

Solving Verilog X-Issues by Sequentially Comparing a Design with itself.

You'll never trust unix diff again!

Mike Turpin

ARM Ltd, Cambridge, UK

Mike.Turpin@arm.com

ABSTRACT

This paper first introduces a generic methodology to perform sequential equivalence checking, using a property checker rather than a dedicated equivalence checking tool. Sequential equivalence checking itself has many useful applications in the development of an RTL design, e.g. binary/gray-code/one-hot recoding, pipeline retiming, and IP configuration checking.

This paper goes on to describe a novel, yet elegantly simple, solution to a number of X issues in Verilog. The technique sounds like a waste of time: sequentially comparing a design with itself!

However, outputs can be non-equivalent due to dangerous X's lurking inside the design. This approach allows a design to be optimized (via synthesis don't-cares and smaller non-reset flip-flops) without compromising verification due to different semantic interpretations of X.

Table of Contents

1.0	Introduction.....	3
1.1	The Eureka Moment	3
1.2	Terminology.....	3
1.3	Motivation.....	4
2.0	Logical Equivalence Checking (LEC).....	4
2.1	Matching and Comparing Logic Cones	5
2.2	Applications of Logical Equivalence Checking	6
3.0	Sequential Equivalence Checking	7
3.1	Methodology Developed at ARM.....	7
3.2	Applications for Sequential Comparison.....	10
3.3	Complexity of Sequential vs. Logical.....	13
3.4	Commercial Tools.....	15
4.0	Sequential Comparison for X-Analysis	16
4.1	Problems with X	16
4.2	2-State Sequential Semantics of X.....	16
4.3	Sequential Comparison of a Design with itself!	17
4.4	Debugging Non-Equivalent Outputs	19
5.0	Conclusions and Future Directions.....	20
6.0	Acknowledgements.....	20
7.0	References.....	21

Table of Figures

Figure 1	– Combinatorial Comparison of Logic Cones.....	5
Figure 2	– Simple Pipeline Retiming.....	6
Figure 3	- Testbench for Sequential Compare	8
Figure 4	– Basic Proof Sequence	9
Figure 5	- Testbench with Unmatched I/O	10
Figure 6	- FSM recoding that needs sequential comparison.....	11
Figure 7	- Verifying IP Configurations.....	12
Figure 8	– Complex Pipeline Retiming	13
Figure 9	– Logic Cone for req output	14
Figure 10	- Sequential Cone for the Property P: req ##1 ack -> ##1 ~req	14
Figure 11	- Sequentially Comparing a Design with Itself for X-Analysis	18

1.0 Introduction

A Sequential Comparison methodology has been developed at ARM, using assertions and a formal Property Checker rather than a dedicated Logical Equivalence Checking (LEC) tool like Formality or Conformal. The sequential comparison tells you which outputs are exhaustively proven to be equivalent for all possible input sequences. It also tells you which outputs differ, and for every non-equivalent output you get a debug sequence, from reset, in a VCD waveform.

This technique has been successfully used to check RTL rewrites where equivalence is not combinatorial but is sequential, i.e. the designs have very different internal states but their outputs are identical in every clock cycle. In particular, it's been applied to proving equivalence of IP configuration options in order to replace lots of repeated dynamic simulation.

Traditional LEC tools are combinatorial in nature so cannot prove equivalence of designs that have significantly different internal states, and they will never give a debug sequence showing a difference (they can only give a single combinatorial state-vector to show a difference). However, the big advantage of Logical Equivalence Checkers is that they are very fast and should be used when the two designs have the same internal states.

This paper first describes the sequential equivalence checking methodology using a property checker and assertions. It then goes on to describe a new and exciting use of this technique for X analysis.

1.1 The Eureka Moment

The second part of this paper describes a novel and innovative use of sequential equivalence checking that only came about by accident, when describing some built-in sanity checks to a colleague. ARM's *seq_compare* script has some sanity checks to avoid inadvertently comparing the exact same design twice, either the exact same file location or two separate copies that unix diff shows are 100% identical.

I explained to my colleague that these checks are needed to avoid mistakes that could go unnoticed, because such a comparison of the same design with itself must always be equivalent ... or must it? Mid-sentence it occurred to me that a design can indeed be different to itself in a *sequential* comparison, but only if there are any X's inside the design that can affect the output. Turns out that this is a good technique for analyzing whether X-assignments or X-storage in a design are safe. This check can only be done by sequential equivalence checking, and not by a Logical Equivalence Checker (at least in current tools).

