2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Challenges of VHDL X-propagation Simulations

Karthik Baddam, Imagination Technologies, Kings Langley, UK (karthik.baddam@imgtec.com)
Piyush Sukhija, Synopsys Inc, Reading, UK (piyush.sukhija@synopsys.com)

Abstract

Finding X bugs in a timely fashion is important for a successful tape out. Verilog simulation based tool that addresses X issues were presented before. In this paper we present VCS VHDL simulation based X-propagation tool, highlight challenges encountered in the process and how they were addressed. We request the IEEE P1076 VHDL Analysis and Standardization Group (VASG) [1] to consider adding realistic X-propagation semantics into the LRM. We discuss debug challenges faced using the X-propagation technology and present suggestions on how to minimise debug time. We also open up the discussion of X specific coverage.

1 Introduction

In digital hardware design, X is used as a modelling construct to model unknown or don't care value. Digital gates in silicon cannot output X value, they can only output logic 0 or 1. When X value is seen at a net in simulation, the same net in silicon could take the value of logic 0 or 1. RTL synthesis tools interpret Xs differently to verification tools (such as simulation and formal tools) and because of this difference, Xs in design can lead to simulation-silicon mismatch. In the context of VHDL designs, by X we mean the standard logic type with values of X, U and – (don't care). The issues surrounding Xs has been discussed before in [7, 10]. We summarize the key issues here. Sources of X in RTL include:

1. Power aware simulations, isolation cells, power island.
2. Explicit X assignment in designs.
3. Implicit X sources that includes uninitialized flops, latches, memories and floating signals.
4. Functional violations such as floating buses, bus contention, range overflow, divide by 0.
5. Multiple drivers.

So what is the problem with X? The current HDL language LRM (such as Verilog and VHDL), treats X as a distinct value, whereas in silicon there is only logic 0 or 1. So any comparison that involves X with a non X value would not match, but in silicon it is nondeterministic. So in simulations that consider X to be distinct value stop the X from propagating [6, 7, 10]. In this paper we refer to this behaviour as X-optimism.

```
1  signal cond, a, b, c: std_logic;
2  if (cond = '1') then
3      c <= a;
4  else
5      c <= b;
6  end if;
```

Code 1: Example VHDL code to demonstrate X Optimism

cond	*a*	*b*	*c* (LRM)	*c* (silicon)
X	0	0	0	0
X	0	1	1	0 or 1
X	1	0	0	0 or 1
X	1	1	1	1

Table 1: RTL X-optimism compared to silicon behaviour for example in Code 1

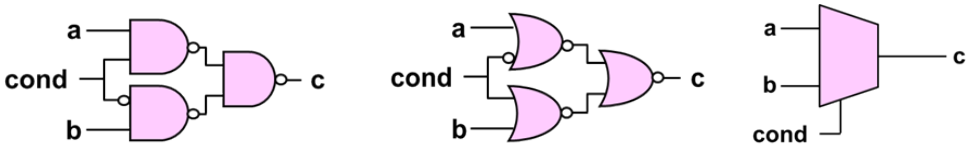
As an example, consider the example code in Code 1. If the signal *cond* is unknown, then as per VHDL LRM, X is not equal to logic 1, so *c* gets the value of *b*. The Table 1 summarizes the RTL X-optimism issue for example in Code 1. The problem with X optimism in this example is when *cond* is unknown and the inputs have different value.

2 Existing Solutions

2.1 Gate level Simulations (GLS)

Traditionally GLS has been used as a way to catch X related issues. However GLS is run too late in the design cycle, as the design needs to go through synthesis process for a gate level netlist to be available. There is another problem with GLS, in that it tends to be too pessimistic [7]. Debugging at GLS level is also a difficult task, as the debug is performed on a transformed view of the design.

The example listed in Code 1 can be synthesised into gates in different ways, some of which are shown in Figure 1. Table 1 summarizes the difference in RTL X-optimism and GLS pessimism for this example.



(a) GLS 1 (b) GLS 2 (c) GLS 3

Figure 1: Different gate level implementations of example Code 1

cond	*a*	*b*	*c* (LRM)	*c* (silicon)	GLS 1	GLS 2	GLS 3
X	0	0	0	0	0	X	0
X	0	1	1	0 or 1	X	X	X
X	1	0	0	0 or 1	X	X	X
X	1	1	1	1	X	1	1

Table 2: RTL X-optimism vs GLS X-pessimism

2.2 Random Initial Values and 2 State Simulations

It was suggested to eliminate X and use random initial values [2]. This can be achieved by using 2 state simulations or by initialising signals that have X with a random value. This approach eliminates X-optimism and X-pessimism issues. However as Sutherland noted in [7], these approaches end up masking X bugs and that ensuring all combinations of initial values to test is difficult.

