

The Dangers of Living with an X (bugs hidden in your Verilog)

Version 1.1 (14th October, 2003)

Mike Turpin
Principal Verification Engineer

ARM Ltd., Cambridge, UK
Mike.Turpin@arm.com

ABSTRACT

The semantics of X in Verilog RTL are extremely dangerous as RTL bugs can be masked, allowing RTL simulations to incorrectly pass where netlist simulations can fail. Such *X-bugs* are often missed because formal equivalence checkers are configured to ignore them, which is a particular concern given that equivalence checking is fast replacing netlist simulations. This paper gives examples of such problems in order to raise awareness of X issues in many different parts of the design flow, which are often poorly understood by RTL designers and EDA vendors alike. It gives practical advice on how to overcome X issues in new designs (including good coding styles) and techniques to investigate them in existing designs (including automated formal proofs). New terminology is introduced to differentiate subtle interpretations of X by EDA tools, along with recommendations to avoid problems. In particular, this paper describes how to change the default settings of equivalence checkers to find hidden bugs (that are otherwise far too sneaky to detect). In short, if you are using EDA tools for simulation, code-coverage, synthesis or equivalence checking, you must be aware of the problems and solutions described in this paper.

Table of Contents

1	INTRODUCTION	4
1.1	GOAL: SEMANTICALLY -RIGOROUS IP.....	4
1.2	HOW TO READ THIS PAPER.....	5
1.3	HISTORY OF THIS PAPER.....	5
2	WHAT DOES X MEAN?	5
2.1	SYNTHESIS SEMANTICS: <i>DON'T-CARE</i>	5
2.2	SIMULATION SEMANTICS: <i>UNKNOWN</i>	6
2.3	SIMULATION SEMANTICS: <i>WILDCARD</i> FOR CASEX AND CASEZ.....	7
2.4	EQUIVALENCE CHECKING: <i>2-STATE CONSISTENCY</i> OR STRICT <i>2-STATE EQUALITY</i>	8
2.5	FORMAL PROPERTY CHECKING: <i>2-STATE SEQUENTIAL</i>	9
3	WHY ARE X'S USED?	10
3.1	IMPROVED SYNTHESIS (DON'T-CARES FOR MINIMIZATION).....	10
3.2	IMPROVED VERIFICATION (X-INSERTION AND X-PROPAGATION).....	10
4	WHY ARE X'S DANGEROUS?	10
4.1	BUGS MISSED BY RTL SIMULATIONS.....	10
4.2	BUGS MISSED BY EQUIVALENCE CHECKING.....	14
4.3	MISLEADING CODE COVERAGE.....	19
5	WHY ARE X'S INEFFICIENT?	20
5.1	UNNECESSARY X'S IN NETLIST SIMULATIONS.....	20
5.2	NON-MINIMAL SYNTHESIS OF DON'T-CARES.....	21
5.3	LIMITED SPEEDUP WITH 2-STATE SIMULATION.....	24
5.4	SLOWER FORMAL VERIFICATION	24
6	HOW TO FIND DANGEROUS X'S	25
6.1	ADDITIONAL SIMULATIONS.....	25
6.2	RTL CODE INSPECTION	26
6.3	DETECTING X'S WITH ASSERTIONS.....	26
6.4	CHANGE DEFAULT SETTINGS OF EQUIVALENCE CHECKERS	27
6.5	AUTOMATIC FORMAL PROOFS OF UNREACHABLE (DEADCODE) ASSIGNMENTS.....	27
6.6	INTERACTIVE FORMAL PROPERTY CHECKING.....	28
6.7	NETLIST SIMULATIONS.....	29
7	HOW TO AVOID DANGEROUS X'S	29
7.1	GOOD RTL CODING PRACTICE	29

7.2	REMOVING REACHABLE DON'T-CARE X-ASSIGNMENTS.....	30
7.3	REPLACING X-INSERTION WITH ASSERTIONS	30
7.4	ENABLING X-PROPAGATION	30
7.5	AVOIDING UN-INITIALIZED REGISTERS.....	30
7.6	FUTURE: SYSTEM VERILOG AND VERILOG 2XXX	31
8	CONCLUSIONS	31
9	TOP-TEN RECOMMENDATIONS	32
10	ACKNOWLEDGEMENTS	33
11	REFERENCES	34

1 Introduction

The aim of this paper is to raise awareness of dangerous X issues in Verilog RTL and introduce techniques to analyze otherwise undetected bugs in RTL designs. New terminology is used to distinguish subtle X semantic problems, illustrated with simple examples that accurately reflect our experience of real problems found by ARM.

Some of these issues have already been raised by Bening [Bening 99], Galbi [Galbi 02], Foster [Foster 03] and other references in section 11. What's new in this paper is the revelation that you can actually find most hidden X-bugs with equivalence checkers, providing you configure them to be sensitive to X issues. This paper also describes how to use formal property checking to determine which X's are safe and which are dangerous (and should therefore be removed from your RTL). For X-bugs that cannot be found by formal verification, RTL coding guidelines are given.

Many engineers have polarized views on X's, some seeing them as a good thing (e.g. for synthesis or X-propagation) and others believing them to be an intrinsically bad idea, since they can mask real design bugs, and should never be used. This paper argues against the widespread use of X's, particularly for don't-cares, but also shows how selective use of X's can improve design and verification (e.g. it recommends that every case default assigns X).

Understanding problems caused by X semantics is extremely important. Many designers are blissfully unaware of the issues around X, which can have devastating effects on many different parts of the design flow including:

1. **RTL Simulation:** X semantics in RTL can mask bugs - expensive validation tests can pass because they are not being used effectively to stress the design.
2. **Code Coverage:** X semantics in RTL can give both optimistic results (i.e. claims a branch is covered when it's unreachable) and pessimistic results (i.e. claims a coverage hole when it's reachable)
3. **Equivalence Checking:** all too often, equivalence checkers will miss differences in RTL and netlist simulations caused by subtle X semantics (normally due to incorrect usage, even with default behavior)
4. **Synthesis:** designers often rely on don't-cares to produce efficient logic, but can be disappointed with their non-minimal results and long critical paths

Often due to limited understanding of X issues, bugs can be missed and left in the shipped product (discovery then is far more expensive) or left dormant - only to reappear when a new version of your synthesis tool chooses a different logic minimization!

As verification is 70% of the work, the risk of adding an undetected bug (which could result in a costly silicon re-spin) should outweigh any general improvement to logic minimization. Time saved from verification can be used to help improve any critical paths in the synthesis by careful consideration of the RTL (especially as adding don't-cares will not necessarily improve things as you may expect).

1.1 Goal: Semantically-Rigorous IP

ARM is keen to promote good coding styles and techniques to ensure *semantically rigorous* designs i.e. identical semantics throughout the design flow. ARM has recently undertaken a lot of work in this area, including extensive use of formal verification techniques, to avoid such problems on the latest ARM1136J(F)-STM core. ARM's IP is integrated into many different designs so the impact of our RTL quality is magnified many times – as highlighted by a quote from Harry Foster [Foster 03]:

“What about restricting the designer's coding style to ensure semantic consistency? Although this would be the ideal situation, ... many of today's designs involve integrating other organizations' intellectual property (IP) cores, as well as existing legacy code. ... All it takes to ruin your semantically consistent coded RTL is for someone else's IP to drive an X value into your pure RTL.”

This paper has therefore been written to promote awareness of X-issues both internally at ARM and externally (for designers and EDA vendors). It gives a detailed account of many lessons learned, and provides guidance on how to design semantically rigorous RTL. Simple examples are given to illustrate sometimes complex problems, but each is

based on our experience of real problems. This paper contains recommended practice that will be new to most engineers, e.g. how many people change the default X-settings of formal equivalence checkers to avoid missing bugs?

1.2 How to Read this Paper

As this paper covers a lot of material, some readers may prefer to start by reading the ten recommendations in section 9. If you're already following these then you're done and your RTL should be tolerant of X's. However, most RTL designs contain reachable X-assignments and so the majority of readers will have an incentive to read the other sections too (at least as reference material). The formal techniques described in section 6.5 will be new to most readers, but these should not be daunting in any way (if you can run an RTL linter you can run these automatic proofs). There is some repetition in the paper, so that each section can be read in isolation as reference material.

1.3 History of this Paper

Version 1.0 (12th August 2003). Technical Committee Award winning paper at Boston SNUG (8th September 2003).

Version 1.1 (14th October 2003). Added section 4.1.1 (Latch Behavior in RTL Simulation), improved RTL coding rules in section 7.1, and made some corrections and improvements from feedback at SNUG.

2 What Does X Mean?

This section explains the Verilog semantics of X for different parts of the design flow. The most worrying thing is that there are different semantics!

Engineers are taught about two X-semantics, *don't-care* for synthesis and *unknown* for simulation. The overloading of X in Verilog to cover different concepts often causes misunderstandings and can lead to hidden bugs in your RTL, as detailed in section 4. To help understand subtle X-semantic issues, this paper uses the terminology below to compare and contrast different semantics of X.

1. **Don't-Care:** Synthesis semantics (not important if assignment is 0 or 1, allowing better minimization)
2. **Unknown:** Simulation semantics (not known if value is 0 or 1, but evaluation will take just one path)
3. **Wildcard:** Simulation semantics (specific to `casex` and `casez` statements in Verilog RTL)
4. **2-State Consistency/Equality:** Equivalence Checking semantics (try *both* X=0 and X=1 paths for *every* X)
5. **2-State Sequential:** Property Checking semantics (try *both* X=0 and X=1 paths and *sequences* for *every* X)

Formal tools will try and fail the design by stressing *every* X with both 0 and 1, so a pass is an *exhaustive* verification of all possible 2-state (i.e. 0 or 1) settings of X's (see Figure 2 for an illustration of this). Equivalence checking is a purely *combinatorial* verification of two designs, whereas property checking is a *sequential* verification of one design (e.g. it can consider the effects of an X stored in a register).

Equivalence checkers compare two designs, and can be set up to treat X's differently in each design. There are two sensible configurations, which gives rise to the *Consistency* and *Equality* terminology (see section 2.4). This terminology is similar to that used by the Synopsys Formality equivalence-checker (e.g. see "Design Consistency" in the Formality User Guide [Synopsys 02]), but is applicable to other equivalence checkers.

2.1 Synthesis Semantics: *Don't-Care*

The simplest semantics for X is treating it as a don't-care assignment, allowing synthesis to choose a 0 or 1 to improve the logic minimization. Note that a single X-assignment (e.g. `default` of a `case`) can in fact represent multiple don't-cares, each of which can be assigned to different values.

Synthesis tools tend to have two separate phases:

1. **Minimization:** producing minimal Boolean expressions for logic functions (using any don't-cares)

2. **Optimization:** mapping the minimized logic to a target synthesis library (taking account of signal timings)

Minimization often produces a sum-of-products form which tends to get optimized using NAND-NAND logic (rather than directly using AND gates driving an OR gate). An important point to note is that all don't-care X's are resolved during minimization (so different target libraries will not change this mapping).

2.2 Simulation Semantics: *Unknown*

In Verilog, X is modeled as one of the four possible logic values (along with 0, 1 and Z). It's important to understand that X is just a separate enumeration that has no direct relationship with 0 and 1. The semantics of X are fixed and can only crudely consider the range of possible 0/1 combinations.

The table below summarizes the interpretation of X by some Verilog operators (a full description can be found in the Verilog LRM [IEEE 95]). Boolean operators and the *ternary* (i.e. conditional ?) operator will, individually, propagate X's in a sensible way. However, it's interesting to note that the `if` statement will optimistically consider X as if it were 0 (this causes many X problems in Verilog RTL).

Unary NOT		AND		OR		Exclusive-OR		Exclusive-NOR		CONDITIONAL (if, ?)	
~X	X	(X & X)	X	(X X)	X	(X ^ X)	X	(X ~^ X)	X	f = X ? 00 : 01;	0X
		(0 & X)	0	(0 X)	X	(0 ^ X)	X	(0 ~^ X)	X	if (X) f=00; else f=01;	01
		(1 & X)	X	(1 X)	1	(1 ^ X)	X	(1 ~^ X)	X		

Table 1: Interpretation of X by Verilog Operators

The modeling of X as a separate enumeration will inevitably mean the loss of information. Boolean operators will carefully propagate X's but information can be lost when they are combined, e.g. consider the following example.

```
assign b = a & ~a;
```

Verilog Snippet 1 – Example of X-Pessimism (loss of context information)

From a Boolean algebra viewpoint it's obvious that `b` will always be zero. However, an X on `a` will also cause `b` to become X in a Verilog simulation (but in a real circuit, `b` would still be 1'b0). The unary negation operator propagates the X, throwing away information that the new result should be a symbolic “~X”.