1.2 Terminology

This paper uses terminology including:

- LEC Logical Equivalence Checking tools, such as Formality and Conformal
- Assertion Statement about the design, that needs to be verified
- Assumption Constraint on the design environment, that can be used by formal tools
- Property Either an assertion or an assumption

1.3 Motivation

ARM's motivation for sequentially comparing a design with itself includes:

1. Avoiding the dangers of X explained in references [Bening 99], [Foster 03], [Galbi 02] and [Turpin 03]. This is particularly important to an IP provider like ARM, where the RTL will be implemented several times using different synthesis flows. These references explain that X's can cause differences between RTL simulation and real silicon, which are not picked up by design flows that rely on Logical Equivalence Checkers.
2. Improving formal proof results of assertions, which can fail due to reachable X-assignments (or X-storage) in the design. You have to ask the question: *"If I can't prove the design is equivalent to itself, what chance do I have to formally prove other assertions?"*
3. Two new ARM processors are currently under development, and will initially be used in two automotive applications (ABS: Anti-lock Braking System and ESP: Electronic Stability Program) that are safety critical. Each application will use two cores with a simple voting system that will fail-safe if the processors disagree, and reset the processors. ARM does not want these identical processors to disagree due to X-assignments that are minimized differently or non-reset DFFs that power up as different values.

This paper describes how sequential comparison of a design with itself improves upon the automatic formal proofs of reachable X-assignments, and non-reset DFFs, described in reference [Turpin 2003] - which argued that even a reachable X might be safe if it's not read when it is at X. Such reachable-but-safe X's are hard to check manually but can be found using this new technique.

2.0 Logical Equivalence Checking (LEC)

There are many commercially available Logical Equivalence Checking (LEC) tools, including: Cadence's Conformal, Mentor's Formal-Pro and Synopsys' Formality. This section describes how they work, to give important background information for comparing and contrasting against sequential comparison described later in this document.

All of these LEC tools use formal methods to exhaustively compare large designs very quickly e.g. 500k gate designs compared in 20 minutes. This speed is possible due to a divide-and-conquer approach that performs lots of small combinatorial comparisons.

2.1 Matching and Comparing Logic Cones

Figure 1 below illustrates a comparison between RTL and Netlist.