2.3 Simulator Specific X-propagation Options

VCS simulator from Synopsys [8] has proprietary (non LRM compliant) options to change X-optimism at RTL. Experience of using such a simulation tool for Verilog HDL is presented in [5]. This solution is called as X-propagation. The general concepts of this are summarised here.

X-optimism at RTL happens at conditional statements (like `if`, `case`). X-propagation aims to propagate Xs where ever a conditional statements encounter X. Revisiting the example in Code 1, in X-propagation mode when *cond* is unknown and the inputs have different value, the output *c* value will propagate X. VCS X-propagation technology offers two options to propagate Xs. One is a realistic mode and another pessimistic mode. VCS refers to them as Tmerge and Xmerge respectively. Table 3 summarizes the RTL X-propagation output for example in Code 1.

cond	*a*	*b*	*c* (LRM)	*c* (silicon)	X-propagation realistic	X-propagation pessimistic
X	0	0	0	0	0	X
X	0	1	1	0 or 1	X	X
X	1	0	0	0 or 1	X	X
X	1	1	1	1	1	X

Table 3: RTL X-propagation for example in Code 1

2.4 Formal Model Checking Tools

Formal model checking tools can exhaustively verify properties written in specification languages such as PSL or SVA hold for all combinations of input sequences. Certain model checking tools such as Cadence IFV [3] also support X modelling and X-propagation in a similar way to the VCS X-propagation. These tools also have the ability to automatically add X-propagation specific assertions.

Turpin in [11] suggested sequentially comparing the design to itself using formal tools. The idea here is that for any X sources in the design, the formal tool will try all possible combinations of values. Formal tools such as Cadence JasperGold [4] offer ability for such kind of X-propagation analysis.

Because of the exhaustive nature of formal model checking tools, it is often possible that the tools find false failures due to missing constraints. To avoid these false failures, users have to develop properties that constrain the inputs to valid sequences or users can waive these failures. Given the exhaustive nature of formal model checking, it is logical to use formal X-propagation solution for designs that already have formal verification environment and to designs where input constraints can be added within project timescales.

Real Intent's Ascent XV [6] does structural and formal analysis to report sources of X. It also has capability to interact with logic simulators to report and correct X-optimism behaviour at run time. We did not use this tool and its suggested methodology, so cannot comment on its effectiveness.

3 Why Simulation Based X-propagation?

Verification is signed off based on a verification environment, typically after achieving a target coverage goal. So validating the design for Xs based on this environment ensures that there are no X related issues for the sign off coverage. If the verification environment is simulation based, then using X-propagation simulation ensure the sign-off coverage is validated for X. The effort involved in enabling X-propagation for an existing environment is also very minimal, for VCS additional elaboration options are sufficient.

GLS is also used to for other reasons apart from finding X bugs, such as catching simulation synthesis mismatches missed by equivalence checking, timing violations not caught by static timing analysis tools, power analysis, power intent validation. Since GLS is simulation driven, it also makes sense to run RTL X-propagation before starting GLS.

4 VCS X-propagation for VHDL

Verilog X-propagation is already implemented and available, so porting it to VHDL should be straight forward. Verilog and VHDL are different languages, even though both of them are primarily used for digital logic design. Most of the concepts from Verilog X-propagation can be ported to VHDL. However VHDL approach to modelling digital hardware actually makes it difficult to apply X-propagation. In this section we cover the issues faced and approaches to address them.

4.1 VHDL Conditional Statements

All conditional statements in VHDL should be X-propagation aware. These statements include `if`, `case`, `when−select`.

4.2 VHDL Two State Data Types

VHDL has two state and nine state data types. Commonly used two state types are `integer`, `boolean`, `enum`, `bit`, `bit_logic`. Two state types cannot have value X. Commonly used nine state types are `std_logic`, `std_logic_vector`, `unsigned`, `signed`. For X-propagation it is necessary to allow two state types to have X for the portions of VHDL code that is used to model hardware design.

4.3 VHDL Enums

4.3.1 Problem

VHDL enums are commonly used to encode state machines (eg: `type STATE_TYPE is (INIT, PROC);`). However as per LRM, the enum type can only have values defined in its type. Enum based state machines, if left uninitialised, default to the first value of the enum type. This default behaviour can mask X bugs in RTL simulations.

4.3.2 Solution

A solution to address this would be to implicitly add an extra X value to the enum type as its first value. This will cause the uninitialised enum based signals by default to the implicitly created value. Any conditional statements that use enum signals should check for the implicitly created X value and propagate X accordingly.