The *Unknown* simulation semantics of X leads to two unwanted effects:

1. **X-Pessimism:** ambiguous results lead to more X-assignments than are really necessary
2. **X-Optimism** interpretation of X will take just one `if/case` branch when many should be considered

Good examples of problems caused by these two effects are described by Lionel Bening [Bening 99]. We have already seen one example of X-Pessimism in Verilog Snippet 1, now for a common example of X-Optimism

```
always @ (CLK or nRESET)
  if (nRESET == 0)
    Count <= 3'b000;
  else if (CountEnable)
    Count <= NxtCount; // no update if CountEnable==1'bX
```

Verilog Snippet 2 – Example of X-Optimism

The above example shows a common way to allow synthesis to achieve clock-gating i.e. whenever `CountEnable` is low the 3-bit `Count` register does not need to be clocked and power can be saved. A problem occurs if any reachable don't-care X's have been assigned to `CountEnable`, to help minimize its driving logic. These don't-cares may improve synthesis but what happens in RTL simulation when there's an X on `CountEnable`? According to the Verilog LRM [IEEE 95] the second branch is only executed if `CountEnable` is 1'b1, so no update occurs when `CountEnable` is X. Effectively, an RTL simulator is only considering one possible interpretation of the X on

CountEnable (if CountEnable is in fact minimized to 1'b1 here, the netlist simulation would give a different result). X-Optimism can also occur in a case default that terminates X's with a 2-state (i.e. 0 or 1) assignment.

The ideal simulation semantics for X would be: “*can be either 0 or 1*”. An ideal simulator would then have to consider the effects of both possible settings for every X-assignment (the downside is that simulations could dramatically slow down as a result). Such a simulator would not be a Verilog simulator, as its semantics would differ to the Verilog standard. However, this ideal X semantics is used by formal verification tools (as discussed in sections 2.4 and 2.5).

2.3 Simulation Semantics: Wildcard for casex and casez

The use of X (and Z) in casex, and Z in casez, is described in section 9.5.1 (page 110) of the Verilog LRM [IEEE 95] as: “*don't-care conditions in case comparisons*”. The Wildcard terminology is arguably better because:

- **don't-cares are normally associated with X-assignments (rather than comparisons)**
- **unlike don't-care assignments, casex expression X's are ignored by synthesis and equivalence checking**

Before talking about casex and casez, it's worth reviewing the semantics of the less exotic case statement. Consider the example below of a case statement that has an X in one case-item and an equivalent nested if expression (in comments).

```
case (sel)
  1'bX:    f = 2'bXX;    // if (sel === 1'bX) f = 2'bXX;
  1'b1:    f = 2'b10;    // else if (sel === 1'b1) f = 2'b10;
  1'b0:    f = 2'b01;    // else if (sel === 1'b0) f = 2'b01;
  default: f = 2'bZZ;    // else f = 2'bZZ;
endcase
```

Verilog Snippet 3 - Example case-statement and equivalent if-expression (in comments)

Important things to note about case statements are listed below (which may help to dispel a few myths):

- RTL simulation considers every case-item above, but synthesis only considers two (highlighted in bold)
- a Verilog case statement is priority encoded (just like a nested if expression)
- the case-expression is effectively compared to the case-item with a triple-equal (===) case-equality
- case-items in a standard case statement can contain X's (designers often forget this)
- the default in the example above will only be hit when sel is Z (and propagates the Z onto the output)
- normally (when case-items don't have explicit X's) an incoming X on a case-expression will hit the default in RTL simulation but netlist simulation may be different

In an RTL simulation, whenever the 1'bX case-item in Verilog Snippet 3 is hit there will be an X propagated onto the output. However, synthesis (and equivalence checking) will ignore this case-item – a semantic inconsistency. The *Semantic Overlap* between simulation and synthesis is highlighted in bold – for both the case statement and the nested if. Now consider the same example, but with case replaced by casex:

```
casex (sel)
  1'bX:    f = 2'bXX;    // if (sel === sel) f = 2'bXX;
  1'b1:    f = 2'b10;    // ...
  1'b0:    f = 2'b01;
  default: f = 2'bZZ;
endcase
```

Verilog Snippet 4 – Replacing case with casex (wildcard reduces semantic overlap)

The `casex` semantics say that the `1'bX` above will be treated as a wildcard and effectively match anything; consequently all the subsequent case-items are redundant. The equivalent `if` statement tests the null operation: (`sel == sel`), as (`sel == 1'bX`) would have different semantics (explicit test of X, rather than wildcard).

For synthesis, the `casex` statement will simply be interpreted as a single assignment (in the above example a don't-care assignment, so the synthesis tool is free to do anything). This small amount of semantic overlap between simulation and synthesis is again highlighted in bold.

Important things to note about `casex` statements include:

- the wildcard is 2-dimensional (an X can occur on any case-item *and* on the case-expression)
- a wildcard in the `casex` expression can cause simulation/synthesis mismatches
- ordering is very important (if `1'b1` was the first case-item above, the resulting assignment could differ)
- `casex` wildcards are for both X and Z (and `?`, a shorthand for Z)
- `casez` does not allow X as a wildcard (an X in a `casez`-item is the same as for a case-item)

An example problem from using `casex` is described in section 4.2.5.

2.4 Equivalence Checking: 2-State Consistency or strict 2-State Equality

The concepts in this section should be applicable to all formal equivalence checkers, but are based on our experience of using both Verplex Conformal and Synopsys Formality at ARM. These tools have different terminology for the two designs, and for different X-related comparison modes. This paper uses the Synopsys Formality terminology of *reference* and *implementation* designs (termed *golden* and *revised* by Verplex Conformal).

Equivalence checkers compare two designs (reference versus implementation) by matching primary I/O and internal state, then individually comparing combinatorial logic for:

- **Next-States:** cones of logic driving the inputs of registers
- **Outputs:** cones of logic driving the primary outputs

In both cases, the cones inputs are driven by primary inputs or the outputs of internal registers. Equivalence checkers exhaustively compare the two cones by driving all 0/1 combinations onto the inputs. By driving each cone input to both 0 and 1, equivalence checkers indirectly consider X's from primary inputs and X's stored in registers.

Any explicit X assignments in both designs will be kept in the logic cones and the equivalence checker *can* try both X=1 and X=0 in order to fail the comparison. However, an equivalence checker will not necessarily do this – it depends on how it's configured to treat X's in *each* design. There are two sensible modes:

1. **2-State Consistency (same or less X's):** This is the default way that most equivalent checkers work. The Formality User Guide [Synopsys 02] says that “an implementation is *consistent* with a reference design when it is functionally equivalent, given that a don't-care state (X) in the reference design can be represented by either a 0 or 1 state in the implementation”. *2-State Consistency* differs from *Don't-Care* semantics in that the implementation can have the same or less don't-cares as the reference, but not more (note that there is an inherent direction from reference to implementation).
2. **2-State Equality (exactly the same X's):** More stringent test of equality to ensure that the don't-care space is identical. In addition to achieving *2-State Consistency*, Formality requires “the [implementation design] set of don't-care vectors matches that of the reference design set”.

The basic idea is that *2-State Consistency* is sufficient for RTL vs. netlist comparisons and most RTL vs. RTL comparisons because it allows the implementation design to have the same or less don't-cares (in the case of a netlist there will be no don't-cares). If you want exact comparison of X-space, e.g. for RTL Verilog vs. translated RTL VHDL, then you need to run a comparison with *2-State Equality* semantics (normally after running a first pass with the default). However, this basic idea is misleading because:

1. **2-State Equality can miss Sequential differences:** Section 4.2.5 shows why even this strict comparison mode can miss differences when X is stored in a register, due to the combinatorial nature of equivalence.
2. **2-State Consistency can miss RTL vs. Netlist Simulation differences:** Section 4.2.2 shows why this comparison mode is not enough if you have any reachable don't-cares in your RTL.
3. **Don't-Care both sides can introduce RTL bugs:** Section 4.2.1 shows how bugs can creep into RTL when X's are treated as don't-care for both sides of an RTL vs. RTL comparison (the default for some tools).

A better explanation of these modes is given below, which is applicable to all comparisons (not just RTL vs. RTL).

- **2-State Consistency:** Only try setting Implementation X's to find differences (Ref X's are don't-care)
- **2-State Equality:** Try setting X's in *both* Reference and Implementation designs

Notes are given below for setting comparison modes in the two equivalence checkers used at ARM.

2.4.1 Comparison Modes in Synopsys Formality

In Formality, these two comparison modes are in fact termed Design-Consistency and Design-Equality. This paper uses 2-State terminology to highlight the dual X=0 and X=1 verification, although it does not convey its exhaustive nature (i.e. for *every* X). By default, Formality uses 2-State Consistency but you can change this with the command:

```
set verification_passing_mode equality
```

2.4.2 Comparison Modes in Verplex Conformal

In Verplex Conformal, you need to use the “set x conversion” switch correctly for running these modes. This mechanism is very open and flexible, but the default mode is inappropriate for RTL vs. RTL comparisons. Some documentation for this can be found in the Conformal Reference Manual [Verplex 03], but it doesn't explain the implications of each setting or how to use it— which should be as follows:

1. **2-State Consistency:** `set x conversion DC -golden; set x conversion E -revised`
2. **2-State Equality:** `set x conversion E -both`

This will produce *E* points in the comparison, which are pseudo-inputs to drive X to any value. The manual documents that the default setting is in fact:

Default: `set x conversion DC -both`

This does not even perform 2-State Consistency and can be dangerous for RTL vs. RTL comparisons (see section 4.2.1 for details).

2.5 Formal Property Checking: 2-State Sequential

Formal tools do not adhere to the semantics of X in RTL Verilog but will try and fail the verification by stressing the design with X=0 and X=1 settings for *every* X (a much better representation of any possible synthesized design). Unlike equivalence checkers, formal property checkers consider the sequential behavior of a design – allowing them to track possible values of X's through internal registers (termed 2-State Sequential in this paper).

Strict 2-State Sequential can be performed by many formal property checkers including:

- **Averant's Solidify**
- **Jasper-DA's Jasper-Gold**
- **Synopsys' Magellan** (dual semantics - also considers *Unknown* simulation semantics via VCS)
- **Verplex's Black-Tie**
- **0-In's Confirm**

A formal property checker that supports *2-State Sequential* semantics is a powerful analysis tool for finding X-related problems in RTL, particularly if it has such checks in an automated form (see section 6.5).

Note that some property checkers may support *2-State Sequential* but actually use *Unknown* or *Don't-Care* semantics as default, which are faster but could miss X-related bugs. When evaluating different property checkers, you should ensure that they are running the same semantics as this can make a big difference to performance results.

3 Why Are X's Used?

This section describes some of the reasons for using X's in RTL Verilog. Unfortunately, most people that use them don't realize the implications of X-semantics and can be adding bugs that they cannot detect!

3.1 Improved Synthesis (don't-cares for minimization)

X-assignments in RTL are mainly used as don't-cares for improving logic minimization during synthesis (well known techniques for this include K-Map and Espresso, but most synthesis tools use proprietary algorithms). This can work well in reducing the logic, but sometimes doesn't produce the minimal results you would expect – see section 5.2.

When adding a don't-care X-assignment you may unknowingly be adding a bug to your design that you will not be able to detect. You need to ask the question: "is it really a don't-care?". The fact that your simulations are passing can be irrelevant – see section 4.1 for a description of this problem and section 4.2.2 for an explanation of why equivalence checkers normally miss them.

As discussed in section 1, the risk of adding an X outweighs any general improvement to logic minimization. Instead, designers can manually improve critical paths as necessary and only use X's that are proven to be unreachable.

3.2 Improved Verification (X-insertion and X-propagation)

Some designers deliberately insert X's into their RTL to catch exceptional conditions (when they should really be using assertions instead). A common myth is that X's will then stress the design by testing both 0 and 1 values in Verilog simulations – but this is often not the case due to X-optimism (see example in section 4.1.2). Another use of X-insertion is to trap missing assignments in complex state machines (see section 2.1.1 of [Cummings 03]).

