Improving X debug in X prop and GLS simulations

Karthik Baddam, Piyush Sukhija

Imagination Technologies, Kings Langley, U.K
Synopsys Northern Europe. Reading, U.K

www.imgtec.com
www.synopsys.com

ABSTRACT

In RTL simulation the unknown state, X, which can be 0 or 1 introduces a verification challenge. X-optimism inherent in RTL simulation of some coding styles can filter an X and prevent it from reaching an observation point where it can be detected. As a consequence, time-consuming gate-level simulation (GLS) has been traditionally used to catch X-issues late in the design flow.

X's may simply be introduced by flip flops which due to oversight have not been initialized. The challenge of X is further complicated both by modern power-management techniques which turn off power to parts of the design which are not in use and initialize related registers to X. Also, if 3rd party IP is used, it may not be practical to design out X-optimism in the IP code. The risk of non-detection of X errors increases.

RTL-based simulation X-propagation features can help in catching X issues by alleviating X-optimism. X's created by uninitialized flip flops or power management techniques can propagate through logic, even 3rd party IP. Multiple detections of X, however, may trace back to a single source. Consequently, X debug, even with X-propagation techniques can be tedious and time-consuming task.

In this paper we present an algorithm that aims to minimize debug time of Xprop simulation failures, and locate potential root cause of failures in fully automated way. An implementation of Verdi-based algorithm is also presented.

Table of Contents

1. Introduction – Understanding X problem

1.1 What is X ?

In digital hardware design, X is used as a modeling construct to model unknown or don't care values. Digital gates in silicon don't output an X value; they can only output 0 or 1. When an 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 X's differently to verification tools and because of this difference X's in design can lead to simulation-silicon mismatch. Sources of X 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 contentions, 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 of a signal with the value X with a non-X value would not match resulting in an alternative branch of code being selected, which generally doesn't propagate the X. In silicon the value of X means 0 or 1 and must propagate as 0 or 1. In this paper we refer to this behavior as X-optimism.

```
if (cond = '1') then
    c <= a;
else
    c <= b;
end if;
```

Code 1: Example RTL 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 behavior 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 and the X-value of cond doesn't propagate. 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. The unknown value of cond creates an indeterminate result on c, namely 0 or 1. The X has be filtered out of the RTL simulation and there is no indication of an indeterminate result.

1.2 How are X bugs found?

Various approaches have been suggested to find X bugs. Most of these are mentioned in reference [8]. Formal X verification tools are gaining traction, but are not covered here, as the focus of this paper is in simulation-based debug. Two main simulation based X verification approaches are discussed here.

1.2.1 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. GLS also tends to be much slower than RTL simulation. GLS can be X-pessimistic too pessimistic. 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 synthesized into gates in different ways, some of which are shown in Figure 1. Table 1 summarizes the difference in RTL X-optimism and GLS X-pessimism for the three different implementations.

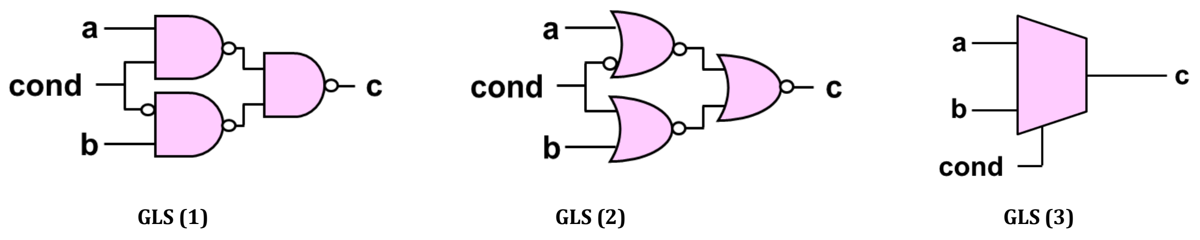


Figure 1: Different Gate Level implementation for Code 1

Cond	a	b	c (RTL)	c (HW)	c (1)	c (2)	c (3)
X	0	0	0	0	0	X	0
X	0	1	1	0/1	X	X	X
X	1	0	0	0/1	X	X	X
X	1	1	1	1	X	1	1