4.4 VHDL Integers

4.4.1 Problem

It is common in VHDL to use `integer` for digital hardware design. For example, it is common to convert nine state type to `integer`, perform datapath type operations (such as addition, multiplication, etc.) and convert the result back to nine state type. As per LRM, when such conversions happen the X is always converted to logic 0. However the same conversion cannot be guaranteed in silicon.

4.4.2 Solution

There are two potential solutions to address this. One where all `integer` data type have an implicit X value, similar to the enum types discussed earlier. Converting an X value to an `integer` should result in this implicit X value. This solution is pessimistic than real silicon, as even a single bit being X in the source type will result in X

The second solution is to fundamentally change the `integer` data type to allow them to model X, which would be similar to VHDL `unsigned` data type.

VHDL users can also evaluate their code to use `unsigned` data type instead of `integer` where applicable and avoid this problem.

4.5 VHDL Integer Divide By Zero

4.5.1 Problem

Most of the VHDL simulators exit with an error message when they encounter divide by zero. This scenario is quite likely in X-propagation when nine state to two state conversions result in logic 0 for Xs. However in any digital circuit that implements division function, the result of a divide by zero is allowed (unlike simulators which exit).

4.5.2 Solution

Divide by zero should result in X.

VHDL users can also evaluate their code to use `unsigned` data type instead of `integer` where applicable and avoid this problem.

4.6 VHDL Array Index

4.6.1 Problem

In VHDL, arrays and multi-dimensional arrays are declared using types. An example is shown in Code 2. All array types including the commonly used `std_logic_vector` are declared to use `integer` as index. So to access an element from this array, one would have to use an `integer` data type.

VHDL arrays and multi-dimensional arrays (MDA) are frequently used to model hardware. MDAs are also used to model memory macros. Using an integer index with such arrays to read and write is common in HW designs and one of the HW structures this can synthesise to is an mux. So it is desirable for X-propagation to handle array accesses realistically.

First, any conversion to integer looses X, there by masking potential bugs. The result of reading from an array with an index X should propagate X appropriately. Similarly the result of writing to an array with an index X should propagate X. Same problem exists for multi-dimensional arrays.

```
1   type MY_NIBBLE_TYPE is array (0 to 3) of std_logic;
2   type MY_NP_ARRAY_TYPE is array (0 to 2) of std_logic;
3   signal my_bit_input : std_logic;
4   signal my_bit_output : std_logic;
5   signal my_nibble : MY_NIBBLE_TYPE;
6   signal my_np_array : MY_NP_ARRAY_TYPE;
7   signal my_nibble_index_int : integer range 0 to 31;
8   signal my_nibble_index : unsigned(1 downto 0);
9   --
10  my_nibble_index_int <= to_integer(my_nibble_index);
11  --
12  my_nibble(my_nibble_index_int) <= my_bit_input;
13  my_np_array(my_nibble_index_int) <= my_bit_input;
14  --
15  my_bit_output <= my_nibble(my_nibble_index_int);
16  my_bit_output <= my_np_array(my_nibble_index_int);
```

Code 2: Example VHDL code for arrays

4.6.2 Solution

Firstly, conversion of nine state to integer should propagate X.

For array writes, when an index in an array write operation is X, then every element in the array should be merged with the write value. As per the example in Code 2, a write with an index X should result in the merge of my_bit_input with every element in the array my_nibble.

For array reads that are power of 2, When an index in an array read operation of a power of 2 array is X, then the result should be a merge of all the elements in the array. As per the example in Code 2, a read with an index X from power of 2 array my_nibble, should result in the merge of every element in the array my_nibble.

For array reads that are non-power of 2, When an index in an array read operation of a non-power of 2 array is X, then the result should be X. The reason being, the index can be out of array bounds, in which case the result is unknown. As per the example in Code 2, a read with an index X from non-power of 2 array my_np_array, should result in X.

4.7 VHDL Data Operations

4.7.1 Problem

Common data operations, such as addition, multiplication, shift, division, etc. are not X-propagation aware.

Consider the addition involving two 4 bit vectors (z <= a + b;, result of the addition, z, will contain X if any of the operands contain X.

signal name	example values 1	example values 2	example values 3
a	000X	000X	010X
b	0X00	0X0X	0X0X
z X-propagation aware	00X0X	00XXX	0XXXX
z non X-propagation aware	XXXXX	XXXXX	XXXXX

Table 4: X-propagation aware addition example

4.7.2 Solution

Data operations should be X-propagation aware. X-propagation aware result for the addition example discussed is shown in Table 4. The idea here is that the result of a bit 0 plus X will result in X, but its carry will not be X. Similarly a bit 1 plus X will result in X along with its carry. And an X plus X will result in X along with its carry.

