'bx, will combine (0, 1), thus setting a = 'bx since the true intention
//of the unknown value isn't known

if (rst)  a = 1;
else      a = 0;
```

***if* example 2:**

```
//during simulation rst = 'bx, will combine (1, 1), thus setting a = 1 since the regardless of
//the unknown value a will always be 1

if (rst)  a = 1;
else      a = c;     //c is 1
```

***if* example 3:**

```
//during simulation rst = 'bx, will combine (101, 000), thus setting a = 3'bx0x  regardless of
//the unknown value a will always have '0' for the second bit, but the other two are unknown

if (rst)  a = 3'b101;
else      a = 3'b000;
```

***if* example 4:**

```
//during simulation rst = 'bxxx and no else clause the values will  combine the value on entry to
//the if statement with on exit .  So combine the two values (101, 000), thus setting a = 3'bx0x

//value on entry a = 3'b101;
if(rst)  a = 3'b000;
```

***if* example 5:**

```
//during simulation rst = 'bx, will combine as above a = 3'bx0x, but b is not set in the else block
//therefore b will be set to 3bxxx

if (rst)
  begin
    a = 3'b101;
    b=3'b111;
   end
else  a = 3'b000;
```

2.3 X-Propagation *Case* Statement Simulation

Simulation of c*ase* statements is similar to *if* statement except multiple *case* item matches are possible. Notice for normal standard Verilog, at most one case item can be selected. For X-propagation *case* statement with any X bits in the select, *case* items are selected if they are compatible. Compatible means either the scalared bit is the same or the select bit or the case item bit is an X. If no case items are compatible and there is no *default* item, the case statement is skipped. If there are X bits in the *case* selection expression and a *default* item is present, all case items that are compatible and the default are executed using the assign combine X-propagation algorithm.

Some simple *case* examples:

***case* example 1:**

```
s = 4'b01x0;
case (s)
 4'b0000:    a = 3'b000;
 4'b0100:    a = 3'b010;    //first match
 4'b0110:    a = 3'b100;   //second possible match
endcase
```

The two possible right hand side values are combined to assign a = 3*'bxx0;*

***case* example 2:**

```
s = 4'b01x0;
case (s)
 4'b0000:    a = 3'b000;
 4'b0100:    a = 3'b010;    //first match
 default:      a = 3'b101;    //all possible combinations for 'x' are not accounted,  include default
endcase
```

The two possible right hand side values are combined like before but now the default must also be combined to assign a = 3*'b10x;*

***case* example 3:**

Bits are preserved and combined for each bit. This example illustrates a part-select combining and preserving:

```
b = 3'b101;
a = 3'b00x;
case (a)
 3'b110:     b[1:0] = 2'b11;
 3'b001:    b[1:0] = 2'b00;     //first case match
 3'b000:    b[1:0] = 2'b01;     //second case match
```

```
        endcase
```

The b[2] bit is preserved since it is never assigned to and bits [1:0] are assigned the combined values, b = '3b10x;

combined *case/if* example:

Any combination of if/case/assign nested statements will combine the right hand side values for X selects and assign.