X-propagation is then used to try and make the error observable. This technique can be useful but will only work if your Verilog RTL perfectly propagates X's, which is very rare (see section 7.4). Again, assertions can be better suited for this task because they increase the observability of bugs by immediately stopping the simulation and reporting its origin. However, X-propagation can sometimes be useful in RTL simulations (any validation method that detects bugs is welcome).

4 Why Are X's Dangerous?

X-assignments in RTL can cause pessimistic and optimistic results, as already described in section 2.2 (and by Lionel Bening [Bening 99]). X-Pessimism can lead to wasted debugging effort during verification. Worse still, X-Optimism means that bugs can lay dormant in a design (undetected by RTL simulations or equivalence checking).

Other problems arise from different RTL simulation and synthesis semantics, e.g. wildcard in `casex`, which again cannot be found by equivalence checking. Equivalence checkers are very useful tools that can find functional bugs and X-related bugs, but only if they are used with care – and not with the default settings.

This section details the non-intuitive effects of RTL semantics, backed up by simple examples that illustrate real problems found at ARM.

4.1 Bugs Missed by RTL Simulations

Due to subtle X semantics in Verilog, RTL simulations can pass tests that netlist simulations could fail.

Unfortunately, far fewer netlist simulations are performed because:

1. Netlist simulations are much slower than RTL simulations, so are not cost effective for entire regressions.
2. Equivalence checking is more complete and rigorous.

The second assumption is a valid one in general (ARM strongly recommends equivalence checking as a rigorous method of finding differences) but not where X's are concerned. Equivalence checkers can easily miss differences due to optimistic RTL interpretation of X.

4.1.1 Latch Behavior in RTL Simulation

In Figure 1 below, a case statement is used in Verilog RTL to describe a simple AND function. Unsurprisingly, it is synthesized to an AND2 gate. However, the waveform illustrates that RTL simulation differs from netlist simulation.

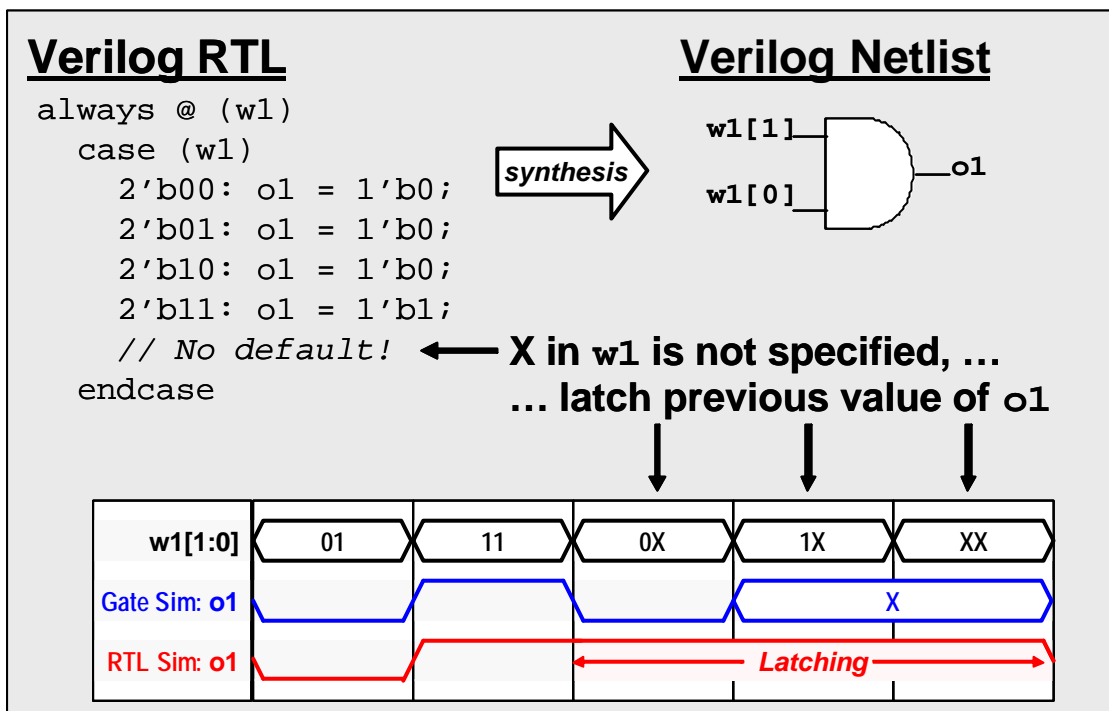


Figure 1: Simulation differences caused by Latch behavior

Simulation differences occur when any X appears on the w1 signal, maybe from an explicit don't-care assignment (alternatively via a primary input or un-initialized register). Any X on w1 will mean that w1 cannot match any of the four case-items. As there's no default line the output o1 must keep its previous value, which in the above waveform means that o1 will stay latched high. The most interesting difference occurs when w1==2'b0X:

- **Netlist simulation correctly shows o1 low**
- **RTL simulation incorrectly keeps o1 high (the opposite value)**

Every netlist will behave like this and differ from RTL simulation, regardless of the minimization of the X on w1. Synthesis tools do not warn of behavioral latches, but such differences between RTL and netlist simulations become dangerous when the RTL verification only passes certain tests because of this interpretation of X.

This example can also be used to compare different semantics of X, as illustrated in Figure 2 below. Formal verification tries *both* 0 and 1 for *every* X and consequently determines that o1 should go low for w1==2'b0X. Hence, formal analysis of the RTL is an accurate model of any synthesized netlist but this is a double-edged sword:

- **Pro:** Formal property checkers *can* find functional bugs hidden by X semantics in RTL simulations
- **Con:** Equivalence checkers *cannot* find the difference in this example, irrespective of configuration

This example is best handled by RTL coding style guidelines for avoiding simulation and synthesis mismatches, as described in sections 7.1 and 7.4.

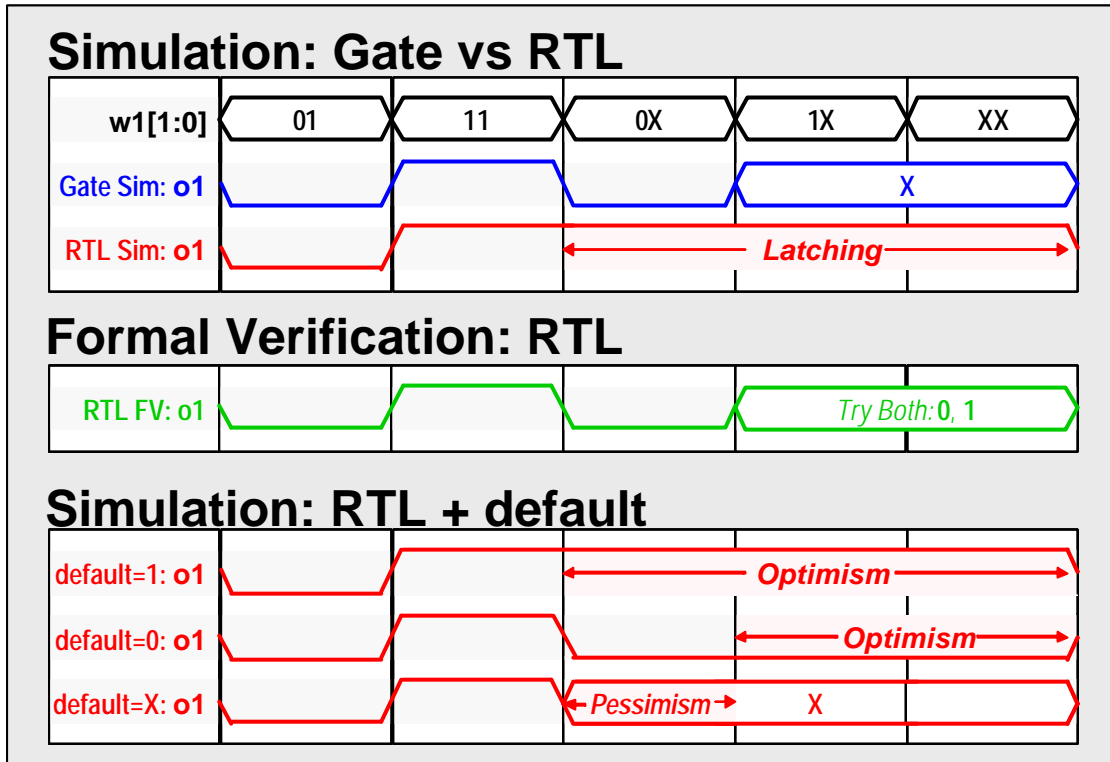


Figure 2: Comparison of X Semantics

The obvious RTL coding guideline to fix this example is to always use a `default` to match incoming X's, but the question is what value the `default` should assign. Figure 2 illustrates the three possible `default` assignments:

- **default: o1=1** Same result as X-latching, with real difference for `w1==2'b0X`
- **default: o1=0** Optimistically keeps o1 low when it could be high in a netlist simulation
- **default: o1=X** Closest match to netlist simulation, with pessimistic X for `w1==2'b0X`

For the most accurate result, the `default` should propagate X's rather than optimistically terminating them.

4.1.2 Optimistic X Semantics in RTL Simulation

A common problem with Verilog RTL is X-Optimism around `if` statements. Consider the following example:

```

always @ (i1 or i2)
  case ({i1, i2})
    2'b00,2'b01: o1 = 1'bX; // don't-care X-assignment
    2'b10:       o1 = 1'b0;
    2'b11:       o1 = 1'b1;
    default:     o1 = 1'bX; // X-propagation
  endcase

always @ (i2 or o1)
  if (o1) o2 = 1'b0;
  else   o2 = i2;

```

Verilog Snippet 5: Example of X-Optimism

In this example, an X on output o1 will not stress both branches of the subsequent if statement that assigns output o2. Instead, the X will only choose the else branch and expose problems when i2 is assigned to o2. What if the don't-care is minimized to 1'b1 – if you run netlist simulations you could fail a previously passing test! In fact, the X-assignment actually represents two separate don't-cares – as illustrated in Table 2 below.

		RTL Simulation	
Input i1	Input i2	Output o1	Output o2
0	0	X _a	0
0	1	X _b	1
1	0	0	0
1	1	1	0

Table 2: RTL Simulation

Consider a simple toggle test on each output, to check that they can go to either 0 or 1. An RTL simulation that sets every possible combination of inputs i1 and i2 will show that output o2 can be toggled. However, the only assignment to 1 on output o2 occurs due to optimistic X interpretation. The output-toggle test passes on RTL but would it also pass on netlist simulation? The two X's in Table 2 are interpreted as don't-cares by synthesis, allowing four possible implementations of output o1 – as illustrated in Table 3 below.

		X _a =0, X _b =0 Cct #1: o1=(i1&i2)		X _a =0, X _b =1 Cct #2: o1=i2		X _a =1, X _b =0 Cct #3: o1=(i1==i2)		X _a =1, X _b =1 Cct #4: o1=(~i1 i2)	
i1	i2	o1	o2	o1	o2	o1	o2	o1	o2
0	0	0	0	0	0	1	0	1	0
0	1	0	1	1	0	0	1	1	0
1	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	1	0	1	0

Table 3: Four Possible Netlist Simulations

Table 3 shows that two of the four possible netlists would actually fail the simple toggle test, with output o2 being a constant 0. This is an example of RTL simulation passing a test that a netlist simulation could fail – depending on the minimization of don't-cares. This shows that even if RTL simulations pass exhaustively, it does not confirm that X assignments are truly don't-care (in this case they are *do-cares* for 50% of all possible netlists).

The recommended comparison mode for equivalence checking RTL vs. netlist is *2-State Consistency* and this will not find this sort of RTL vs. Netlist simulation difference. Section 4.2.2 explains why this RTL will compare as equivalent to the most likely synthesis (missing the simulation differences completely).

Recommendation 1: **Even if all RTL simulations pass, a don't-care should be considered to be a don't-know unless it's proven to be unreachable.**

Section 6.5 shows how to formally verify that don't-cares are unreachable, using automatic proof techniques.

4.2 Bugs Missed by Equivalence Checking

This section describes why equivalence checking often misses differences in both RTL vs. RTL and RTL vs. netlist comparisons. It provides examples that illustrate unsuitable defaults in the tools, inadequate documentation leading to incorrect usage, and fundamental limitations in this technology. Differences are missed due to the use of reachable don't-care X-assignments and X's stored in registers. All of the examples show false positive results – something that's considered an anathema to equivalence checking. As equivalence checking is such a crucial part of today's design-flows, this is a very good argument to avoid such X's.