At the time of writing this paper, VCS solution supported realistic X-propagation solution for all the issues described here apart from the data operations.

Note to IEEE P1076 VHDL Analysis and Standardization Group (VASG)

We request the IEEE P1076 VHDL Analysis and Standardization Group (VASG) to consider adding realistic X-propagation semantics highlighted in this paper into the LRM. Motivation for this request is to make VHDL X-propagation semantics match realistic silicon behaviour as closely as possible, and more importantly, to get tool vendors to consistently model X-propagation.

5 Bugs and False Negatives

As part of using X-propagation, we found certain bugs as well as false negatives. This section describes them.

5.1 Bugs

We found typical non-reset flop bugs through X-propagation that can also be found by GLS, only difference being easier and sooner than GLS. One type of bug that we didn't expect to find was a read before write bug. The bug was a functional in nature, in a block that was still in design verification cycle, but was missed because of X-optimism.

5.2 Self Gating

In [5], Evans et al. discussed re-convergence problem in X-propagation. We faced a similar issues, which we termed it as *self-gating*. Consider the dumb down example in Code 3. In real silicon, the signal a would be known at the second rising edge clock, however in GLS or X-propagation simulation signal a will always be X. Waveform for silicon expected behaviour is shown in Figure 2.

Initially we worked around this problem by depositing the signal a with a non X value. However in silicon there is still scope for the logic driven by a to load its unknown value in the first clock cycle.

After discussing this coding style pattern with VCS R&D folks, they came with an implementation where X-propagation is disabled for branch statements that have self gating (line 3 in this example). But even with this workaround, signal a still remains at X. To fully work around this issue, we had to rewrite the logic in line 3 to "`if (a /= '0') then`".

```
1   process (clk) begin
2   if (rising_edge(clk)) then
3       if (a = '1') then
4           a <= 0;
5       end if;
6   end if;
7   end process;
```

Code 3: Example VHDL that shows self-gating

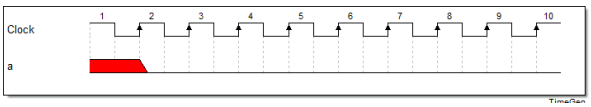


Figure 2: Expected silicon behaviour of self gating example for Code 3

We have also seen cases where the self gating structure is separated by an intermediate signal. The only way for us to workaround this is to deposit them. The feedback from the designer of this particular style was that they could have rewritten this code to avoid the problem if known earlier in the design cycle.

6 X-PROPAGATION DEBUG

Debug of X-propagation simulation failure is generally much better than that of GLS. None the less it is still a time consuming task to root cause the source X. Debug tools such as Verdi [9] provide features such as *Active X Tracing* to aid in debug. In our experience, such features can only help if one knows where to look for. To elaborate, if any white box checks are triggered due to an X bug, then the debug can start with this check to trace the source X. If none of the white box checks are triggered, and the simulation test locks up, then the debug involves tracing the inputs thought the design. In our experience such X bugs are more difficult to debug compared to when a white box check is triggered.

As part of our collaboration with VCS R&D team on X-propagation, we explored options to improve the debug time of X-propagation simulations. These are described below.

6.1 X-propagation Flip Flop output transitioning to X debug aid

When an X is loaded into a flip flop, it could imply one of the three things. 1. A buggy X is propagated. 2. A non buggy X is propagated, but is not necessary to load the X. Can save power by not loading this X. 3. X is loaded as part of initialisation sequence and is OK to load the X.

VCS now has a configurable option that generates messages whenever an X is loaded into a flip flop. The idea here is to know when an X is propagating from one pipeline stage to the next. In an ideal world none of these messages should be reported. However we have seen behavioural ram models that purposely assign X to the output flops. The configurable nature of this option allowed us to exclude such models from reporting these messages. These messages have been helpful in narrowing the scope of debug in X-propagation simulations.

6.2 X-propagation Lockup Debug Methodology

As discussed earlier, lock up failures in X-propagation simulations are difficult to debug. One approach we took for debugging X-propagation lock up is described below.

1. Get waveforms for both non X-propagation and X-propagation simulation on the same design and test.
   (a) Refer to the non X-propagation waveform as the reference waves.
2. Get a list of all the signals that have X at the end of X-propagation simulation. Refer to this as *potential bug signal list*.
3. From this list remove, signals that do not have any event (value change).
4. For each of the signal in this list, query the reference waves at the test end time, and remove the signal from the *potential bug signal list* if its value is X.
5. Debug each of the signal in this list to find the source of lock up.