```
        If1 = 'bx;
        c = 'bx;
        if (if1)
         begin
          a = 4'b1111;
          b = 4'b1111;
         end
        else
         begin
          case (c)
           2'b00:
            begin
             a = 4'b1111;
             b = 4'b0111;
            end
           2'b01:
           begin
            a = 4'b1111;
            b = 4'b1111;
          end
        endcase
end
```

The *if* statement combines the *if/else* section along with the nested *case* statement to combining values to reach: *a=1111 b=x111*.

3 X Detection of *always @(X)* Events

CVC's X-propagation has the capability to check for X related potential edges that may or may not trigger the change operator for event controls on *always* @ blocks. For variable state transitions that change to or from values containing an X, special X-propagation handling will occur when the always block event control guard is evaluated. When the transitions bits are 0->X, X->0, 1->X, X->1, CVC saves the value on entry to the block for all variables on the left-hand-side. These values are then

combined on exit using the new updated values set inside of the body of the always block, if the block can possibly be executed. The idea is that a potential edge may or may not trigger the always event control when the X transition has 0 and 1 substituted. If it is possible for the event control to not trigger, the left hand side values on entry to the always event control block will be combined with the assignments that would occur if the always event control were triggered.

There are two types of potential X-propagation edges. The first type involves *posedge* and *negedge*. For X->1 and 0->X transitions, the potential *posedge* may or may not trigger the always event control. For *negedge*, X->0 and 1->X transitions may or may not trigger the event control. Transitions to and from 'Z' values do not trigger X-propagation. Variables wider than one bit (vectored) will trigger as normal except when only X (no 1 or 0) transitions are made.

The potential always edge X-propagation feature is complicated and can easily cause otherwise correct designs to not correctly initialize so there are CVC **+xprop** options which allow: turning off all edge X propagation features, only perform X-propagation always edge simulation on scalars (including *posedge* and *negedge*), and only perform **+xprop** edge simulation for *posedge* and *negedge*. See section 4 for an explanation of how to use these options with **+xprop**.

Some simple examples will illustrate how CVC handle X-propagation on always blocks:

X edge example 1:

```
//On the posedge transition of 'clk' 0->X, it will combine the value of 'rst' upon entry to the block to
//the value of 'rst' on exit.  So if rst = 0 on entry it will combine the value (0,0) = 0, however if the
//value on entry is 1 it will result in (1, 0) = X

    always @(posedge clk)  rst = 1'b0;


//CVC also checks for X transitions on vectored variables.  These are only combined on transitions
//which are a result of no real 1/0 bit changes just values which go from 'X'.
```

X edge example 2:

```
    //vec has transition 2'00 -> 2'b0x  will handle this as an X transition
     always @(vec)  val = 2'b10;
```

X edge example 3:

```
    //vec has transition 2'00 -> 2'b1x  will handle this will not handle has an X transition
    //as it had a real bit change from 0->1
     always @(vec)  val = 2'b10;
```

4 CVC X-Propagation Invoking Options

CVC has a number of options related to X-propagation to expose and locate X-bugs and control X-propagation for your design. X-propagation options can be invoked using the following options:

+xprop Turn on X-propagation. This turns on CVC X-propagation feature which helps locate and resolve possible bugs due to Xs in the design. This is the main option and must be selected. The other **+xprop** options modify the behavior of this option.

+xprop2 Turn on X-propagation style 2. This turns on advanced CVC X-propagation which helps locate and remove Xs all together from the design. This can be used instead of **+xprop**.

+xtrace Turn on X-propagation tracing for all conditions which trigger X-propagation. All tracing is placed into a text file with default name 'cvc.xprop'.

+xtracemax=[number] Set the maximum number of trace statements to record. This is used along with **+xtrace** to avoid creating large 'cvc.xprop' files during simulation. If not used, the default 1000 statements will be written. Set **+tracemax**=0 to turn the limit off.

+xprop_eval Turn on much more pessimistic algorithsm for RTL binary and unary reducing |, & type operators

+xprop_exclude Dump a file called 'cvc.xprop.exlcuded' which reports all statements which are not included in X-propagation giving the exclusion reason for each statement.

+no_xedges Do not include always @(x) X edge detection for X-propagation.

+no_vector_xedges Do not include always @(x) X edge detection for variables greater than one bit (only scalars).

+only_pos_neg_xedges Only include always @(posedge/negedge). Regular @(var) without posedge/negedge will not be included in X-propagation X detection.

4.1 Alternative Style 2 X-Propagation (+xprop2 option)

CVC also implements another way of propagating X's in designs that does not interact with the **+xprop** option. The **+xprop2** option can be used as an alternative option to **+xprop**. Namely, using the **+xprop2** option will check every *if* and *case* select for X values, if X values occur during simulation it will execute all statements with the right hand side values assigned to X. So it will assign X values to all variables assigned to beneath the *if/case*, thus propagating Xs. This does no 'combining' of values as **+xprop** does. It just assigns the values to X. The advantage of **+xprop2** is the performance is much better and it is used to find X values from designs for which no X's are allowed.

4.2 Locating X-Propagation Occurrences During Simulation

It order to find locations where Xs occur during simulation when X-propagation code is entered. The **+xtrace** option can be used along with **+xprop**/**+xprop2**. A message is written to a file called 'cvc.xprop' during simulation giving the time and statement location for every eligible **+xprop**

statement whose condition evaluation has X bits. The 'cvc.xprop' file will contain a list of statements where X-propagation occur, an example *if* statement has the following format:

