X-Propagation Woes: Masking Bugs at RTL and Unnecessary Debug at the Netlist

Lisa Piper, Vishnu Vimjam
Real Intent, Inc.
Sunnyvale, CA USA
lisa@realintent.com, vishnu@realintent.com

Abstract::* This paper presents a complete and practical methodology to comprehensively solve the X problem in RTL design. It begins by reviewing common sources of Xs, and describes how they cause functional bugs as well as unwarranted debug that prolong verification cycles. Solving the X problem helps minimize simulation and synthesis iterations and enables various design analyses (e.g. power analysis), normally performed on netlists, to begin sooner. The pros and cons of various point solutions to this problem are described. The technologies discussed include structural analysis, formal analysis, coding for X-accuracy, and simulation techniques such as random seeding of state initial values. It is essential that a complete solution address both X-optimism and X-pessimism woes as well as be applicable to all sources of Xs, facilitate debug, provide coverage analysis, and enable automation, high performance, and usability. The requirements of a complete and practical solution. based on feedback from users who deal with X issues are provided. The summary of our interaction with users is that the X problem is multi-dimensional and needs a holistic solution that brings to bear the combination of strucural analysis, simulation and formal analysis to solve effectively. We describe our user experiences and a case study based on our proposed solution.

Keywords-component; X-propagation, X-optimism, X-pessimism

I. INTRODUCTION

SOC integration levels approaching a billion transistors per chip, tremendous pressure to shrink the verification cycle, and the power minimization imperative have compounded an existing issue where Xs in simulation can mask functional bugs and cause unnecessary Xs in netlists.

This paper begins with an introduction to the issues caused by X-propagation, including defining the X state, common sources of Xs, and a description of the issues of X-optimism and X-pessimism. We then discuss the pros and cons of available solutions for preventing the masking of bugs by X-optimism in RTL and the unnecessary debug caused by X-pessimism on netlists. Feedback from customer interactions is discussed and a complete and practical solution is proposed.

The SystemVerilog standard defines an X as an “unknown” value which is used to represent when simulation cannot definitely resolve a signal to a “1”, a “0”, or a “Z”. Synthesis, on the other hand, defines an X as a “don’t care”, enabling greater flexibility and optimization. Unfortunately, Verilog RTL simulation semantics often mask propagation of an unknown value by converting the unknown to a known, while gate-level simulations show additional Xs that will not exist in real hardware. The result is that bugs get masked in RTL simulation, and while they show up at the gate level, time consuming iterations between simulation and synthesis are required to debug and resolve them. Resolving differences between gate and RTL simulation results is painful because synthesized logic is less familiar to the user, and Xs make correlation between the two harder. Unwarranted X-propagation thus proves costly, causes painful debug, and sometimes allows functional bugs to slip through to silicon.

Continued increases in SOC integration and the interaction of blocks in various states of power management are exacerbating the X problem. In simulation, the X value is assigned to all memory elements by default. While hardware resets can be used to initialize registers to known values, resetting every flop or latch is not practical because of routing overhead. For synchronous resets, synthesis tools typically club these with data-path signals, thereby losing the distinction between X-free logic and X-prone logic. This in turn causes unwarranted X-propagation during the reset simulation phase. State-of-the-art low power designs have additional sources of Xs with the additional complexity that they manifest dynamically rather than only during chip power up.

II. UNDERSTANDING THE ISSUES

In this section, we review the current understanding of the X-optimism and X-pessimism issues that result from the presence of Xs in a design. Note that the same issues can occur with Z values, but Z’s tend to be limited in scope and very intentional. First, we review what Xs are and where they come from.

A. Definition of an X

Xs do not exist in real hardware. X is an abstract value introduced for the sake of algebraic semantics, and different tools interpret them differently.

Simulation semantics of an X are defined by the IEEE 1800 SystemVerilog standard[1]. Simulators interpret an X as a value in 4-state logic (`0, 1, X, Z`) that represents an “unknown” logic value. There are four data types defined in the standard that use 4-state logic: `logic, reg, integer,` and `time`. All of these data types have a default value of X.

Synthesis tools treat Xs differently. They interpret the X value as a “don’t care” instead of an “unknown”, allowing for greater synthesis optimizations.

B. Common Sources of Xs in a Design

There are many sources of an “unknown” (X-source) in simulation. Some are by design, some are a result of errors, and some are the result of X’s driving the inputs to the design.

The most common X source are uninitialized flops and latches, many of which come from memories in the design. While some designers use the practice of initializing everything, many applications don’t have that luxury due to real estate and routing constraints of a reset signal.