In practice we found this method to reduce the debug scope significantly. However it was still time consuming to go through the entire list. Using designer insight in combination of this list helped root cause lock ups in X-propagation simulations sooner.

6.3 Is Fully Automated X-propagation Debug Possible?

Can we root cause X-propagation failures automatically? Given the same design, test and waveforms from non X-propagation simulation, we certainly believe that automatic root causing of X-propagation failure is possible. This is something we would like to use and would encourage our colleagues in the industry to work toward.

7 WHEN TO STOP X-PROPAGATION SIMULATIONS?

How can we convince ourselves that there are no X bugs left in the design? The general approach to X-propagation verification is to rerun all the sign off test cases with X-propagation options turned on and ensure to get the same level of coverage and pass rate as normal simulations. This is the best approach if do able. However there are two types of situations when all the sign off tests cannot be rerun with X-propagation simulations.

One such situation is where an acceleration platform (such as emulation, FPGA based prototyping)

is used to run majority of sign off tests. As of today we are not aware of an acceleration technology that support the X-propagation semantics. Rerunning all the acceleration tests in simulations is not practical due to prohibitively long runtime.

Another situation is where a SoC is assembled from various verified IP. Generally speaking, the SoC integrators only verify integration of these IPs and they don't aim to verify all aspects of the IP at the SoC level. It is possible for one IP to feed Xs into another IP. Rerunning all the integration tests at the IP level will not be enough to catch X bugs introduced in the SoC.

7.1 Flip Flop X State Coverage

Our approach to this problem is as described below. Check the state of every flip flop at the end of test. If any of the flip flops value is X, then save it to a list (in a file). For each test, such a list will be saved. At the end of all the tests, intersection of all the lists will produce flops that were either not tested, or ended in an X. If a flop from this list is never tested, then adding tests to cover those is possible. If a flop ends in X despite a targeted test, will need investigation. If the intersection of all the flops is not empty, then it indicates that the flop is not activated nor tested and it needs addressing. This approach is similar to code coverage, it that, it doesn't prove lack of X bugs but it does highlight lack of tests to cover any potential bugs. If necessary one would extend this approach to all signals in the design, not just flops.

One point to note that in a low power intent simulation, the flop values should be queried before any power shutdown operation.

8 Conclusion

We have highlighted the issue with the current VHDL LRM with regards to RTL X-optimism, and reasons for why GLS is not the correct method to catch X bugs. We have discussed VHDL X-propagation hurdles and how we addressed them in VCS simulator. We would request the IEEE P1076 VHDL Analysis and Standardization Group (VASG) to add realistic X-propagation semantics into the LRM. We have also presented debug strategies for X-propagation and finally opened up the discussion on X-propagation coverage using flip flop X state coverage.

References

[1] IEEE P1076 VHDL Analysis and Standardization Group (VASG), 2015. URL http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome. [Online; accessed 09-May-2015].

[2] Lionel Bening. A two-state methodology for rtl logic simulation. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 672–677, 1999. doi: 10.1109/DAC.1999.782029.

[3] Cadence. Incisive Formal Verifier, 2015. URL http://www.cadence.com/products/fv/formal_verifier/Pages/default.aspx. [Online; accessed 09-May-2015].

[4] Cadence. JasperGold: X-propagation verification app, 2015. URL http://www.jasper-da.com/products/jaspergold_apps. [Online; accessed 09-May-2015].

[5] Adrian Evans, Julius Yam, and Craig Forward. X-propagation: An alternative to gate level simulation. In *Synopsys Users Group Conference*, San Jose, 2012.

[6] Lisa Piper and Vishnu Vimjam. X-propagation woes: masking bugs at rtl and unnecessary debug at the netlist. In *Design and Verification Conference*, San Jose, 2012.

[7] Stuart Sutherland. I'm still in love with my x! In *Design and Verification Conference*, San Jose, 2013.

[8] Synopsys. VCS, 2015. URL http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx. [Online; accessed 09-May-2015].

[9] Synopsys. Verdi automated debug system, 2015. URL http://www.synopsys.com/Tools/Verification/debug/Pages/Verdi-ds.aspx. [Online; accessed 09-May-2015].

[10] Mike Turpin. The dangers of living with an x. In *Synopsys Users Group Conference (SNUG)*, Boston, 2003.

[11] Mike Turpin. Solving verilog x-issues by sequentially comparing a design with itself. In *Synopsys Users Group Conference (SNUG)*, Boston, 2005.