4.2.1 Equivalence-Check Problem #1: *Don't-Care* both sides can introduce RTL bugs

As described in section 2.4, equivalence checking semantics of X-assignments gives you two choices to try and find differences – *only* try setting X's in the implementation (*2-State Consistency*) or try setting X's in *both* designs (*2-State Equality*).

When the implementation is a netlist there should be no X-assignments, but the direction is very important for RTL vs. RTL comparison. One thing that an equivalence checker should never do is interpret X-assignments in the implementation RTL as *Don't-Care*; otherwise a constant 0 or 1 in the reference RTL could be incorrectly passed as equivalent to an X in the implementation (but what if this is then synthesized to the opposite polarity?). To avoid such bugs creeping into RTL, you must never pass as equivalent any implementation design that has more reachable don't-care X-assignments than the reference design.

The default setting in Verplex Conformal does not have any idea of direction; it treats X's in both sides as don't-cares and so could give a false positive result for RTL vs. RTL. We first saw this problem in an RTL Verilog vs. translated RTL VHDL comparison (being an IP house, this comparison is often done at ARM and we can double check our results with different equivalence checkers).

Verplex Conformal has a very open and flexible mechanism (called "set x conversion") that can be used to explore both *2-State Consistency* and *2-State Equality*. This flexibility allows other modes, including the dangerous configuration that is the current default in Verplex Conformal:

- **Don't-Care in both designs:** set x conversion DC –both

Most users don't change this setting, so the potential is still there for false-positive results with RTL vs. RTL comparisons. This issue has been highlighted to Verplex and is expected to be addressed this in the next version of the tool. It's not a trivial issue, as badly chosen settings can affect performance, but it needs to be addressed to avoid bugs being introduced into RTL.

4.2.2 Equivalence-Check Problem #2: *2-State Consistency* can miss RTL vs. Netlist Simulation differences

The documentation in the Formality User Guide [Synopsys 02] incorrectly states:

"[Equality] is appropriate only when both the implementation and reference designs are RTL designs, because gate-level designs do not have don't-care information."

Intuitively, this seems to make perfect sense but it's insufficient when you have reachable don't-cares in your RTL. This section shows how bugs can be missed with the recommended (*2-State Consistency*) settings but can be found with the stricter *2-State Equality* comparison.

Consider the example in section 4.1.2, which passes a simple RTL toggle test that fails in some netlists (see Table 3). If you read this RTL into Synopsys DC, synthesis will result in the minimal form – a trivial circuit shown below.

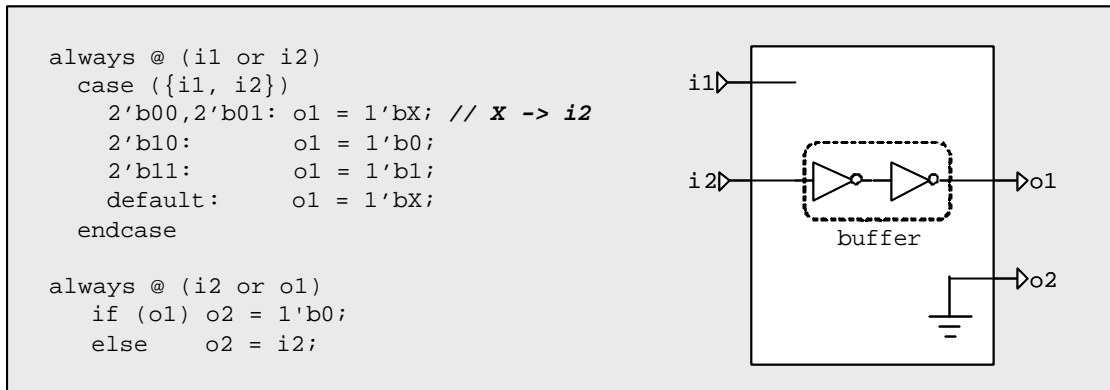


Figure 3: Synthesis to Minimal Form

Consider running an exhaustive simulation on the RTL code, and an exhaustive simulation on the netlist. The results are shown in the table below.

		RTL Simulation		Netlist Simulation	
Input i1	Input i2	Output o1	Output o2	Output o1	Output o2
0	0	X	0	0	0
0	1	X	1	1	0
1	0	0	0	0	0
1	1	1	0	1	0

Table 4: Different Simulation Results for RTL and Netlist Simulations

The simulation results are different for the netlist and the RTL, highlighting how a passing test in the RTL simulation can fail in netlist simulation. This is due to the optimistic interpretation of X-assignments in an `if` statement, leading to just the `else` branch being active when output `o1` is set to X.

Rather than run netlist simulations (which are much slower than RTL simulations) people now rely on equivalence checkers to find functional differences. Unfortunately, the default modes of both Synopsys Formality and Verplex Conformal will pass this RTL vs. netlist comparison as being equivalent. This is because equivalence checkers will treat X's in the reference RTL as *Don't-Care* (which can legitimately be synthesized to 0 or 1), without considering the effects of the *Unknown* X-semantics in RTL simulation.

So, what does this result mean? In *2-State Consistency* mode, the equivalence checker:

- Has verified that the synthesis tool has correctly implemented the circuit
- Has missed a n important difference between RTL and netlist simulation, due to a synthesis don't-care

If you switch over to a stricter *2-State Equality* comparison, you will find the difference between RTL and netlist for this example. Each X is modeled as a pseudo input that can be set to 0 or 1 for each input combination, as illustrated in the table below. This confirms the results in Table 3, that the difference occurs when input `i1` is low and input `i2` is high.

		RTL Reference (X = pseudo-input E in reference RTL)			Netlist Implementation	
Input i1	Input i2	Pseudo Input E	Output o1	Output o2	Output o1	Output o2
0	0	0	X=0	0	0	0
0	0	1	X=1	0	0	0
0	1	0	X=0	1	1	0
0	1	1	X=1	0	1	0
1	0	-	0	0	0	0
1	1	-	1	0	1	0

Table 5: Difference found by 2-State Equality Semantics

So, the current default settings for equivalence checkers are not sufficient for RTL vs. netlist comparisons when there are reachable don't-care X-assignments in the RTL. Question is, should you use *2-State Consistency* or *2-State Equality* for RTL vs. netlist comparisons? A recommendation is given in section 4.2.4 for a safe methodology that also considers the problem of false negatives.

4.2.3 Equivalence-Check Problem #3: 2-State Consistency can miss RTL vs. RTL differences

2-State Consistency is the recommended default mode for RTL vs. RTL comparisons, with *2-State Equality* used to check that the don't-care space is identical (e.g. for Verilog to VHDL translations).

However, the example netlist in Figure 3 could just as easily be modified RTL – with the don't-care X-assignment rewritten as $o1 = i2$. So, the problem described in section 4.2.2 is equally applicable to RTL vs. RTL comparisons. However, for RTL rewrites in the same HDL (e.g. Verilog to Verilog) you will have a chance to spot the differences by rerunning RTL simulation regressions.

4.2.4 Recommended Settings for Equivalence Checking

Just using *2-State Equality* seems to be a good idea because it can find a real simulation difference for output o2 in sections 4.2.2 and 4.2.3. However, there are problems with this approach.

This strict comparison mode will also show that output o1 is non-equivalent (in rows 2 and 3 of Table 5). Is this correct? This difference does not fail the RTL simulation, but this is just a toggle test on each output and does not check any functional dependence of output o1. Table 3 shows that output o1 is different for every logic minimization, which sounds like a recipe for disaster (especially as these differences are masked in RTL simulations). Bottom line is that this X is reachable and so this difference has to be considered as a bug.

You can also get non-equivalences from *2-State Equality* that are false negatives, i.e. not a real difference that would show up in simulation. This is due to the fact that equivalence checkers cannot determine if the don't-care X-assignments are reachable (and could therefore cause a simulation difference).

To avoid missing bugs in both RTL vs. Netlist and RTL vs. RTL comparisons, the first recommendation is:

Recommendation 2: Always start an equivalence checker in the strict *2-State Equality* comparison mode (the default settings of the tool can miss bugs).

If this comparison passes then you're done. However, such a comparison will sometimes lead to differences that are false negatives – hence the following.

Recommendation 3: If a comparison fails with strict *2-State Equality* but passes with *2-State Consistency*, the pass is acceptable provided you can prove that all X's causing differences are unreachable (i.e. the failures are false negatives).

See the techniques described in section 6.5 to prove that these X's are unreachable (maybe these techniques could be included in future versions of equivalence checkers to automate this recommended flow). Finally, when comparing RTL Verilog against RTL VHDL you really need to know that the don't-care space is exactly the same.

Recommendation 4: RTL Verilog vs. translated RTL VHDL should be equivalence checked using *2-State Equality* (to ensure that the don't-care space is identical).

4.2.5 Equivalence-Check Problem #4: Sequential Differences missed if X's stored in Registers

Equivalence checkers exhaustively compare combinatorial logic cones between primary I/O and internal registers, by driving the cone inputs to all possible 2-state combinations and comparing the results. They indirectly consider X's stored in registers as 0 or 1 because register outputs are inputs to the logic cones (along with primary inputs).

However, equivalence checkers do not consider the *sequential* relationship between an X stored in a register and the corresponding X in the next-state logic (from the previous clock cycle). This loss of reachability information can cause false negative failures in the comparison result and also some misleading debug information.

This is a fundamental limitation of current equivalence checkers, which cannot be overcome by changing to an even stricter comparison mode. It is also an advantage - the comparison is very fast because only combinatorial logic is checked. There are some *sequential* equivalence checkers (particularly in academia) but these are restricted in what they can compare (need to be targeted at a small part of the circuit). There is also a limited *sequential* check in Synopsys' Formality, see [Synopsys 02] for the `set_paramters -retimed` command, but this is only suitable for small isolated parts of a design.

Consider the following RTL that uses `casex` to concisely specify case-items.

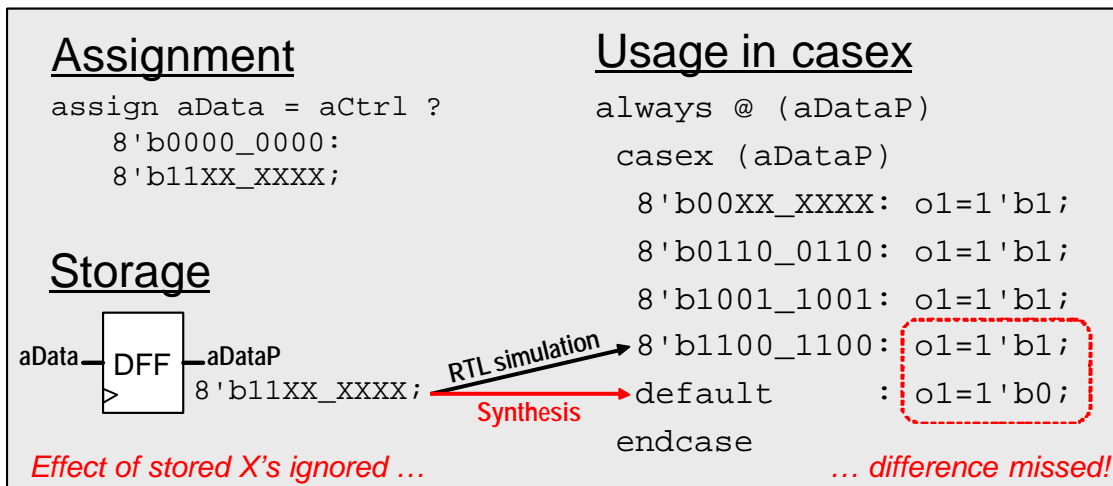


Figure 4: False Positive Equivalence due to X-storage

Most users are familiar with the fact that you can use wildcards in `casex` (and `casez`) to write concise case-items that will match many 0/1 combinations. However, many do not realize that wildcards are 2-dimensional – an X or Z in a `casex` selection expression is also treated as a wildcard and can match in unexpected ways. If this incoming X is stored in a register, equivalence checkers will ignore this wildcard (but RTL simulation will not).

Consider the example above, where 6 out of 8 bits can be assigned to X as don't-cares. When these X-assignments are stored in the register, RTL simulations will always pick the 4th case-item. Synthesis will result in each X-assignment being minimized to a 0 or 1. Statistically, netlist simulation will pick the default 98% of the time (given 2^6

possible minimizations) but it may even be higher (as every X to zero would make sense here, to reduce the logic for the conditional selection). The problem is that the `default` assigns 1'b0 whereas the 4th case-item assigns 1'b1 to the output.