Blocks that support low power can also have X-sources when power is off or being restored. When a block comes out of a low power mode, depending on the wake-up mechanism, initialization is generally effected more by retention flops and isolation cell strategies added to gate implementations rather than via the traditional reset signals. Low power brings with it a separate type of initialization analysis.

Another common source of Xs is explicit assignment. Explicit assignments are used to flag illegal or unexpected conditions. Other common error conditions that cause an X value in simulation include bus contention, range violations, a memory-read before initialization, and signals that are not driven.

All of these can result in “unknown” values that will be either a logic-1 or a logic-0 in real hardware, but cannot be determined at simulation time.

C. X-optimism Masks Functional Bugs

X-optimism is an RTL phenomenon where what should be an unknown value becomes a known value in simulation. X-optimism is common in RTL `if` and `case` constructs. A value of Z can also cause optimism to occur, but Zs in a design are relatively uncommon.

To understand how an unknown value can suddenly become a known value, let’s analyze what happens with the simple `if-else` statement that is shown below in Figure 1.

```
//if-else example
logic s;
always_comb
   begin
     if (sel)
        out = a;
     else
        out = b;
   end
```

input *sel*	*output* *out*
1	a
0	b
X	b

Figure 1. X-optimistic If-Else Statement

When `sel` is a `1’b1`, the output is the value of `a` and when `sel` is `1’b0`, the output is the value of `b`. But notice what happens when `sel` is an X. Here, the X value is interpreted as though `sel` is `1’b0` and the output is the value of `b.` The “unknown X” is now masquerading as a known value. In real hardware the `sel` signal might in fact have been a `1’b1`, which means that the correct value could have been the value of `a`. Similarly, as shown in Figure 2, a `case` statement can also be prone to X-optimism. When `sel` is `1’b1`, the output is `a`, when `sel` is `1’b0`, the output is `b`, and when `sel` is an unknown X, simulation semantics define that the output retains its previous value. In real hardware, it may or may not change value depending on the actual value of the `sel.`

```
//if-else example
reg s;
always_comb
   begin
     case (sel)
       1: out = a;
       0: out = b;
     endcase
   end
```

input *sel*	*output* *out*
1	a
0	b
X	prev

Figure 2. X-optimistic Case Statement

X-optimism is a serious issue because it can mask functional bugs. On the other hand, it can occur frequently to ill effect since it only becomes a problem when the output is being used. An example might be Xs on a shared address bus that is not driven until a device acquires the bus. These Xs are only an issue when the data is being sampled.

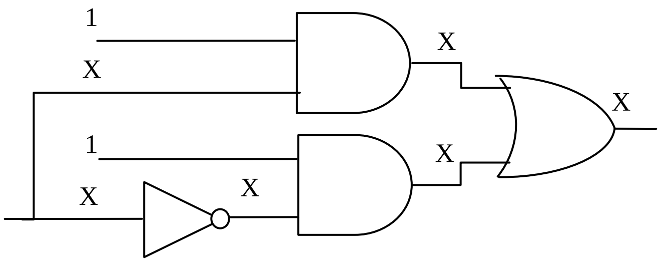
In summary, a situation is said to be X-optimistic when an unknown value incorrectly becomes a known value, thereby potentially masking functional bugs in RTL. X-optimism issues will be revealed in gate level simulations, however gate level simulation is slow, iterations back to RTL to fix the bug are costly, and debug is painful. The biggest challenge with X-optimism is identifying when it causes observable functional differences, instead of simply identifying when it occurs.

D. X-pessimism Causes Unnecessary Xs

X-pessimism is the situation when a deterministic value in hardware becomes an unknown value in simulation. It is an issue primarily observed at the netlist level, though it can be sometimes visible in RTL simulations as well. It occurs when there is reconvergence and loss of correlation between converging X values. It is common in multiplexors and decoders.

Let’s look at a simple multiplexor. In real hardware, we know that when both inputs are the same value, the output value matches that value, independent of the value of the selector. But look what happens in gate level simulation when both inputs are a `1’b1` and the selector is an X value.

In general, a two input multiplexor with a selector is implemented as `(a && selector) || (b && ~selector)`. Figure 3 shows the gate level implementation of a multiplexor. When the selector is `1’bx` and `a` and `b` are both `1’b1`, the logic reduces to `“1’bx || ~1’bx”,` which simulators resolve to an X value per the standard.



```
assign  out = (X && 1) || (~X && 1)
assign  out = X || ~X
assign  out = X
```

Figure 3. Two Input Mux Example of Pessimism

In summary, X-pessimism occurs when an X value appears unnecessarily. X pessimism results in differences between RTL and gate level simulations that must be resolved.