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

1.2.2 Simulator specific X-propagtion option.

Digital simulation technologies have proprietary (non-LRM compliant) options to change X-optimism at RTL. This solution is called as X-propagation. The general concepts of this are summarized here. X-optimism at RTL happens at conditional statements (like if, case). X-propagation aims to propagate Xs where ever 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 (T-merge) and another pessimistic mode (X-merge). Table 3 summarizes the RTL X-propagation output for example in Code 1

Cond	a	b	c (LRM)	c (silicon)	T-merge	X-merge
X	0	0	0	0	0	X
X	0	1	1	0/1	X	X
X	1	0	0	0/1	X	X
X	1	1	1	1	1	X

Figure 3: VCS XPROP behavior for Code 1

2. Current X Debug Challenges

2.1 RTL Simulation Based VCS XPROP

VCS X-propagation allows us to catch X issues early during RTL simulation with design visibility. X-propagation T-merge algorithm removes X-optimism barriers for the X to propagate (unlike RTL X-optimism) through the design to an observation point.

One important consideration of simulation based X-propagation solution is that place where the error is detected and the actual X root cause could be in entirely different parts of the design separated by 1000's of clock cycles. Debug of X's using waveforms involves considerable amount of signal driver tracing either through source code and/or temporal view as shown in Figure 4-6. To clarify further please see use case in figure 4-6, signal domain_active (Figure 4) goes X during simulation causing simulation failure.

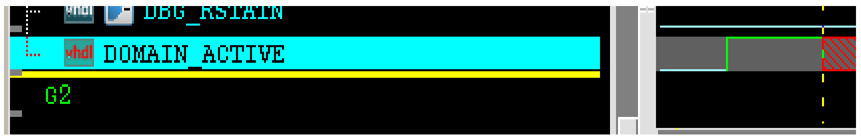
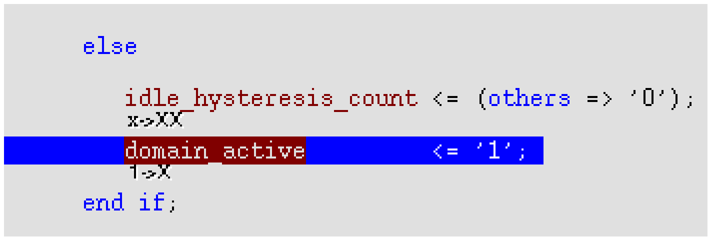


Figure 4: X failure in simulation

To successfully debug this, manual driver tracing (Figure 5) or active driver tracing (Figure 6) in a temporal view can be applied on a signal-by-signal basis. Although active driver tracing greatly speeds up finding the root cause of a single X failure, each root cause can have multiple failure detections. If each X failure detection is debugged one-at-a-time, time will be wasted finding yet another consequence of a previously identified root cause. Alternatively resimulation after each root cause is fixed would also slow down the process and may not be practical for complex designs with many root causes of X's. The challenge is to identify unique root causes of all detected X's.



```
    else
      idle_hysteresis_count <= (others => '0');
      x->XX
      domain_active            <= '1';
      1->X
    end if;
```

Figure 5: X debug through source code

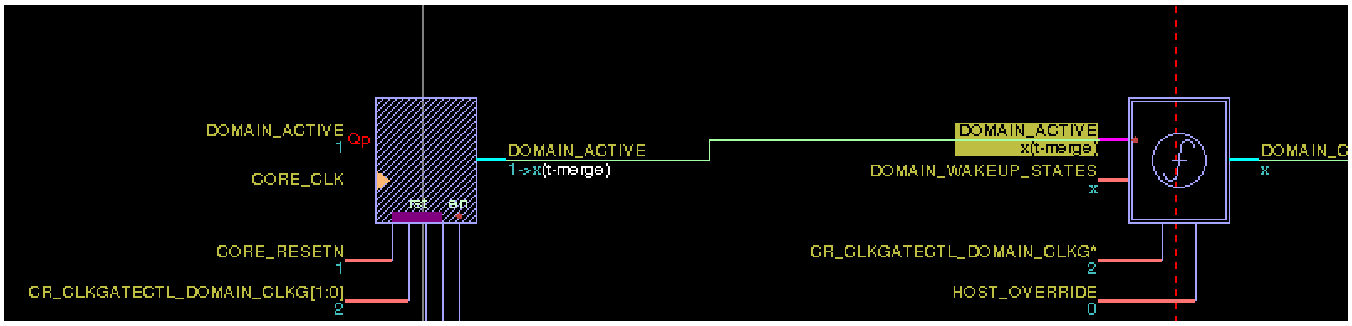


Figure 6: X debug through temporal view

In the following sections we propose a solution to automate the identification of unique root causes of X failures.