This example shows a difference between RTL simulation and netlist simulation, but will equivalence checking spot this difference? The result depends on the X comparison mode:

- **2-State Consistency:** 100% equivalent. Output `o1` and next-state `aData[7:0]`.
- **2-State Equality:** 67% non-equivalent! Next-state `aData[5:0]` (but output `o1` is equivalent!)

The strict comparison mode reports differences in the `aData[5:0]` next-state function, but only because the equivalence checker can always set the X's to the opposite value that synthesis chose. Despite this difference, the `o1` output is reported as equivalent due to the combinatorial nature of the comparison— so the user may dismiss the reported differences as unimportant.

The reason for output `o1` being equivalent can be investigated with an RTL vs. RTL comparison, where the modified RTL just assigns: `aData[5:0]=5'b00000`. The output function is unchanged so will be reported as equivalent.

There are many problems associated with `casex`, including its subtle semantics (along with the abundance of X-assignments in most RTL) and difficult translation to VHDL, so they should never be used. This guidance is also given by Don Mills [Mills 99], who goes on to say that `casez` is much safer (as you don't typically have Z-assignments in RTL). However, they should be avoided if possible because `casez` is still difficult to translate to VHDL and does not fully propagate X's (if an X on the case-selection matches a Z-wildcard in a case-item). Hence the following:

Recommendation 5: Never use `casex` statements as they are just too dangerous, and avoid `casez` whenever possible.

4.2.6 Simulation Semantics Ignored by Synthesis and Equivalence Checking

This paper has already shown two examples that cause fundamental problems for equivalence checking, even with a strict *2-State Equality* comparison, due to its combinatorial nature:

- **Figure 1:** Simulation differences caused by Latch behavior (100% equivalent)
- **Figure 4:** False Positive Equivalence due to X-storage (differences found, but too easily ignored)

These examples show differences between RTL and netlist simulations that are due to the affects of X semantics on RTL simulation. RTL coding guidelines can reduce these affects, but a more general solution is to avoid reachable X's in the first place (see Recommendation 1) and thereby improve the effectiveness of equivalence checking.

Formal property checkers do consider *sequential* semantics, so they can be used to analyze X-assignments to see if they are reachable (see automated proof techniques in section 6.5) or safe in another way (e.g. only reachable when not read, as described in section 6.6). This leads to the following recommendation:

Recommendation 6: Use automatic property checking to prove that an X is unreachable, or interactive property checking to prove that the X is not stored in a register.

4.2.7 Synthesis Semantics Ignored by Simulation

Consider the example below that uses synthesis pragmas to give extra information to synthesis (which are ignored by RTL simulation).

```
case (1'b1) // synopsys full_case parallel_case
  sel[0]: HSize=2'b10;
  sel[1]: HSize=w1;
  sel[2]: HSize=w2;
  sel[3]: HSize=2'b01;
endcase
```

Verilog Snippet 6: One-hot Case with Bad Coding Style (two synthesis pragmas)

The above Verilog will look odd to many designers, but is perfectly legal. It's effectively a priority encoded nested if expression, with every case-item compared in turn i.e. "if (sel[0]==1) HSize=2'b10; else if (sel[1]==1) ...". Although this doesn't look like an X-issue, the case statement in Verilog Snippet 6 is equivalent to the following casex statement:

```
casex (sel) // synopsys full_case parallel_case
  4'bXXX1: HSize=2'b10;
  4'bXX1X: HSize=w1;
  4'bX1XX: HSize=w2;
  4'b1XXX: HSize=2'b01;
endcase
```

Verilog Snippet 7: Alternative One-Hot Case with Bad Coding Style

Both of the two bad coding styles above use synthesis pragmas to tell the tool not to synthesize priority encoded logic (which would otherwise be required to avoid clashes when more than one bit of sel is asserted). The parallel_case pragma tells the synthesis tool that all four sel bits are mutually exclusive (i.e. one-hot), allowing it to synthesize logic equivalent to Verilog Snippet 8.

```
assign HSize[1:0] =
  ({2{sel[0]}} & 2'b10)
| ({2{sel[1]}} & w1)
| ({2{sel[2]}} & w2)
| ({2{sel[3]}} & 2'b01);
```

Verilog Snippet 8: Sum-of-Products form of One-hot Mux

What if the selection lines are not mutually exclusive? Consider sel[0] and sel[3] asserted at the same time, as highlighted in bold. The RTL simulation of Verilog Snippet 6 will set HSize to 2'b10 when sel[0] is asserted, due to the priority encoding of a Verilog case statement. However, a netlist simulation of the sum-of-products logic will turn the multiplexer into a merging component— setting HSize to 2'b11. This particular problem was found by formal proof of the AHB property: HSIZE < 2'b11, fortunately before the design was shipped.

Equivalence checkers can be told to follow the same synthesis semantics for full_case and parallel_case, which is clearly dangerous from the above example. Instead, you should avoid using these pragmas altogether (see Cummings [Cummings 99] for more information). Some people write a one-hot multiplexer as a sum-of-products directly in their RTL – if you do this then you should add a one-hot assertion (the OVLassert_one_hot assertion also checks for X's on the selection) and, ideally, use automatic formal proofs to ensure it's always safe to use (see section 6.5).

4.3 Misleading Code Coverage

Due to X-Optimism in Verilog RTL semantics, it's possible for code-coverage to:

- **claim a branch is not covered when it's logically reachable (by some 2-state combinations)**

- **pass a branch as covered when it's logically unreachable (i.e. no 2-state combination)**

Such problems have been found at ARM by careful comparison of code-coverage reports with automatic property checking results (see section 6.5.1).

The following simple example illustrates both problems. Due to a single don't-care X-assignment that's reachable, an exhaustive RTL simulation (in fact, just input `i1` at 0 then 1) will claim that two assignments to output `o1` are coverage holes. This is because the `2'b01` and `2'b10` assignments can only occur when `i1` is low, in which case both wires `w1` and `w2` are X; consequently the final `else` branch must be taken. Code coverage will pass the `2'b00` and `2'b11` assignments as being hit, but deadcode analysis (see section 6.5) will show that the final `else` assignment is impossible to reach (because `w2` is the inverse of `w1`).

<pre> assign w1 = i1 ? 1'b1 : 1'bX; assign w2 = ~w1; // Note: ~X=X always @ (i1 or w1 or w2) if (i1) o1 = 2'b00; else if (w1) o1 = 2'b01; else if (w2) o1 = 2'b10; else o1 = 2'b11; </pre>			
		Code Coverage (i1=0; i1=1;)	Automatic Formal Proofs
	←	covered	reachable
	←	hole !	reachable
	←	hole !	reachable
	←	covered	deadcode !

Figure 5: Code Coverage affected by X-Optimism and X-Pessimism

It's important to realize that the code-coverage tool is correctly implementing the Verilog semantics, but this is arguably not appropriate for its task i.e. a test metric to ensure every branch has been hit (to have a good chance of finding bugs). Code coverage is *not* incorrect in the above example, but it's certainly misleading. This example highlights why:

- **it's sometimes impossible to reach 100% code coverage**
- **100% code-coverage could be optimistically high (with deadcode left in your RTL)**

Both these problems can make it difficult to find bugs, which leads to another recommendation.

Recommendation 7: Use automatic property checking to investigate code coverage reports.

Automatic property checking is excellent at finding deadcode, but branches reported as reachable might in fact be *conditionally reachable*. In the above example, the `2'b01` and `2'b10` assignments are both reported as reachable but are in fact mutually exclusive – it depends on the minimization of the reachable X-assignment.

5 Why are X's Inefficient?

Some X assignments in Verilog RTL can be inefficient rather than dangerous. They can reduce the efficiency of EDA tools or the productivity of engineers.

5.1 Unnecessary X's in Netlist Simulations

As should be expected, netlist simulations more closely model an actual circuit than RTL simulations. However, they can still suffer from X-pessimism and loss of context information (see Verilog Snippet 1). An example of this is shown in Figure 6 below, where both sides of a Multiplexer are masked by X's – even when the inputs are identical (in which case the output would be driven by the input in a real circuit).

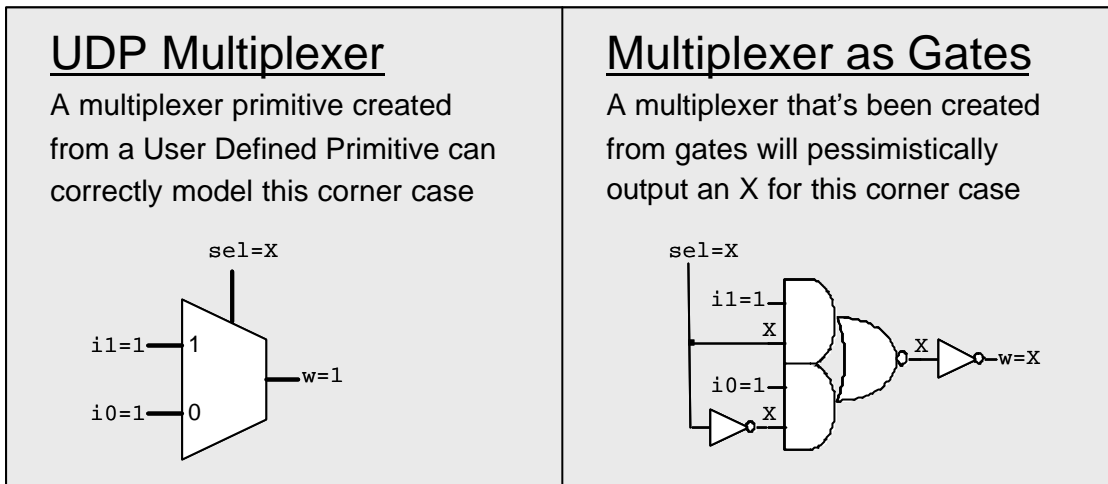


Figure 6: Multiplexer Smashed into Gates can Pessimistically output X's

X-pessimism means that netlist simulations can produce false-negative results, which may require extra debugging effort. However, you should not miss X-related false positives (assuming you run enough netlist simulations).

5.2 Non-Minimal Synthesis of Don't-Cares

One of the main reasons given for using X-assignments is to improve logic minimization. It's true that don't-cares allow better minimization but it's not always the case that you get the minimal result you expected, and it's often difficult to determine that it's gone wrong (when did you last inspect large schematics?).

At ARM we have a coding policy that discourages certain Verilog styles that can lead to semantic differences between RTL and netlist simulations, including:

- **parallel_case :** synthesis semantics ignored by simulation (not priority encoded)
- **casex :** simulation semantics ignored by synthesis (X on case-expression)

At ARM, multiplexers with one-hot selection are used for areas that are speed critical. They are either written directly in a sum-of-products form or as a one-hot case statement – an example of which is shown below. Note that the RTL below follows good coding-practice, unlike the examples in Verilog Snippet 6 and Verilog Snippet 7.

```
case (sel)
  3'b001: f=i0;
  3'b010: f=i1;
  3'b100: f=i2;
  default: f=2'bXX;
endcase
```

Verilog Snippet 9: One-hot Case with Good Coding Style

Prior to logic minimization, Synopsys DC does “template matching” to see if it recognizes design-intent for synthesis from the coding style. The initial template matching phase recognizes the bad coding styles in Verilog Snippet 6 and Verilog Snippet 7, and duly minimizes the logic to an efficient 2-stage sum-of-products form. This ideal minimization of a 3-bit one-hot multiplexer is illustrated in the Karnaugh Map below.