III. SURVEY OF APPLICABLE TECHNOLOGIES

This section reviews known technologies for detecting and preventing X-propagation issues.

A. Simulation Analysis

Simulation is when input sequences are applied to a model and the output is checked. All simulators provide differentiation through features, some of which are associated with X propagation issues.

1) Waveform Tools

Waveform tools[9] provide various features to help detect and debug X-propagation issues in simulation. For example, most waveform viewers show an X value as a distinct state that is both a `1'b1` and `1'b0` in the color red. This is useful in visualizing X values. Most viewers will also allow tracing the drivers of an X.

Unfortunately, identifying true X-optimism issues in this manner is cumbersome. Let's say an X was detected in netlist simulation. First, one must determine if this is a legitimate X or an artifact of X-pessimism. Then, one must manually separate legitimate X propagation from X-optimism. Finally, the process is further complicated because there may be much logic and many clock cycles between an X-optimism occurrence and its manifestation as a failure.

2) RTL X-optimism Detection and Reporting

Some simulators have a compile time switch for detecting X-optimism[4]. It will drive the control of X-optimistic constructs to a `1'b1` and a `1'b0` when the control is a `1'bx` value, and flag an error when the output is different in the two cases.

This approach is clearly has an overhead in simulation time but does enable detecting X-optimism occurrences as they happen. The problem, once again, is that X-optimism often exists in designs without affecting functionality. So, the additional simulation time could end up being wasted and lead to further wasted effort debugging false negatives.

3) Randomization of Initial Values

Some simulators have a feature that will allow you to set the initial state of all `reg`'s in a design to 1'b1, 1'b0, or a random value. You can run multiple times to cover multiple scenarios.

This can be a good solution for addressing X-pessimism issues, but not for detecting correct initialization and X-optimism issues. Even with multiple runs, you are still likely to miss a functional bug if the simulator doesn't happen to exercise the combination of values that triggers an issue. It also only addresses X's from uninitialized `regs`, when, in fact, there are many other sources of X in a design.

B. Structural Analysis

Structural analysis analyzes the characteristics of the RTL to identify potential issues. Lint tools such as Ascent Lint[10] can do a structural analysis of the RTL and alert the designer to X-hazards that might cause X-propagation issues. Specifically they might identify explicit X-assignments, signals within the block that are used but not driven, or signals that are used before being assigned. All such things may cause X-propagation issues. Unfortunately, a structural analysis tool cannot determine if a real issue exists and does not include any sequential analysis.

C. Formal Analysis

Formal analysis is a systematic and automated way of exhaustively testing a design for a specific issue at hand. This section discusses the applicability of the three types that come up in discussions of X-propagation.

1) Equivalency Checking

It is a misconception of many that equivalency checkers catch X-propagation issues. Equivalency checking deals with formally verifying two versions of a design to be functionally equivalent. Practical equivalence checkers analyze combinational logic cones bordered by state elements, ports, or black boxes [5]. Equivalence is based on binary values on the bordering and internal signals.

2) Model Checking

Model checking allows a user to write properties in specification languages such as PSL or SVA [3] so that formal engines may exhaustively verify that those properties hold for all combinations of input sequences. Formal tools that support 4-state analysis will typically verify that properties hold for all combinations of 1'b1 and 1'b0 on all X sources. While this can mitigate the X-optimism and X-pessimism issues, a problem with model checking is that the user must develop these properties, including properties that constrain the inputs to valid sequences. There are also automatic model checkers that automatically extract specialized checks like the reachability of an X-assignment statement or bus float/contention from the RTL and apply sequential formal analysis for checking. Unfortunately, they do not solve the X-propagation problem in entirety. Model checking also has capacity challenges[7].

3)Symbolic Simulation

With symbolic simulation, inputs are parameterized by Boolean variables, and outputs are represented as a function of those variables[6]. Symbolic simulation can exhaustively verify the design by checking that the output remains the same regardless of the presence of Xs. It can also provide a counter example that shows the sequence of inputs required to trigger the error condition.