```
########## X if condition    ############ #####
At time 2000
if (rst)  => 'bx
At clock.v : 1255
#########################################
```

4.3 X-Propagation of Further Xs in Expressions (+xprop_eval option)

CVC has an option that makes procedural binary expression *and* and *or* operators more pessimistic. For binary operators '|', '&', '||', and '&&', along with unary '&', '|', '~|', "~&' operators, the standard Verilog evaluation rules that for '|', 1 means true and for '&' and 0 means false are changed. The standard Verilog semantics for these can cause Xs to 'disappear', using +**xprop_eval** will preserve Xs across these operands. The **+xprop_eval** feature can be used with or without **+xprop**.

For example:

```
a = 'bx1;
c = 'b11;
b = a | c;   //regular Verilog gets b=2'b11, thus the X 'disappers'.
```

with **+xprop_eval** the | operator preserves the Xs, so b = 2'bx1.

4.4 Locating Which If/Case/Always Statements Are X-Propagation Eligible

Currently CVC X-propagation is restricted to if/case/assign statements. All other statements will not be included with X-propagation and will be simulated using regular Verilog semantics. The **+xprop_excluded** option will produce a file called 'cvc.xprop.excluded' which will list all the *if/case* statements that for some reason are not eligible for **+xprop** processing. All statements are included as eligible for **+xprop2** propagation.

Reasons for ineligible **+xprop** statements:

- Illegal statement type (only if/case/assign allowed).
- Net is used on both lhs/rhs of procedural assigns (=) in the *if/case.*
- Select expression is a real value.
- Select expression is a value declared to be 2 state.
- Arrays are used under the *if/case*.
- Nets on the lhs are forced using *force.*
- Hierarchical references are not allowed on the lhs.
- The assign types do not match for each net under the *if/case*.
- Variable indexed part selects ([+:] , [-:]).
- If conditionals that use !== and ===.

4.5 Selecting or Deselecting Specific Module Types for +xprop Simulation

Instead of using **+xprop** to turn on X-propagation for an entire design, it may be desirable to only turn it on for certain modules. Common use is turning on X-propagation for the design under test (DUT) but not the testbench.

The CVC config file mechanism used for setting toggle coverage and PLI access is also used for selecting and deselecting modules for which **+xprop** simulation is needed. Use the "**-optconfigfile [file name]"** option where in the design to turn on **+xprop** simulation for specific modules.

For +prop simulation for some modules use:

module {<module type name list>} {xprop};

To turn off +xprop simulation for some modules use:

module_exclude {<*module type name list*>} {xprop};

To turn on +xprop simulation for all modules types under a particular instance use:

tree (-1) {<*module type name list*>} {xprop};

This is the most common case to turn on X-propagation for all module beneath the design under test:

tree (-1) {DUT} {xprop};

The properties can be **{xprop}** for **+xprop** for some module types, {**xprop2}** for **+xprop2**, and **{xprop_eval}** for **+eval**.

Notice all the **+xprop** options must be turned on for all instances of a module type even if the "**instance {instance name list} {xprop};"** form is used, **+xprop** is turned on for all instances of the module type that the instance has.

See file how_to_use_new_toggle_coverage_feature.README in the release doc directory for more information on using the **-optconfigfile**.