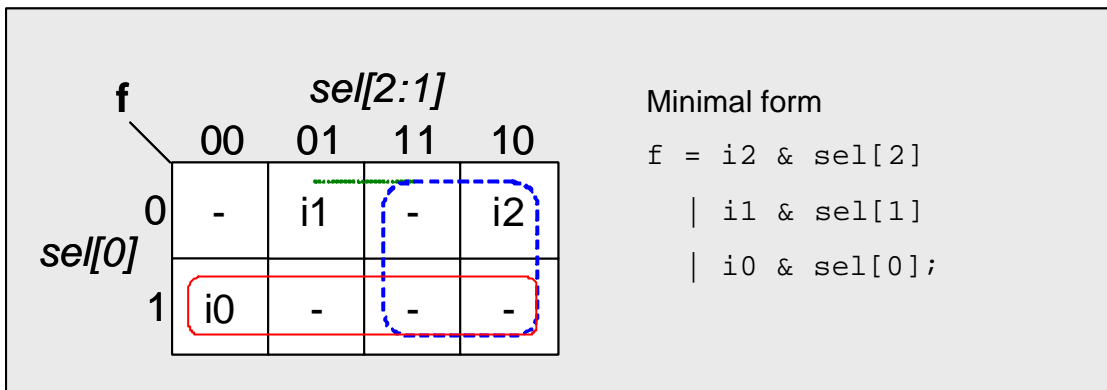


Figure 7: Minimal K-Map of a One-Hot Multiplexer

Rather than implement this minimal sum-of-products directly in an OR-AND form, it is optimized to a NAND-NAND form as shown below (which also uses a composite gate from this library). Note that during optimization the minimal form will not change (which is chosen during the HDLC or Presto compilation stage).

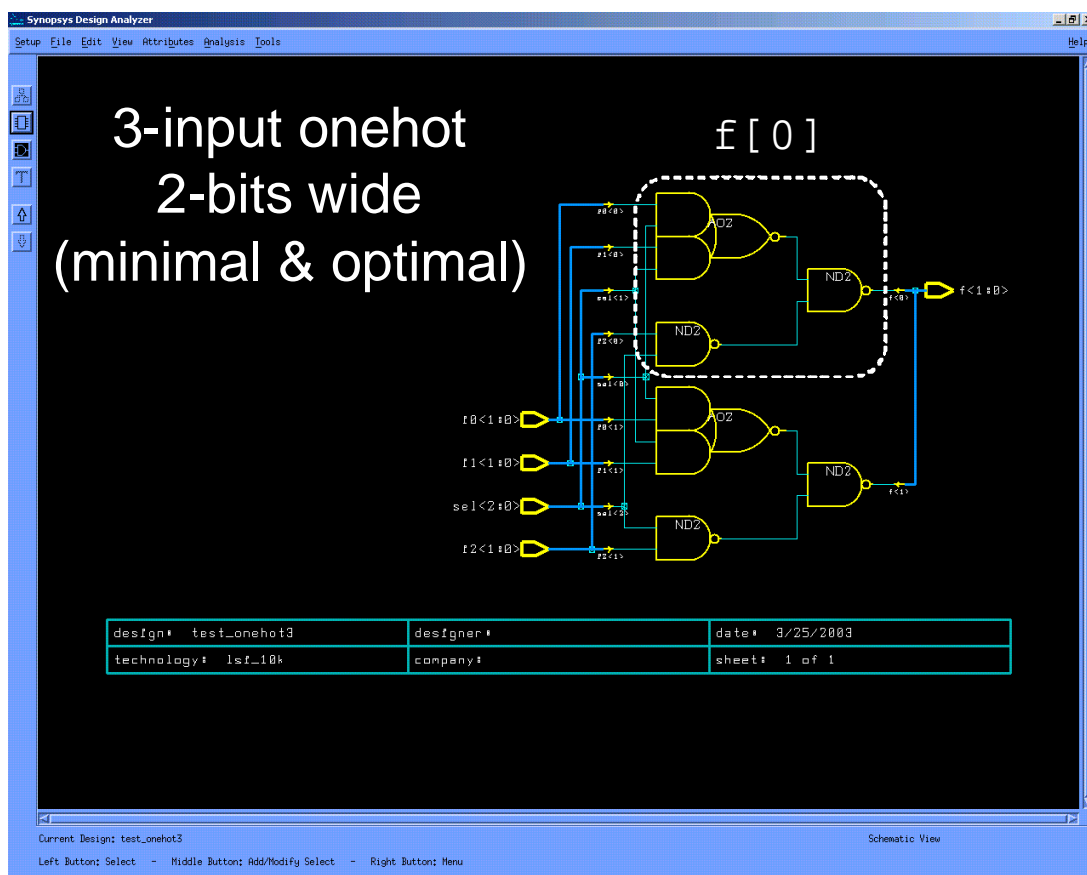


Figure 8: Minimal Synthesis for One-Hot Multiplexer

Unfortunately, the template matching in Synopsys DC does not recognize the standard Verilog one-hot case statement in Verilog Snippet 9 (despite following good coding practice). It then goes onto its proprietary minimization algorithms but produces a non-minimal result, as illustrated below.

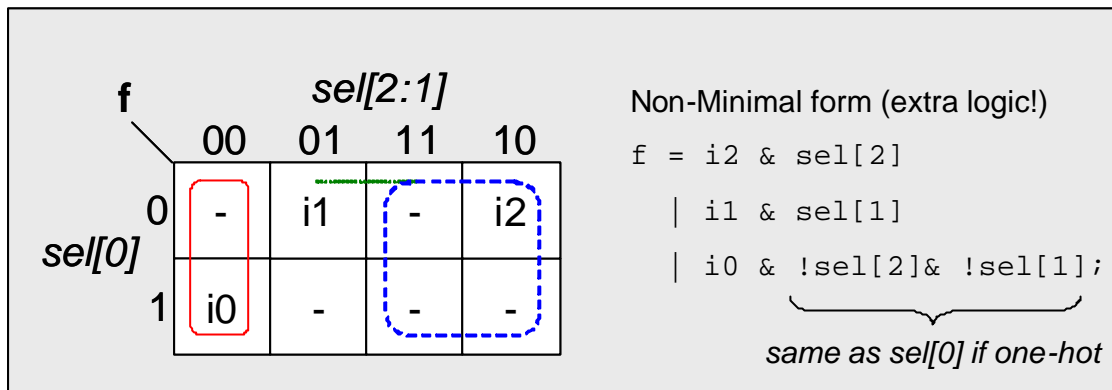


Figure 9: Non-Minimal K-Map for a One-Hot Multiplexer

This inefficient minimization produces an expression with extra terms and, finally, results in extra gate stages as shown in Figure 10 below. This means more gates, wires, a slower circuit and increased power consumption.

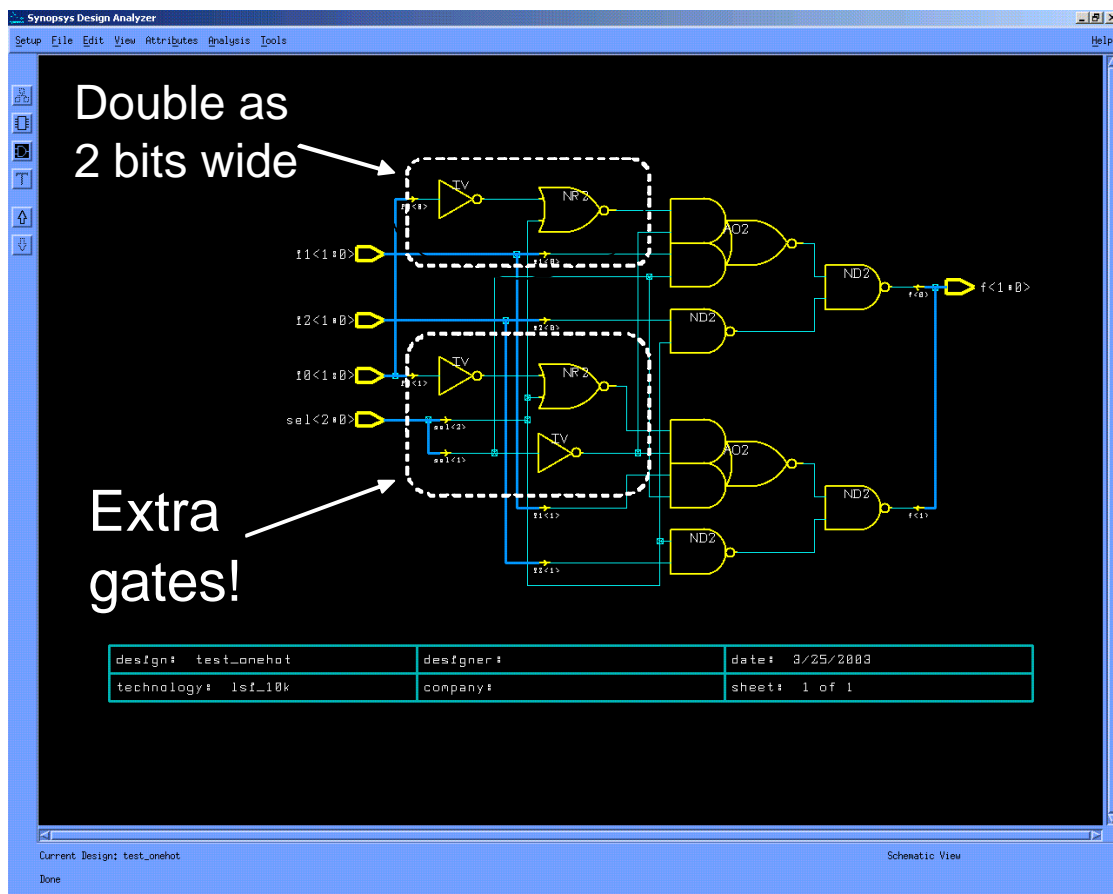


Figure 10: Redundant Gate Stages from Non-Minimal Synthesis

This issue was highlighted to Synopsys and fixed in version 2003.06-SP1, but only for the Presto compiler.

If you are using an older version of Synopsys DC, or use 2003.06-SP1 but with the HDLC compiler, you will still see this problem. Aside from rewriting it directly as a sum-of-products form, there is a workaround that sometimes

manages to steer Synopsys DC's logic minimization to the best choice. That workaround is to set the zero-case-item's don't-care to 0, as shown below in Verilog Snippet 10.

```
case (sel)
  3'b000: f=2'b00; // workaround: set zero case-item don't-cares to 0
  3'b001: f=i0;
  3'b010: f=i1;
  3'b100: f=i2;
  default: f=2'bXX;
endcase
```

Verilog Snippet 10: Workaround to Improve One-Hot synthesis

This zero-case-item assigning zero can stop Synopsys DC's minimization algorithm from using the zero term to produce non optimal min-terms, by stopping the bad i0 loop shown in Figure 9. However, this workaround is not guaranteed to produce minimal logic - particularly when the case statement has multiple assignments with shared terms (or if the case-expression is used elsewhere). This leads to the following recommendation.

Recommendation 8: **For one-hot logic on a critical path, write the RTL directly in a sum-of-products form (rather than case) and add a one-hot assertion checker.**

5.3 Limited Speedup with 2-State Simulation

Some Verilog simulators have a 2-state (binary) mode that, in theory at least, can lead to a significant speed up in simulation time. If your RTL is written in a good coding style, specifically to avoid reachable don't-care X-assignments, then you can save a lot of time with regressions. Even if you do need to do some 4-state simulations, a 2-state mode could be used for quick functional regressions. However, you still might not achieve the speed ups that you would expect from 2-state simulation.

For example, the VCS User Guide [Synopsys 02] describes the +2state switch in the Synopsys VCS simulator - which tries to model as much as possible in 2-state but has to retain 4-state simulation for some signals (e.g. if compared via the == case-equality operator). This sounds ideal but we have experienced very little speed up at ARM because:

```
"As stated in IEEE Std 1364-1995, page 33, the comparison in the case
equality and inequality operators matches the comparison in a case
statement. Therefore if a signal is in the case expression in a case
statement (or a casez or casex statement) then the signal must retain
four state simulation so VCS can look for X and Z value bits."
```

So, the performance benefit of 2-state simulation will be very limited if you have lots of case statements in your Verilog RTL. This issue has been highlighted to Synopsys and they are investigating the possibility of providing a mechanism to bypass this (which should be safe with a suitable RTL coding style for case statements). Note that SystemVerilog has separate 2-state datatypes to overcome this problem (see section 7.6).

5.4 Slower Formal Verification

Formal verification exhaustively applies all 0/1 combinations (and also sequences for property checking) to verify a design. In addition to this, it can also set X's in any way to try and break the verification. Therefore the presence of X's (either as don't-care assignments or un-initialized registers) will significantly slow down formal proofs.

The most significant reduction in don't-care X's will probably come from a minimization approach (e.g. Espresso or Synopsys DC). For the ARM1136J(F)-STM over 5,500 X's were removed from the Core in this way and as a result the deadcode automatic property checking changed from a weekend run to under 2 hours. Similarly, compilation time for formal verification tools can be speeded up significantly.