Unfortunately, there are practical limitations to symbolic simulation. Symbolic simulation techniques explode in memory consumption rapidly with design size and function. Like all formal tools, one can get false negatives in symbolic simulation due to inclusion of illegal sequences of inputs. Another limitation of both conventional and symbolic simulation is that they can only verify over a specified number of cycles [7]

D. X-Accurate Coding

One approach for managing X-propagation is through strict coding guidelines to ensure X-accurate coding. The conditional operator (? :)[1] is immune to X-optimism and can be used in place of `if-else` and `case` constructs, but it is not as readable and is still prone to X-pessimism. Figure 4 contrasts a simple example of common coding, which is X-optimistic, X-pessimistic coding and X-accurate coding.

The X-optimistic coding is so because when the value of `sel` is unknown, the output is assigned the value of `2’b01`, as though `sel` was a `1’b1`. In real hardware, it could have been either a `1’b1` or a `1’b0,` so simulation results could mask an issue.

The X-pessimistic coding will use the case equality operator (===)[2] to first check if the value of the control signal (`sel`) is a `1’bx`. If so it will propagate that X value by assigning all bits of the output to `1’bx`. This will prevent the optimism, however it is pessimistic since the MSB of the output would have been a `1’b0` independent of the value of `sel`. Pessimistic coding will cause unnecessary debug.

Ideally, the coding should be X-accurate, meaning that it should correct for X-optimism by propagating the X value, but without introducing X-pessimism. The X-accurate code for the example is also shown, where each bit of the output is evaluated independently.

X-optimistic Coding

```
always @(*)
   if (sel==1’b0)
        g = 2’b00;
  else
        g = 2’b01;
```

X-pessimistic Coding

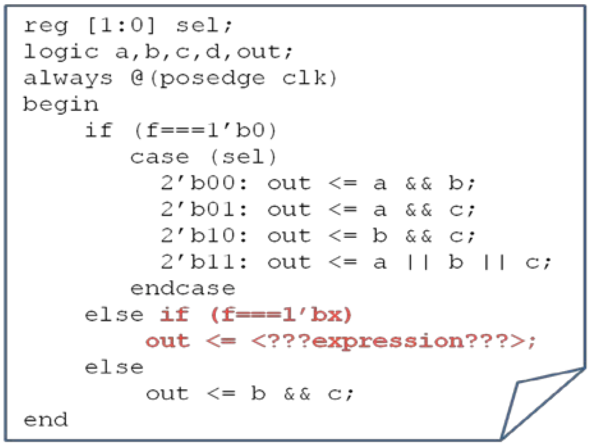
```
always @(*)
   if (sel==1’b0)
        g = 2’b00;
   else if (sel===1’bx)
        g = 2’bxx;
  else
        g = 2’b01;
```

X-accurate Coding

```
always @(*)
   if (sel==1’b0)
        g = 2’b00;
   else if (sel===1’bx)
        g = 2’b0x;
  else
        g = 2’b01;
```

Figure 4. X-accurate Coding

The example above makes X-accurate coding look easy, however it can become complex very quickly as the number of paths in the control-flow graph increases. A more realistic but still simple example is shown in Figure 5, where there are embedded `if` and `case` statements, and the assignment values are expressions rather than just constants. It shows how it is non-trivial and becomes error prone to manually do X-accurate coding. Automation is needed.



```
reg [1:0] sel;
logic a,b,c,d,out;
always @(posedge clk)
begin
    if (f===1’b0)
       case (sel)
         2’b00: out <= a && b;
         2’b01: out <= a && c;
         2’b10: out <= b && c;
         2’b11: out <= a || b || c;
       endcase
    else if (f===1’bx)
        out <= <???expression???>;
    else
        out <= b && c;
end
```

Figure 5. Non-obvious X-Accurate coding

IV. CASE STUDIES DRIVE REQUIREMENTS

Real Intent has been working to address X-propagation issues for several years now. There have been several customer studies to help refine the requirements of a complete and practical solution. We have worked with both designers and

---

[1] The conditional operator returns the value of the first expression when the condition is true and the second condition when the expression is false. When the condition is ambiguous (x or z), then both both expressions are evaluated. An x value is returned unless both expressions return the same 0 or 1 value, in which case that value is returned.

[2] With the case equality operator, bits that are x or z shall be included in the comparison and shall match for the result to be considered equal.

verification engineers mired in X's to propose a practical and complete flow that combines methodology and technology as appropriate.

A. *Customer Profile #1*

This interaction was with a verification engineer to be able to detect real X-optimism issues in RTL, before synthesis. It was not a goal to eliminate all X's in the design, or even all X-optimism occurrences. They only want to detect and correct the ones likely to cause functional differences. In addition, they want to use their existing infrastructure without having to develop additional code or assertions.