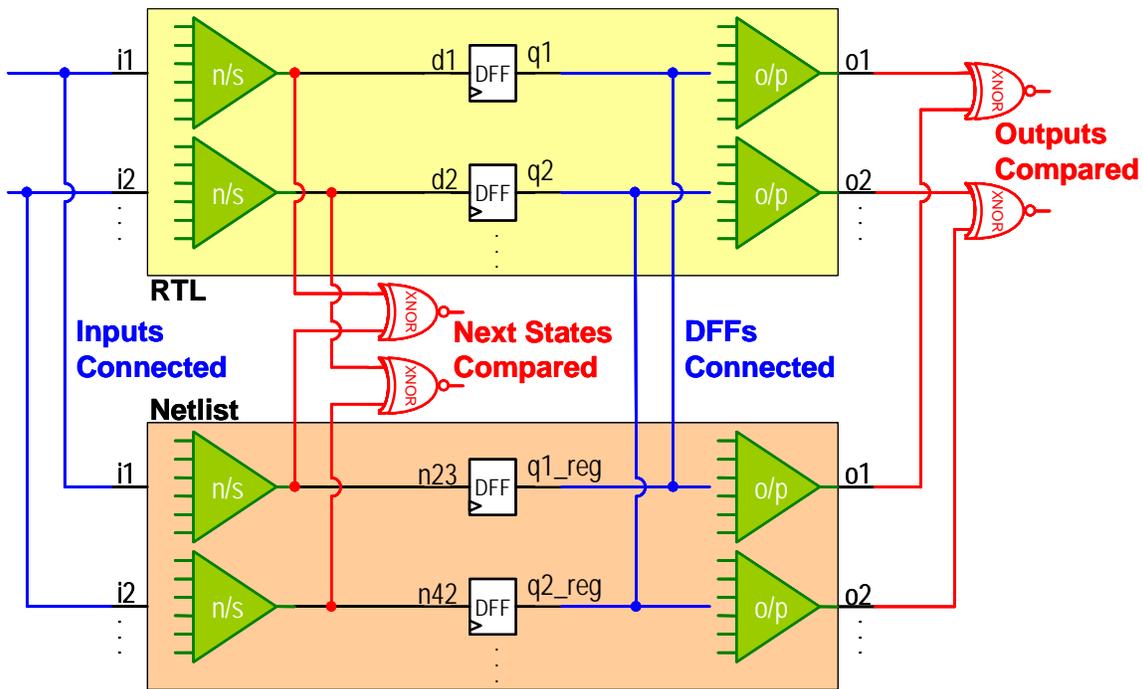


Figure 1 – Combinatorial Comparison of Logic Cones

When comparing two designs, like RTL and netlist in Figure 1 above, a Logical Equivalence Checker must match key points from 3 main groups:

1. Primary Inputs
2. Internal State (DFFs and DLATs)
3. Primary Outputs

The matching itself is performed initially by name (as this is quick) with optional wildcards, but then by signature analysis (of fan-in and fan-out). Once matched, a Logical Equivalence Checker compares all logic cones that fall in one of two groups:

1. Output functions
2. Next-state functions (e.g. driving d-input to a DFF)

If all key points are match, and all logic cones are equivalent, then the designs are formally verified as being 100% equivalent.

Solving Verilog X-issues by Sequentially Comparing a design with itself!

2.2 Applications of Logical Equivalence Checking

This section describes several applications of Logical Equivalence Checking.

2.2.1 RTL vs. Netlist Comparisons

This is the most popular application of Logical Equivalence Checking, for ensuring that synthesis or manual changes, have correctly implemented the RTL. It is far faster than comparing RTL simulation vs. Netlist simulation, and is an exhaustive verification method.

2.2.2 RTL Verilog vs. RTL VHDL

This is typically performed to check results from an automated HDL translator.

2.2.3 Simple RTL Retiming

One common RTL modification is to retime logic in a pipeline, to speed up the clock frequency of the design. This is illustrated in Figure 2 below, where the logic between stage 2 and stage 3 is far longer than anywhere else and contains all the critical paths of the design. The original RTL is modified by moving combinatorial logic between stages – leading to balanced logic cones with similar critical paths.

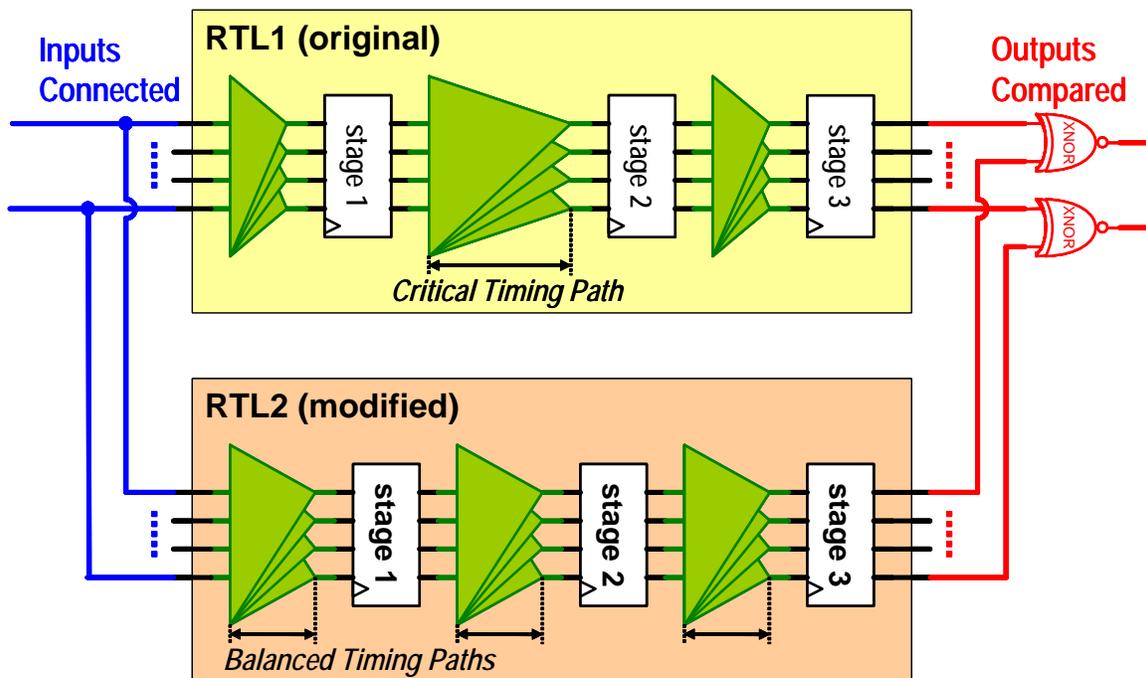


Figure 2 – Simple Pipeline Retiming

If the pipeline retiming is simple enough, it's possible to check the modification with a Logical Equivalence Checking (LEC) tool. The simplest form of retimed logic is moving an inverter from one stage to another, which only requires the affected DFFs to be matched *inverted* by the LEC tool.