3. Design Debug Automation Using Verdi

3.1 What is Verdi Interoperability Apps?

Verdi interoperability Apps are applications developed using API calls (TCL and C) available from Verdi. In this Paper, we discuss an application developed using Novas programming interface that allows users to get signal values from FSDB (Verdi created waveforms database) and KDB(knowledge database for Verdi), and automate X debug traceability, signal value comparisons and the reduction of signal set for debug for unique root causes. Like with other applications, main advantage lies in there reusability across design and thus saving significant debug time.

Detailed information on getting started on Verdi Interoperability Apps and command references are mentioned in References at the end of the paper.

4. Our Proposal for X debug.

Our proposal for X debug relies on Verdi interoperability application that allows users to automate various manual tasks. The application is written in a way that it can be applied to any design which has a Verdi knowledge database, and a Verdi waveform database (FSDB). The application offers significant time advantage for identifying unique root causes of X-failures.

4.1 Where To Begin?

The application will needs few inputs to execute:

1) Failing Design waveform database (FSDB) of a VCS XPROP RTL simulation for a test case.
2) Passing Design waveform database with VCS RTL simulation (non-XPROP/X-optimistic) for same test case used in step 1.
3) Verdi Design knowledge database created for design used in step 1 or 2.

4.2 Algorithm

Based on design debug experience gained from various IMG designs, a method was created to reduce the number of X signals to be debugged in a failing XPROP simulation. That in turn will reduce problem subset to relevant X debug signals to be investigated.

4.2.1 Querying database.

The first step is to extract the X signals are in the failing simulation. The Verdi App uses related APIs calls to query theFSDB database and creates a list of signal with unknown value at the end of simulation. Code 2 below shows partial code for this stage that uses a wrapper (highlighted) utilizing multiple API calls. Detailed code is available at the end of the Paper.

```
//
// Step 1:
//   check max time contains X
//

vector< nFsdbSig_t* > x_sig_vec;
check_max_time_x(fsdb_hdl, fsdb, max_time, x_sig_vec/*O*/);

unsigned int x_sig_size = 0;
for(unsigned int i=0; i<x_sig_vec.size(); ++i) {
  x_sig_size += get_x_sig_size(x_sig_vec[i]);
}
```

Code 2: Querying database

4.2.2 Removing signals with no activity.

Once we have obtained list of signals from Querying Database stage, an API call looks for signals which have no signal value change in simulation. Then this stage will remove signals with no activity on them throughout simulation. Code 3 below shows partial code for this stage that uses a wrapper (highlighted) utilizing multiple API calls. Detailed code is available at the end of this paper.

```
   //
  // Step 2:
  //   check sig VC - remove the signals that don't have VC
  //
  vector< nFsdbSig_t* > vc_sig_vec;
  check_vc(fsdb_hdl, fsdb, min_time, max_time, x_sig_vec,
vc_sig_vec);

  unsigned int x_vc_sig_size = 0;
  for(unsigned int i=0; i<vc_sig_vec.size(); ++i) {
    x_vc_sig_size += get_x_vc_sig_size(vc_sig_vec[i]);
  }
```

Code 3: Removing signals with no activity

4.2.3 Removing signals using passing waveform database.

This stage takes output from section 4.2.2 as input to reduce signals with X further. At this stage we compare remaining signals after stage in 4.2.2 with the corresponding signals from the non-XPROP "passing" simulation. Those signals which were 'U'/'X' in non-XPROP passing simulation plays no role in failing XPROP simulation as passing non XPROP simulation doesn't rely on those signals. Code 4 below shows partial code for this stage that uses a wrapper (highlighted) utilizing multiple API calls. Detailed code is available at the end of this paper.

```
  //
  // Step 3:
  //   check sig X at max time of ref FSDB
  //


  check_ref_max_time_x(ref_fsdb_hdl, ref_fsdb, ref_max_time,
vc_sig_vec/*IO*/);

  x_vc_sig_size = 0;
  for(unsigned int i=0; i<vc_sig_vec.size(); ++i) {
    x_vc_sig_size += get_ref_x_vc_sig_size(vc_sig_vec[i]);
  }

  //

  printf("Begin to dump signals to file...\n");
  for(unsigned int i=0; i<vc_sig_vec.size(); ++i) {
    dump_final_x_sig(final_x_vec/*O*/, vc_sig_vec[i]);
  }
```