X-optimism issues will show up at the netlist level, but it is not practical to debug them there. Netlist issues are more difficult to debug because the design is less familiar after synthesis, there are more X's due to X-pessimism, and simulations are significantly slower. This customer was keen to find X-optimism issues in RTL.

Once a functional issue is detected, the solution needs to provide the information necessary to help isolate the root cause of the problem Debug is a challenge, even at RTL, because the functional anomaly is generally detected at the outputs of a block or chip, which can be many clock cycles after the problem originates. And there can be many X's in the design, most of which are innocuous. Finding the X causing the problem can be like finding a needle in a haystack.

B. *Customer Profile #2*

This customer was a design engineer. X's were viewed as common in the design. This design engineer wanted to understand where the X-sources are in the design, and whether they might propagate to X-sensitive constructs. They did not want any automatic recoding. They wanted the solution to provide information to enable effective manual decisions.

For this customer the requirement was to identify all X-sources. The challenge is precision because noisy reports will cause unnecessary analysis. For example, the list of X-sources should not include X-assignments that cannot be reached. Also, the tool should not identify X-sensitive constructs that were already coded for X-accuracy.

C. *Customer Profile 3*

The third customer we interacted with was primarily concerned about X's originating from powered down blocks. Their chips consist of many power domains. At any time, a block can be in one of full power, reduced power, and powered down states. Testing all the combinations can be a nightmare. Also, low power features are incorporated at the netlist level, where simulations are difficult to debug and very slow.

Powered down block must be restored to full operation very quickly for the power management scheme to be effective. This is done by using retention cells to hold states and isolation cells to ensure constant known values on inputs and outputs of the power domain. Retention cells and isolation cells enable bringing up a block after dynamic power down to be much faster than when power is first applied. These specialized cells are expensive and also consume additional power. Minimizing their count is essential to these power management schemes. This was a primary motivation of this customer for incorporating an X-analysis solution.

From the perspective of X-propagation analysis, the complete solution must be able to consider X's that result when bringing up a block that is powered down. The analysis is similar to determining uninitialized flops after the reset sequence, but requires understanding retention cells and isolation cells that are specified, not in RTL, but in UPF or CPF side files. Another requirement is that you do not want to flag issues during the low power state.

D. *Customer Profile #4*

A fourth customer was another verification engineer that was more concerned with X-pessimism than X-optimism. X-pessimism causes unnecessary X's in gate level logic. Getting the simulations running often requires analyzing where the pessimism is occurring and fixing each case, being careful not to overlook what might have been X-optimism. They stated that they did not like randomization of initial values because it is arbitrary and might not exercise a problem area. The goal was to be able to automatically identify when and where X-pessimism was occurring. Automatically correcting X-pessimism would be even better. This would allow other downstream processes, like power analysis to begin earlier.

In summary, a complete solution must address X-optimism, X-pessimism, X-source analysis, information reporting and debug. Designers typically want to know where issues might be through early static analysis. Verification engineers want to build on the infrastructure they already have to detect and isolate real issues. As much as possible should be done at RTL where debug is easier and simulations are faster.

V. A PRACTICAL X-VERIFICATION SOLUTION

The solution to the X-propagation problem is part technology and part methodology. The proposed solution brings together structural analysis, formal analysis, and simulation in a way that addresses all the problems and can be scaled. Figure 6 shows the use model for the design engineer and the verification engineer.

The solution is static analysis centered for the design engineer and is primarily simulation-based for the verification engineer. Also, the designer centric flow is preventative in nature while the verification flow is intended to identify and debug issues. We will use the I2C testcase from OpenCores to demonstrate the flow.

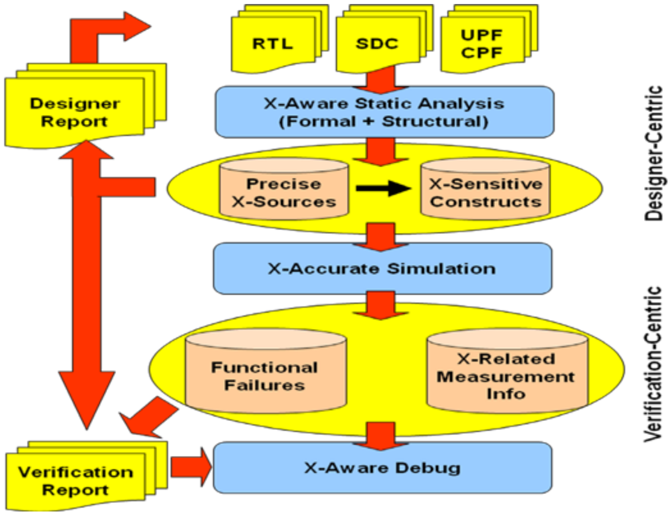


Figure 6. X-Verification Flow

A. *Design Centric Flow*

It was clear from our customer feedback that designers want to understand where in the design X's might originate and what X-sensitive constructs they might propagate to. So automation is needed to determine all the X-sources in the design, trace their propagation through X-sensitive constructs, and present the results in the form of a hazard report.