Solving Verilog X-issues by Sequentially Comparing a design with itself!

LEC tools can also be used to compare more complex retimed logic by turning the comparison into a combinatorial one, using either of the simple approaches below:

1. **Transparency:** Each DFF is replaced by a wire
2. **Normalization:** DFF stages are pushed together, through the combinatorial logic

The problem with the first approach is that you can get a false positive result i.e. two designs with different numbers of DFF stages can be declared to be equivalent. The second approach is better, as it will only succeed if the numbers of DFF stages are the same. Formality takes the second approach, and can also handle local feedback i.e. a DFF preceded by a multiplexer. See section 3.2.3 for more complex retiming that normally requires a sequential approach.

3.0 Sequential Equivalence Checking

Logical Equivalence Checking is fast because it compares logic cones, as illustrated by Figure 1. However, this approach means that a wide range of RTL modifications cannot be compared – where the structure of the internal states differs significantly. This section describes a methodology for performing *Sequential* Equivalence Checking.

3.1 Methodology Developed at ARM

The sequential comparison methodology developed at ARM takes two designs and then:

1. Creates a test-bench that instantiates each design, connecting inputs and comparing outputs.
2. Adds OVL assertions that all `EQUIV_*` Boolean outputs are always true.
3. Runs through a *formal-proof-sequence* flow, also developed at ARM, which automatically proves or fails the assertions.

Step 3 avoids false-negative proof issues, where a property is failed from an unreachable design state, by applying a sequence of different proofs with increasing probability of avoiding unreachable states (at the cost of increased run times). It produces a report of how many outputs have been exhaustively proven to be equivalent, and which outputs differ - giving an input sequence from reset that demonstrates the difference. See section 3.1.3 for more details.

3.1.1 Basic Testbench

Figure 3 below shows the basic structure of the testbench, which takes a black-box view of the designs being compared. This is different to a Logical Equivalence Checker, as illustrated in Figure 1, which needs to compare the internal state of each design.