6 How To Find Dangerous X's

The following sections describe different techniques to investigate X's that can cause problems.

6.1 Additional Simulations

We have seen in this paper how X-Optimism, particularly around `if` statements (and `case` statements with defaults that don't assign X), can mask bugs that can only be revealed by netlist simulations. The problem with netlist simulations is that they are slow to run, so this section describes some alternative RTL simulations.

6.1.1 RTL Simulation #1: All X's to 1 (then all X's to 0)

To reveal problems with reachable don't-care assignments you can try minimizing them in the RTL and rerunning RTL regressions. There are far too many possible combinations to explore, but the two extremes (all X's minimized to 1, and all X's minimized to 0) should be tried. It sounds crude but can be effective, particularly the all X's to 1's (which has more chance of taking a different, i.e. non-X, branch in an `if` statement or default of a `case`).

It's fairly straightforward to write a script to achieve this simple minimization. One way is to first translate every X-assignment into a standard form (either `1'bX` or `{N{1'bX}}` for multi-bit assignments) and then just using a simple sed script to change every `1'bX` (this works if you don't allow `casex` in the RTL).

6.1.2 RTL Simulation #2: Minimized RTL as Logic-Equations

A more realistic way of investigating the effects of synthesis minimization with RTL simulation is to minimize the don't-care X's in the exact same way as your synthesis tool (but without producing a netlist). For Synopsys DC, you can either use the `compile -no_map` command or set the `verilogout_equation` TCL variable to produce minimized RTL in the form of Boolean equations (rather than a GTECH netlist). Minimization is performed in Synopsys DC by the Verilog compiler, but can be different according to the choice of compiler (HDLC or Presto) so you should try both. Note that after the Verilog RTL has been compiled, minimization is not affected by any target library optimizations in either Synopsys DC or Synopsys PC.

You may find that you only have to do this with small parts of your design, as there's no point minimizing blocks that have no reachable don't-care X-assignments. Complex combinatorial blocks with a large don't-care space are ideal targets for this technique (e.g. decoders in a microprocessor).

6.1.3 RTL Simulation #3: Un-Initialized Datapath Registers

A similar problem to reachable don't-care X-assignments are initialization X's from datapath registers that are not reset, and could interact in exactly the same X-optimistic way in `if` or `case` statements. Such initialization X's are not exposed by minimizing the logic and there's no associated X-assignments in the RTL.

The simplest technique is to set the initial value of all registers to `1'b1` (or all to `1'b0`) at the start of a simulation, just prior to reset. After reset, the control registers will be reset to their normal initialization value and an RTL simulation can be run to expose X problems due to un-initialized datapath registers affecting the control logic. Again, two simulation regressions should be tried (first with all-1's, then with all-0's).

A more sophisticated technique is to randomly initialize registers prior to a reset. See Bening [Bening 99] for details of this approach (and the use of netlist simulations to reveal X-issues). Note that some simulators provide flags to initialize registers to all-1's, all-0's or random (e.g. `+vcs+initreg+N` for Synopsys VCS, with $N>1$ for random).

6.2 RTL Code Inspection

This section gives some ideas about how to identify potential problems in your RTL by code inspection.

6.2.1 All Don't-Care X-Assignments

The set of all don't-care X-assignments cannot be found with a simple script, because you need to ignore some. RTL often contains X-assignments that are not synthesis don't-cares, but instead used for X-propagation. These are assignments that are redundant in all 0/1 combinations e.g. a default X-assignment in a case statement where all 2-state enumerations are covered in preceding case-items.

Linters and equivalence checkers can be used to list all potential don't-care X-assignments, e.g. Verplex Conformal will tell you this with the command:

```
report rule checks -verbose
```

Not all reported don't-care assignments will cause a problem, so this list can be further reduced by comparing the RTL with minimized netlists using strict *2-State Equality* (although this may be applicable to just one netlist). This list can be reduced further still, but not by linters or equivalence checkers (which can only do combinatorial analysis). Sequential analysis is often required to formally prove that don't-care X's are in fact unreachable (see section 6.5).

6.2.2 X-Termination Case Defaults

Case defaults that terminate X-propagation should be investigated, and it's possible to find these with a simple script. However, it's still useful to determine if the defaults themselves are reachable (either redundant in all 2-state combinations or unreachable). Again, formal proofs can be used for this (see section 6.5).

6.2.3 Un-Initialized Datapath Registers

You can write a script to search for clocked assignments that don't have a reset term. A more rigorous check (to ensure the reset has been used for the desired affect) is to use automatic-property-checking "reset" checks to find these datapath registers. Any registers that are less than, say, 16-bits wide should be examined to see why they should not be reset (resetting control registers helps to simplify validation and formal verification).

6.3 Detecting X's with Assertions

It's possible to write assertions into your RTL that return an error when an X is found during an RTL simulation. This is best done during RTL development, but it's also possible to automatically add assertions to existing RTL – with the targets being all *if/case* selection expressions. However, adding X-detected assertions throughout your RTL can produce a flood of errors (99.9% of which are false negatives).

Care is required in controlling how an X is detected and when it's deemed to be an error. ARM is proposing two new X-checking assertions for adoption to the Accellera committee for OVL (Open Verification Library):

1. `assert_never_at_x_or_z(clk, reset_n, qualifier, test_expr)`
2. `assert_never_go_x_or_z(reset_n, test_expr)`

The first X-assertion checks if the test expression is *at* X or Z when a qualifier is high (e.g. 32-bit data bus should not have X's when a read is performed). If you want to check a signal at all times, simply set the qualifier to `1'b1`. It's good to add this checker during RTL development.

The second X-assertion checks that if the test expression is at 0 or 1 it does not subsequently *go* to X, which could indicate a problem – particularly on a control signal. You might not care if a signal is X during some initialization period following reset but once the signal goes 0 or 1 you don't want to see it return to X. This checker produces far fewer error messages, which makes it the best choice for automatically adding X-checkers to existing RTL.

These proposed OVL assertions have been written in a careful RTL style that deliberately avoids the use of case-equality (i.e. `===`) in Verilog to check for X/Z values. Instead, the `test_expr` is Xor'd with itself and then Or-reduced, which can only return a 0 or X value. This approach means that these assertions can stay in the RTL for

both 2-state and 4-state simulations, and for 2-state simulations these X-checking assertions are redundant (so a simulator should be able to optimize them away completely).

6.4 Change Default Settings of Equivalence Checkers

If you change the default of equivalence checkers to *2-State Equality* for all comparisons, you may reveal problem X's. However, care is needed as you may also find false negatives. See section 4.2.4 for a recommended approach, and section 2.4 for details on how to configure specific tools.

6.5 Automatic Formal Proofs of Unreachable (deadcode) Assignments

Formal property checking can be used to establish which don't-care X-assignments are unreachable (and therefore safe) due to its *2-State Sequential* semantics (you often need formal verification over time to prove that something cannot occur in the design state). User driven formal property checking can require a lot of effort, but much of the burden of proofs for X's can be automated by extracting standard properties from the RTL itself. This technique is known as Automatic Property Checking (Autochecks) in the Averant Solidify tool, and other commercial property checkers provide similar "super linting" features e.g. Pre-Defined Checks (PDC) in Verplex's Black-Tie, or Automatically Extracted Properties (AEP) in Synopsys' Magellan.

This section describes how the X-analysis autocheck in Solidify (based on the deadcode check but customized for Verilog X-issues) has been used at ARM to investigate many aspects of X reachability, but the methodology and terminology should be applicable to other tools.

Solidify's X-analysis autochecks have three levels to classify reachability of case-items and `if/?` branches:

1. **reachable:** reachable in 2-state (i.e. 0 or 1) combinations
2. **deadcode:** proven not reachable in 2-state (but could be reachable via an X)
3. **redundant:** trivially not reachable in 2-state (e.g. case-default that is only reachable via an X)

All 3 classes above apply to branches with 2-state assignments (e.g. deadcode assignment to `1'b0` is redundant logic, so could indicate a bug in the RTL or at the very least a code-coverage hole). They also apply to X-assignments in the RTL, which are classified as one of three X-categories:

- **x-assignment:** indicates don't-cares, which are good (for synthesis) but only safe if proven as deadcode
- **x-propagation:** X assigned to propagate incoming X's; should only be used if proven to be redundant
- **x-termination:** 2-state assignment that terminates incoming X's; dangerous if it leads to X-Optimism

Every X-assignment (and some 2-state assignments) in your RTL will then be categorized as one of the following possible combinations of branch-class + X-category (if no X-category is given, the branch assigns 0 or 1 values).

Dangerous Categories

- **deadcode:** Redundant RTL (could be a real bug, or at least a code-coverage hole)
- **reachable x-assignment:** Reachable don't-cares reduce semantic overlap
- **reachable default x-termination:** Suffers from X-Optimism (just like `if` statements)
- **redundant x-termination:** No possible reason for 2-state assignment in 2-state redundant code

Beneficial Categories

- **deadcode x-assignment:** Unreachable don't-cares can improve synthesis (non-trivial proof)
- **redundant x-propagation:** Must be better to propagate than terminate (aids debugging)

All of the above combinations are useful to quickly analyze and improve your code. In particular, you want to eliminate reachable x-assignments (but, conversely, maximize deadcode X-assignments to safely improve synthesis).

With some user interaction, it is possible to formally prove that an assignment classified as *reachable x-assignment* is in fact safe (see section 6.6). Note that redundant branches are often thought to be exclusively case defaults, but can include `if` or ternary `? else` branches used explicitly to propagate X's.

6.5.1 Using Autochecks to Improve Code-Coverage

As hinted above, X-analysis (deadcode) formal-autochecks can be used to explain holes in a code-coverage report (if code-coverage says that a branch has not been hit, it can be shown to be unreachable by these proofs).

Conversely, you should double-check that any deadcode reported by autochecks was not falsely reported as being covered by the code-coverage tool. This can happen due to:

- **Glitches that cause a transient to an unreachable branch**
- **Optimistic interpretation of X-semantics**

Autochecks (and interactive formal proofs) have been used at ARM to improve code-coverage reports and to debug EDA tools/flows. See section 4.3 for an example of misleading code-coverage.

6.5.2 Aside: Using Autochecks to Avoid False Negatives in Formal Verification

Formal verification (both equivalence checking and property checking) tools can sometimes give false negative results when they have considered the design starting in an unreachable state (e.g. a one-hot state machine starting at all ones). Results from autochecks can provide additional information about the design's possible state, which can be reused to improve formal verification elsewhere.

6.6 Interactive Formal Property Checking

The previous section described automatic formal proofs to determine the reachability of don't-care X-assignments, which requires very little user interaction. However, sometimes user-driven full property checking is required to determine if some reachable don't-cares are in fact safe.

```
always @ (CLK or nRESET)
    if (nRESET==1'b0)
        Count <= 2'b00;
    else if (CountEn)
        Count <= NxtCount; // NxtCount is only read here

always @ (State or Count or req or abort)
    case (State)
        3'b000: begin
            NxtState = 3'b001; // Idle -> Wait
            NxtCount = 2'bXX; // reachable X-assignment
        end
    <...>
```

Verilog Snippet 11: X-assignment that is reachable but could be safe

Consider the RTL example above, where some control logic consists of a state-machine and a counter. Automatic formal proofs will show that the X-assignment to `NxtCount` is reachable. However, the counter is not used for the first two states, and `NxtCount` is only read to assign `Count`, so this don't-care could be safe if the X is never read. To show that it's safe, you need to formally verify the following property.

```
if (State==3'b000) (CountEn==1'b0);
```

Verilog Snippet 12: Property to prove that X-assignment is safe

If the above property is exhaustively proven, the X-assignment will be reachable but unused. Provided that no reachable X's are assigned to `NxtState`, the above X-assignment to `NxtCount` will be safe. As the property is

just a combinatorial check I've expressed it in Verilog rather than a formal property language, which also allows us to reason about X's in the property. Note that an X on `CountEn` will cause the property to fail but any X on `State` would cause the property to pass (so you need to check for any reachable X-assignments to `NxtState`).

6.7 Netlist simulations

Netlist simulations are much slower than RTL simulations and have been largely replaced in the main by equivalence checking (although "smoke" tests are often run on netlist simulations as a sanity check). Netlists have much better X-propagation than RTL, which is a good reason for running some regressions as netlist simulations (particularly for investigating un-initialized datapath registers).