Code 4: Comparing with passing non XPROP simulation

4.2.4 Logic reduction using automated root cause.

This stage is the most important stage that reduces X signals which are not actually root causes for driving X and are only propagating Xs as a result of VCS X-propagation. This stage runs automated root cause analysis which checks active driver on signals and in case they are driven from another signal, removes the former signal from the list. With every signal removal we remove redundant logic that is propagating X but is not the actual root cause of simulation failure. This stage takes use of Verdi design knowledge database and runs Active X algorithm on signals. Code 5 below shows partial code for this stage that uses a wrapper (highlighted) utilizing multiple API calls. Detailed code is available at the end of this paper.

```
  //
  // Step 4:
  //   check if x signal's fanin FF has other x signal, if yes,
remove the x
  //   == check x signal is FF, if yes, trace it's fanout,
remove other x signal that is not FF in the fanout
  //

    string sig_name;
    map<string, npiNlHandle>::iterator it;
    vector<npiNlHandle> x_ff_vec;

    for(unsigned int i=0; i<final_x_vec.size(); ++i) {
      sig_name = final_x_vec[i];
      it = ff_set.find(sig_name);
      if( it != ff_set.end() ) {

  is_FF_sig[i] = true;
  x_ff_vec.push_back(it->second);
      }
    }

      }
      sig_name = final_x_vec[i];
      str_set_it = fanout_cone.find(sig_name);
      if( str_set_it == fanout_cone.end() ) { // not found in
fanout cone
  is_FF_sig[i] = true;
      }
    }

   }
```

Code 5: Automated driver tracing

4.3 Use Case and Results.

In this section, we highlight an use case done with IMG Video decoder design for an X issue on which this app was tested and initially prototyped on. Below table captures X signals before and after each stage. It also captures time taken by each stage and total time taken by App to output reduced signal list. Overall X signal count was less than 30% of total X signal count from Stage 1.

	Total X signal count	Reduced signal count	Time Taken
Stage 1 (4.2.1)	109137	109137	3 sec
Stage 2 (4.2.2)	109137	55805	3 sec
Stage 3 (4.2.3)	55805	55149	4 sec
Stage 4 (4.2.4)	55149	32936	13 min
		Total Time	13 min 10 sec

Table 2 Results

5. Conclusions and next steps

X debug can be tedious and time-consuming at RTL and Gate-level simulation. X-propagation techniques such as VCS Xprop can help increase effectiveness of RTL simulation by avoiding X-optimism. Gate-level simulation to debug X can be reduced. However, debugging all of the multiple consequences of a X root cause can still be very time consuming.

This paper describes an algorithm to reduce debug effort and was implemented with a Verdi App. We believe this Verdi App can be optimized further and put across as an application that can help reduce debug time for a failing VCS X-prop simulation.

We also wish to highlight that same App can be extended for Gate Level simulation failure as well by reading synthesis information. This is however outside the scope of paper.

6. References

[1] 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.
[2] Adrian Evans, Julius Yam, and Craig Forward. X-propagation: An alternative to gate level simu-lation. In Synopsys Users Group Conference, San Jose, 2012.
[3] Lisa Piper and Vishnu Vimjam. X-propagation woes: masking bugs at rtl and unnecessary debug at the netlist. In Design and Veri_cation Conference, San Jose, 2012.
[4] Stuart Sutherland. I'm still in love with my x! In Design and Veri_cation Conference, San Jose, 2013.
[5] Synopsys. VCS, 2015. URL http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx. [Online; accessed 09-May-2015].
[6] Synopsys. Verdi automated debug system, 2015. URL ://www.synopsys.com/Tools/Verification/debug/Pages/Verdi-ds.aspx. [Online; accessed 09-May-2015].
[7] Mike Turpin. The dangers of living with an x. In Synopsys Users Group Conference (SNUG), Boston, 2003.
[8] Mike Turpin. Solving verilog x-issues by sequentially comparing a design with itself. In Synopsys Users Group Conference (SNUG), Boston, 2005.
[9] Synopsys Verdi Apps https://www.vc-apps.org/gettingstarted.php
[10] Challenges of VHDL X Propagation DVCon_Europe_2015_TA5_2_Paper.pdf