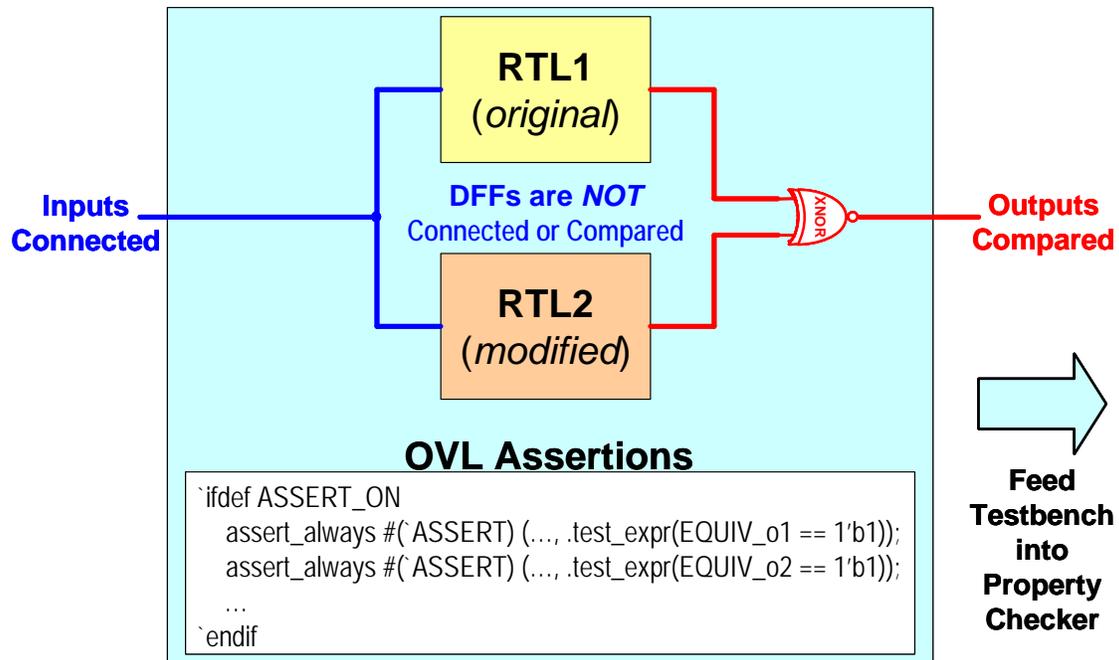


Figure 3 - Testbench for Sequential Compare

This basic testbench is created with a simple script, which can be invoked as follows:

```
seq_compare RTL1.v RTL2.v
```

By default, no formal proofs are run. Instead, the testbench is created and some setup files are created for RTL compilation and formal proofs (which can be modified as appropriate). This two-step approach also allows the OVL assertions themselves to be modified if required, or input assumptions to be added.

3.1.2 Assertions to Prove Equivalence

The testbench of Figure 3 is ideal for property checking, as it's now a single design with a set of assertions to prove that all outputs are equivalent under *all* input sequences. It's also possible to add extra OVL assertions that constrain the input sequences, to prove that all outputs are equivalent under all *legal* input sequences. ARM has also developed methodology to extract and formally verify OVL assertions, which is also used for the sequential comparison methodology.

3.1.3 Formal Proof Sequence

All of the assertions are pushed through an automated sequence of proofs, a simplified version of which is illustrated in Figure 4. This approach avoids false negative issues (failure from an unreachable design state) and tries to maximize the number of exhaustive proofs.

It will only report a failing assertion if the failure is from reset – in which case it shows a debug sequence (dumped as a VCD file). Assertions that are neither proven exhaustively, nor fail from reset, are classed as partial proofs - which can often be turned into exhaustive proofs by

increasing the *proof effort* (that controls how many exhaustive proof methods are tried in the sequence - the higher the effort, the more complex the proofs).

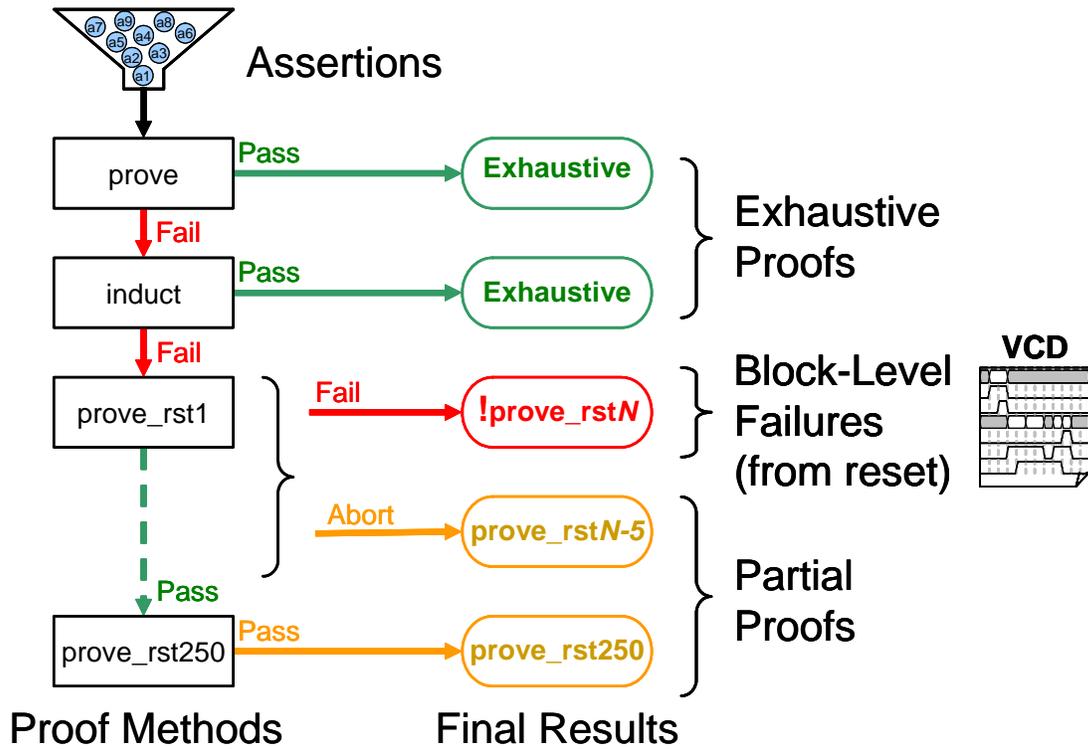


Figure 4 – Basic Proof Sequence

The square boxes on the left column of Figure 4 show examples of the *proof methods* used, e.g. “induct” is an exhaustive proof that uses mathematical induction and “prove_rst250” is a finite proof that holds for any input sequence within 250 cycles after reset. The arrows indicate proof results, with many results leading to the next proof method in the sequence and only conclusive results being logged as a final proof result.