A precise determination of X-sources is key because an over determination of X-sources would lead to an unnecessary analysis of many X-sensitive constructs, an overly long hazard report, and an overly complicated debug. A combination of simulation, structural analysis, and prudent use of formal techniques can be utilized to determine a precise list of X-sources in the design, and a simple structural analysis can identify the X-sensitive constructs in their paths.

Let's analyze a code snippet from the I2C design, shown in Figure 7. Based on the code, and the assumption that all inputs except the clocks and asynchronous resets may have X's, the designer should be informed that signals `wb_rst_i` and `wb_addr_i` are X-sources because they are inputs to the block, and the signal `cr[7:0]` is sensitive to X's on those signals. In addition, if an X-source can propagate to other signals in that construct, such as `wb_acc` or `i2c_al`, the designer should be informed that the `cr[7:0]` is also sensitive to X's on those signals. This information should be provided in conjunction with a source code browser for tracing paths from X-sources as well as to and from X-sensitive constructs. For each signal that might cause X-optimism to occur, the tool should show how an X-source could propagate to the X-sensitive construct.

```
always@(posedge wb_clk_i or negedge rst_i)
  if (!rst_i)
    cr <= 8'h0;
  else if (wb_rst_i)
    cr <= 8'h0;
  else if (wb_wacc)
    begin
    if (core_en &(wb_adr_i==3'b100) )
       cr <= wb_dat_i;
    end
  else begin
    if (done | i2c_al)
      cr[7:4] <=  4'h0;   //command bits
    cr[2:1] <= #1 2'b0;  // reserved bits
    cr[0] <= #1 1'b0;    // clear IRQ_ACK
  end
```

Figure 7. I2C Code Snippet

For netlists, the report is similar but the tool reports signals that are sensitive to X-pessimism. Ideally a product should predict where pessimism is likely to exist from the RTL, though it will be somewhat dependent on the synthesizer.

B. *Verification Centric Flow*

The verification engineer typically wants to determine if X-optimism is causing functional bugs to be overlooked in RTL simulations, and wants to eliminate X-pessimism in netlist simulations. X-accurate modeling will accomplish both, and the existing simulation checkers can be used to detect functional issues. This can be done in a way that does not touch the user's code and is easily integrated. Using the I2C design with an X-accurate model reveals that an error was being masked by X-optimism. Figure 8 shows the simulation output.

RTL simulation is preferred over netlist simulation because it is faster. It is imperative the performance overhead of X-accurate simulation be controlled. The precise determination of X-sources and the resulting minimization of the number of X-sensitive constructs that must be modeled in RTL simulation is, therefore, an essential part of our solution.

Once an issue is discovered, the verification engineer needs further information to isolate the cause. For this, it is useful to know which signals were sensitive to optimism and which control signals had X's. The suggested methodology is to use the simulator's built in assertion counters to track statistics for these signals. We also configure our monitors to print a message the first time a signal is sensitive to X-optimism. This is useful for determining the root cause. Figure 9 shows the statistics for this run and Figure 10 shows the simulation output with messages. How many messages and the types of messages need to be configurable for debug.

```
/** Running simulation with Real Intent Ascent
XV generated X-Optimism models
************************/
XV Info: X_OPT:out models are enabled at time
0 for instance `i2c_TOP

INFO: WISHBONE MASTER MODEL INSTANTIATED
status: 99500 done reset
status: 109600 programmed registers
status: 113600 verified registers
status: 121600 generate 'start', write cmd 20
status: 11443600 write slave memory address 01
status: 21541600 write data a5
.
.
status: 106054600 received 21 from 3rd read
status: 106057600 read + nack
status: 106063600 received 21 from 4th read
status: 106068600 generate 'start', write cmd
20 (Check invalid address
status: 106076600 write slave memory address
10
status: 106079600 Check for nack

ERROR: Expected NACK, received ACK

status: 106082600 generate 'stop'
status: 131082600 Abbreviated Testbench done
```