Netlists can suffer from X-pessimism, e.g. a 2-input multiplexer implemented with an inverter on the selection will not pass through identical values on the data lines when the select is an X (unlike a UDP mux or the ternary `?` operator in RTL). However, X-Pessimism is better than X-Optimism and if the netlist simulation produces no X's you're done. Note that it can be difficult to find cases of X-Optimism in netlist simulations where there is X-Pessimism

7 How To Avoid Dangerous X's

This section describes how to overcome problems related to X (after they have been identified and analyzed using techniques described in section 6).

7.1 Good RTL Coding Practice

The unwanted affects of X semantics can be reduced by following some RTL coding guidelines, including:

1. For `if` statements:
 - a) Never use `if` statements in combinatorial logic (use `case` or ternary `?` instead)
 - b) Only use `if` statements for sequential elements (e.g. flip-flop with asynchronous reset)
 - c) Add X-checking assertions to a clock-gating enables in sequential logic, e.g. `if (~enable)`
2. For `casex` and `casez` statements:
 - a) Never use `casex` (it's far too dangerous)
 - b) Avoid `casez` if possible (Z-wildcard doesn't propagate X's and it's hard to translate to VHDL)
3. For `case` statements:
 - a) Always add a `default` line (to avoid X-Latching)
 - b) Only use the `default` to assign X's (to avoid X-Optimism)
 - c) Never use explicit X's in case-items
 - d) Cover all *reachable* 2-state values with case-items
 - e) Avoid using `case` for one-hot multiplexers on a critical path (use sum-of-products instead)
4. Reduce the number of reachable X's:
 - a) Discourage the widespread use of X-assignments as synthesis don't-cares
 - b) Consider pre-minimizing essential don't-care X's prior to RTL verification (see section 7.2)
 - c) Avoid flip-flops that are not reset (only exception should be for large datapath registers)
5. Avoid synthesis/simulation specific workarounds that change semantics:
 - a) Never use `full_case` or `parallel_case` synthesis pragmas

- b) Avoid `translate_off/on` pragmas that change RTL simulations (e.g. for `casez` X-propagation)

When you cannot follow these rules for any reason, e.g. complexity or legacy, use X-checking assertions and formal property checking to verify the RTL.

7.2 Removing Reachable Don't-Care X-assignments

Previous sections have shown that don't-care X-assignments can cause different RTL and netlist simulation results, which are usually missed by equivalence checking. If you can't demonstrate that such assignments are unreachable (by assertions or, ideally, formal proofs) then you should assign fixed values to these reachable don't-cares. Any don't-cares that can be proven unreachable should be left in (to improve synthesis without causing semantic differences).

When there's a lot of don't-care assignments in a block, e.g. instruction decoders, you will be better off automating the minimization rather than manually deciding on appropriate assignments. You can do this with either:

1. Espresso minimization (keeping the original PLA table as comments in the Verilog)
2. Reading the original RTL into Synopsys DC and producing logic equations rather than netlists (use the `verilog_out_equations` TCL variable, or `compile -nomap`).

Experience at ARM has shown that if using Synopsys DC, the `verilog_out_equations` TCL variable produces more optimal logic than the `compile -nomap` command.

An added benefit from removing X's is that formal property checking will become much faster (as it has much less work to do).

7.3 Replacing X-Insertion with Assertions

You should never insert X's into your code to see if they cause problems. Instead, add assertions to your RTL to act as exception handlers – to raise an error for an unexpected event. Note that X's do not stress RTL simulations with both possible values – instead, only one path will be evaluated!

7.4 Enabling X-Propagation

As discussed in section 3.2, assertions should be used to replace X-insertion as a means of finding unexpected results. However, X-propagation can still be useful for RTL simulations to highlight problems with un-initialized datapath registers and reachable X's (although these should be investigated in a different way). Another reason for encouraging X-propagation is that it must be better than X-termination (as illustrated by Figure 2).

Two recommendations are given below to enable X-propagation.

Recommendation 9: **Avoid using if statements, as they optimistically interpret X's. Instead use ternary (i.e. conditional ?) operators or priority-encoded case statements.**

Recommendation 10: **For case statements, cover all reachable 2-state values with case-items and always add a default (but only use it to assign X's, to avoid X-Optimism).**

In addition to these, you should avoid using `casex` and `casez` (see Recommendation 5). Avoiding `casez` is very interesting as a `casez` wildcard can cause X-termination, but this behavior is a good interpretation of X in terms of considering the effects of both 0 and 1. If you consider an X as a bad thing and you always want to propagate it in RTL simulations, you can add an aggressive post-process X-propagation as described by Galbi [Galbi 2002].

7.5 Avoiding Un-Initialized Registers

Where possible, reset your registers (this avoids X-initialization issues and helps to validate and formally verify your design). The exception is for large datapath registers, where the cost of routing a reset is not acceptable.

7.6 Future: System Verilog and Verilog 2xxx

The SystemVerilog language proposed by Accellera has lots of interesting features that could overcome problems associated with X-semantics, including:

1. Several 2-state datatypes (e.g. bit, byte, int) that have defined semantics (as opposed to modes specific to a particular simulator).
2. Qualifiers for selection (`case/if`) statements, called `unique` and `priority`, to clearly define their semantics for both synthesis and simulation (unlike synthesis pragmas).
3. Assertions that are an intrinsic part of the design language itself, rather than an afterthought or a tool-specific assertion language or library.

SystemVerilog has the potential to eliminate many X-semantic issues, but introduces new problems including the *wild-equality* and *wild-inequality* operators that allow you to write an `if` statement with the semantics of `casex`:

```
if (sel == 1'bX) f = 2'bXX;
```

System Verilog Snippet 1: New Wild-Equality operator

Some of the ideas in SystemVerilog may be included in the next version of Verilog's IEEE 1364 standard, sometimes referred to as Verilog 2xxx.

7.6.1 Proposal for Extending the Semantics of case-items

This paper has highlighted several issues where it's difficult to distinguish between 2-state (i.e. 0 or 1) and 4-state (0, 1, X and Z) semantics – particularly with the `default` line of a `case` statement. This paper has also shown that although the `casex` and `casez` statements are very useful for writing concise RTL descriptions, the semantics of their wildcards are complex and often dangerous.

One way of addressing both issues would be to extend the semantics of case items to give a 2-state wildcard, possibly using the "*" symbol to indicate "either 0 or 1, but not X or Z". This would just be syntactic sugar for case-items and therefore a 1-dimensional wildcard, with simple and intuitive semantics (and would probably match design intent for most existing uses of `casex` and `casez` wildcards).

It would make sense to allow this for all three types of case-statement, so that `casez` and `casex` can be completely dropped in favor of the standard `case`. Due to the priority-encoded nature of `case` statements in Verilog, this would also allow a simple method of writing a 2-state catch-all prior to the default line e.g. as follows:

```
3'b***: o1 = 1'b0; // 2-state default
default: o1 = 1'bX; // X-propagation
```

Non-Verilog Snippet 1: Using proposed 2-state case-wildcard

The above would avoid the messy assignments you sometimes see prior to a case statement, particularly when there are multiple assignments. You may even want to assign X's in the 2-state default (e.g. as an X-insertion bug trap).

8 Conclusions

This paper has highlighted the dangers of X-issues throughout the design flow, to raise awareness both internally at ARM and externally (for designers and EDA vendors alike). Terminology has been introduced to help explain the subtleties of X-semantics and to show what's broken in a typical design flow. Simple examples have been given for illustration, but these are based on experience of real problems found by ARM.

Of particular concern is the interaction of supposedly don't-care X-assignments with optimistic X interpretation, especially around Verilog `if` statements, that can mask bugs. Such a theoretical possibility was investigated at ARM despite no netlists showing this problem, and lead to a great deal of work to produce RTL that was semantically

rigorous throughout the design flow. Who wants a hidden bug to pop up a few years down the line due to a new synthesis release choosing a different minimization and breaking a supposedly mature and validated design?

The subsequent investigation at ARM highlighted many problems with different parts of the design flow. One of the outcomes of this work has been this paper, which concentrates on why bugs can all too easily be missed by formal equivalence checkers (due to incorrect usage, despite being the default of the tool and the recommended approach by the documentation). This paper has proposed a new methodology for using equivalence checking, that changes the default X semantics in order to reveal bugs that would otherwise be hidden. It has also described problems in other parts of the design flow, including why passing RTL simulations and code coverage could be giving a false sense of security.

This paper has also highlighted less critical X issues, which lead to inefficiencies rather than bugs. This includes non-minimal synthesis (sometimes for RTL written specifically to improve a critical path) and inefficient simulation (2-state simulation is a great idea but will not work if your Verilog RTL contains `case` statements).

This paper has given practical advice for overcoming X issues. It has described several techniques for investigating hidden bugs due to X issues and gives recommendations to avoid many of these problems. This paper has also described how formal property checking can be used to distinguish which X's are safe and which are dangerous, using either automatic or interactive proof techniques.

9 Top-Ten Recommendations

This section reiterates the recommendations made throughout this document.

- | | |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Recommendation 1: | Even if all RTL simulations pass, a don't-care should be considered to be a don't-know unless it's proven to be unreachable. |
| Recommendation 2: | Always start an equivalence checker in the strict <i>2-State Equality</i> comparison mode (the default settings of the tool can miss bugs). |
| Recommendation 3: | If a comparison fails with strict <i>2-State Equality</i> but passes with <i>2-State Consistency</i> , the pass is acceptable provided you can prove that all X's causing differences are unreachable (i.e. the failures are false negatives). |
| Recommendation 4: | RTL Verilog vs. translated RTL VHDL should be equivalence checked using <i>2-State Equality</i> (to ensure that the don't-care space is identical). |
| Recommendation 5: | Never use <code>casex</code> statements as they are just too dangerous, and avoid <code>casez</code> whenever possible. |
| Recommendation 6: | Use automatic property checking to prove that an X is unreachable, or interactive property checking to prove that the X is not stored in a register. |
| Recommendation 7: | Use automatic property checking to investigate code coverage reports. |
| Recommendation 8: | For one-hot logic on a critical path, write the RTL directly in a sum-of-products form (rather than <code>case</code>) and add a one-hot assertion checker. |
| Recommendation 9: | Avoid using <code>if</code> statements, as they optimistically interpret X's. Instead use ternary (i.e. conditional <code>?</code>) operators or priority-encoded <code>case</code> statements. |
| Recommendation 10: | For <code>case</code> statements, cover all reachable 2-state values with <code>case-items</code> and always add a default (but only use it to assign X's, to avoid X-Optimism). |

A general guideline is to avoid adding don't-care X's to Verilog RTL as a matter of course and to use good RTL coding practice to avoid some X-issues (see section 7.1).

10 Acknowledgements

I would like to thank Stuart Sutherland for reviewing this paper and providing valuable advice, particularly on SystemVerilog. I'd also like to thank many colleagues at ARM for giving feedback on this paper, along with specialists from several EDA companies.

11 References

- [Accellera 03] "SystemVerilog 3.1", Accellera, 2003
- [Bening 99] "A Two-State Methodology for RTL Logic Simulation", Lionel Bening, DAC 1999
- [Bening 00] "Principles of Verifiable RTL Design", Lionel Bening and Harry Foster, Kluwer Academic Publishers, 2000
- [Cummings 99] "full_case parallel_case, the Evil Twins of Verilog Synthesis", Clifford Cummings, Boston SNUG 1999
- [Cummings 03] "Synthesizable Finite State Machine Techniques using the New SystemVerilog 3.0 Enhancements", Clifford Cummings, San-Jose SNUG 2003
- [Foster 03] "Semantic Inconsistency and its effect on simulation", Harry Foster, IEE Electronics Systems and Software, April/May 2003
- [Galbi 02] "RTL X's – A Treasure Trove of Trouble", Duane Galbi & Lok Kee Ting, Boston SNUG 2002
- [IEEE 95] "IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language", IEEE Computer Society, IEEE Std 1364-1995
- [Keating 02] "Reuse Methodology Manual for System-on-A-Chip Designs", Michael Keating, June 2002
- [Mills 99] "RTL Coding Styles That Yield Simulation and Synthesis Mismatches", Don Mills and Clifford Cummings, Boston SNUG 1999
- [Synopsys 02] "Formality User Guide", Version T-2002.09, Synopsys, September 2002
- [Synopsys 03] "VCS User Guide", Version 7.0, Synopsys, January 2003
- [Verplex 03] "Conformal Equivalence Checking Product Suite Reference Manual", Version 4.0, Verplex, 2003