For debug, traditional LEC tools produce schematics or fan-in vectors (possibly showing unreachable values). In the case of sequential comparison, a failing assertion indicates a difference between two corresponding outputs which can be debugged via the VCD trace.

3.1.4 Testbench for Designs with Differing I/O

Figure 5 below shows how designs with different numbers of inputs and outputs are compared. The first design (RTL1) is considered as the original design, a golden reference model against which all changes are compared. Consequently, all I/O from RTL1 will be I/O for the testbench (with extra outputs XNOR'd with Z, which will always fail as Z can also be set to either 0 or 1).

Solving Verilog X-issues by Sequentially Comparing a design with itself!

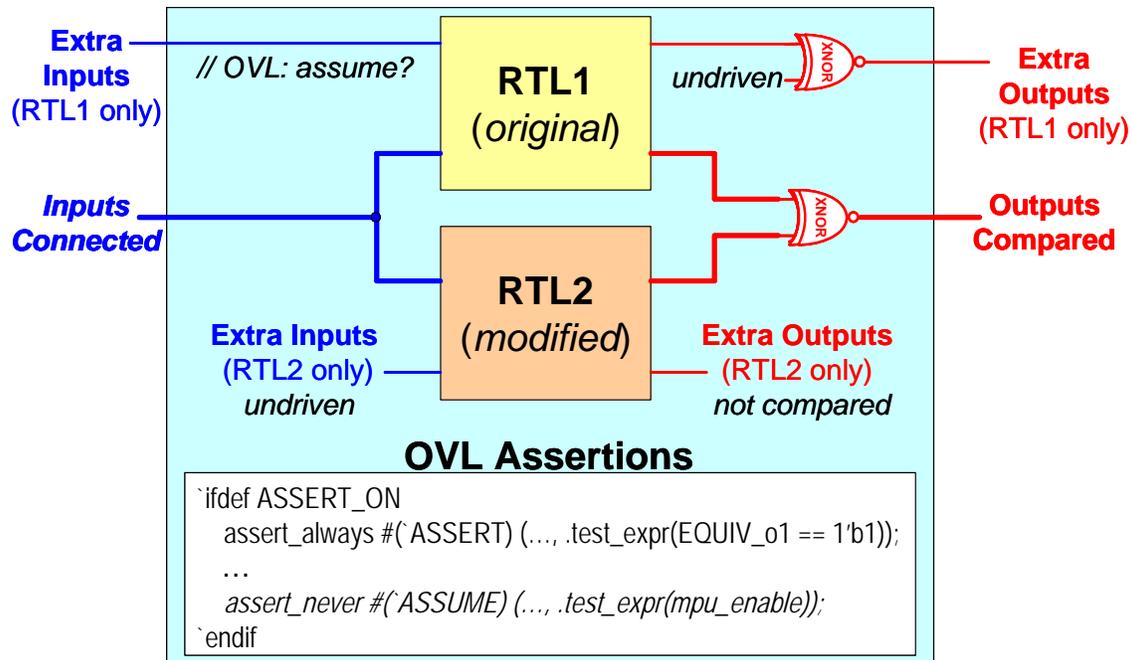


Figure 5 - Testbench with Unmatched I/O

Any additional RTL1 inputs can be constrained by additional OVL assertions configured as assumptions for formal proofs (by setting their `options` parameter to 1).

3.2 Applications for Sequential Comparison

This section describes several applications for sequential comparison.

3.2.1 State Machine Encodings

Logical Equivalence Checkers can handle some simple state machine recodings, but only if the number of states are identical in the two designs (in which case the LEC tool can match state bits in different ways, perhaps with some matched *inverted*).