Figure 8 - Simulation Output with X-Model

```
Coverage Data from VCS Log Files
(Clocks=344108) Control Input Signal Names
Matches    Signal Name
-------- ----- ------------
16         byte_ctrl_bit_ctrl_al
16         i2c_al
38492      wb_adr_i_2_0


Coverage Data from VCS Log Files
 (Clocks=344108) Signals Sensitive to optimism
Matches     Signal Name
-------- ----- -------------
16         byte_ctrl_bit_ctrl_scl_oen
16         byte_ctrl_bit_ctrl_cmd_ack
16         byte_ctrl_bit_ctrl_c_state_17_0
2          byte_ctrl_c_state_4_0
16         byte_ctrl_shift
10         byte_ctrl_core_txd
2          byte_ctrl_core_cmd_3_0
2          cr_7_4
38490      wb_dat_o_7_0
```

Figure 9. Statistics on X-optimism signals

It is important to acknowledge that all X's are not bad, and that it is not practical to eliminate all X's, so you must be able to detect and isolate real issues. Monitoring is critical for debug, however you can't simply print messages every time it occurs because this type of output will slow the simulator to a crawl (e.g. if it had printed that address and data were X 38490 times, the simulation would take a very long time to run). It is also important to turn monitors off during initialization and when blocks are not fully powered.

```
/****Running simulation with Real Intent Ascent
XV generated X-Optimism monitors ********/
XV Info: X_OPT:out and X_OPT:in monitors are
disabled at time 0 for instance `i2c_TOP
INFO: WISHBONE MASTER MODEL INSTANTIATED
status: 99500 done reset
XV Info: X_OPT:out and X_OPT:in monitors are
enabled at time 100000 for instance `i2c_TOP
status: 109600 programmed registers
status: 113600 verified registers
status: 121600 generate 'start', write cmd 20
status: 11445600 write slave memory address 01
status: 21543600 write data a5
.
.
XV Warn: X_OPT:out signal `i2c_TOP.cr[7:4] was
sensitive to X-Optimism at time 106049000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.core_cmd[3:0] was sensitive to
X-Optimism at time 106049000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.c_state[4:0] was sensitive to
X-Optimism at time 106049000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.bit_ctrl.c_state[17:0] was
sensitive to X-Optimism at time 106049000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.bit_ctrl.cmd_ack was sensitive
to X-Optimism at time 106448000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.bit_ctrl.scl_oen was sensitive
to X-Optimism at time 106448000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.shift was sensitive to X-
Optimism at time 106449000
XV Warn: X_OPT:out signal
`i2c_TOP.byte_ctrl.core_txd was sensitive to X-
Optimism at time 108293000
status: 115432600 received xx from 3rd read
status: 115435600 read + nack
status: 125530600 received xx from 4th read
status: 125535600 generate 'start', write cmd 20
(
status: 136955600 write slave memory address 10
status: 147050600 Check for nack
status: 147053600 generate 'stop'
status: 172053600 Abbreviated Testbench done
```

Figure 10. Simulation with Monitors Identifies X-optimism

Looking at the messages printed, the first X-optimism occurs at time 106049000 on signals `cr[7:0]`, which is the command register – a likely culprit. From this I know to look at the waveforms at this point in time. Figure 11 shows the waveform of the relevant signals without the X-accurate model. Figure 12 shows the same with the X-accurate model. Recall that the source code snippet is shown in Figure 7. `wb_wacc` is 1'b0, so the statement being executed at the specified time is:

```
if (done | i2c_al)
   cr[7:4] <=  4'h0;
```

Since `done` is 1'b0 and `i2c_al` is `1'bx`, the control evaluates to `1'bx`, so `cr[7:4]` retains its previous value of `4'b0010`. If `i2c_al` evaluates to a `1'b1` instead of a `1'b0`, then the assignment would have been `4'b0000`. Notice that bit `cr[5]` was sensitive to optimism. From here you can trace the X value of the control signals back to its source with the waveform analysis tool.

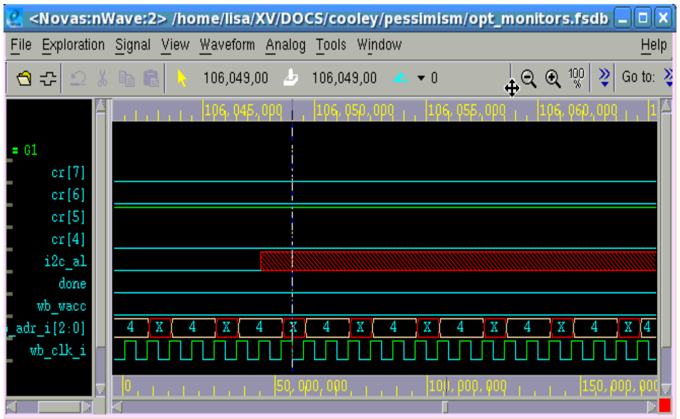


Figure 11. Simulation without X-accurate Model

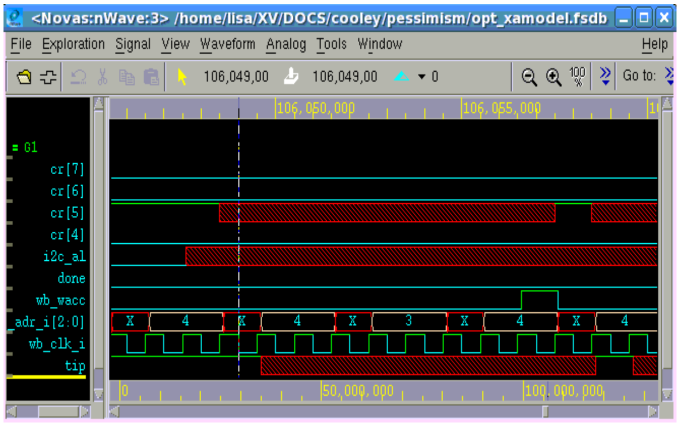


Figure 12. Simulation with X-accurate Model

The suggested methodology is designed to provide both the designer and verification engineer with the tools to prevent X-propagation issues.X-accurate modeling can be used to verify that X-optimism is not masking bugs, and also corrects X-pessimism at the netlist level. Monitors can be enabled for isolation and debug.

VI. SUMMARY

X-propagation issues have always existed but are becoming more prominent as the levels of integration continue to increase and power management schemes become more sophisticated. Higher levels of integration increase the probability of occurrence of Xs and X-related failures, and make detection and debug that much more difficult. Complex power management schemes further increase the potential for the existence of X's. It is therefore becoming a necessity to sign off specifically on X-verification.

This paper has presented a review of the issues caused by X-propagation and surveyed the various point technologies that are being studied to address the problems. Verification objectives gleaned from working with several customers was presented. Designers want to ensure they understand the potential X-sources in their design that might propagate to an X-optimistic construct. The verification engineer wants to be able to verify that X-optimism is not causing functional issues at RTL, and they want to mask pessimism at the netlist level – all of this without significantly impacting RTL simulation speed.

A practical and complete solution was presented that combines several available technologies and addresses both X-optimism and X-pessimism. It provides a static report to the designer for prevention of X-propagation issues, and enables the verification engineer to detect and debug real X-optimism issues and eliminate unnecessary Xs in netlist. The necessary debug information is provided to help determine where the X originated. Precise determination of X-sources and affected X-sensitive constructs is performed with a combination of formal and structural analyses for effective reporting to designers, efficient debug of failures and a minimum overhead on RTL simulation speed. Real Intent's Ascent XV was utilized to demonstrate the key aspects of the flow.

REFERENCES

[1] IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language, IEEE 1800-2009.

[2] Mike Turpin, "The Dangers of Living with an X", SNUG Boston, 2003..

[3] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper c. 2010 SystemVerilog Assertions Handbook, 2nd edition for Dynamic and Formal Verification, ISBN 878-0-9705394-8-7 http://SystemVerilog.us/

[4] Hong-Zu Chou, Kai-Hui Chang, and Sy-Yen Kuo, "Handling Don't-Care Conditions in High-Level Synthesis and Applications for Reducing Initialized Registers, DAC 2009.

[5] Paul Hoxey, Clayton McDonald, and David Guinther. "An introduction to symbolic simulation" EE Times December 19, 2005.

[6] John Harrison, "Formal Verification Methods 2: Symbolic Simulation", Marktoberdorf 2003

[7] Rajeev Ranjan, Yann Antonioli, Alan Hunter, Oleg Petlin. "Formal verification enables safe X handling", December 16, 2008.

[8] Edmund M. Clarke, Orna Grumberg and Doron A. Peled. Model Checking, MIT Press, 1999.

[9] SpringSoft Verdi^{TM} Automated Debug System http://www.springsoft.com/products/verdi-lp?gclid=CLr73oGI86wCFRAq7AodlS0hJg

[10] Real Intent Ascent XV. http://www.realintent.com/real-intent-products/ascent