Formality can handle some comparisons where the number of state bits have changed, e.g. a binary FSM rewritten to be one-hot encoded. However, no LEC tool can prove equivalence of two designs where the numbers of states differ, e.g. binary FSM rewritten to be gray-coded.

Figure 6 below shows a state machine recoding where two of the states have been expanded in order to achieve gray-coding.

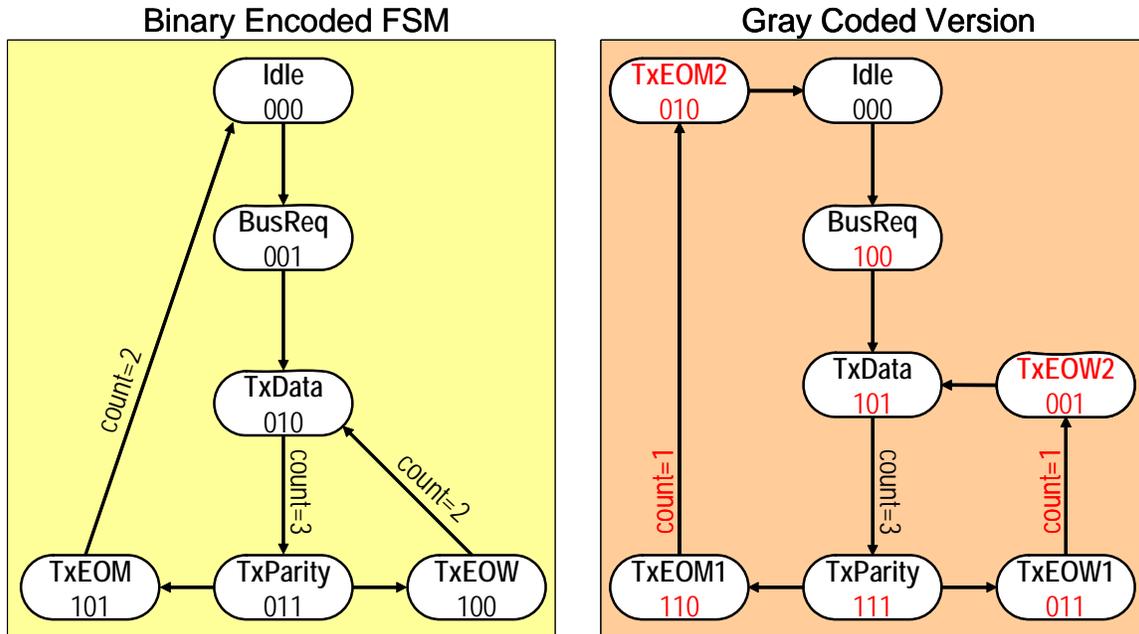


Figure 6 - FSM recoding that needs sequential comparison

LEC tools might be able to match the 3 state bits in each state machine of Figure 6, but will show combinatorial differences as the second design contains two additional states.

A *sequential* comparison can show equivalence if all outputs sequences are the same (for all possible input sequences). If the binary encoded FSM in Figure 6 had been recoded as a one-hot FSM, LEC tools wouldn't even have been able to match all DFFs (3-bits in the binary version vs. 6-bits in the one-hot version).

3.2.2 Checking Configurable IP

The sequential comparison methodology described in this paper was originally created at ARM for checking configurable IP, and has found a number of differences during the development phases of several designs. ARM processors have several configuration options, including Cache size (or Cache-less), TCM (Tightly Coupled Memory) size, number of breakpoints and watchpoints, inclusion of DMA (Direct Memory Access), inclusion of an MPU (Memory Protection Unit). For every option, the amount of dynamic verification could be doubled – which rapidly becomes a big overhead. Instead, formal verification can be used to efficiently replace billions of simulation cycles.

Consider the option of including an MPU. If no MPU is included in a particular implementation, then a default “empty” MPU is put in place which has a small amount of internal state to give appropriate empty-MPU behaviour. For a design with an MPU included, there will also be an enable input to dynamically switch between MPU-on and MPU-off. The behaviour of an empty-MPU is usually the same as MPU-off, so an equivalence check can be performed with the design containing the MPU having an extra input assumption that the MPU enable input is low (off).

Solving Verilog X-issues by Sequentially Comparing a design with itself!

IP configuration comparisons often require a sequential equivalence check, as illustrated in Figure 7 below. Some IP configuration comparisons can be done by LEC tools, or by a mixture of LEC and Sequential comparison (i.e. use Sequential to check outputs that LEC says are different).

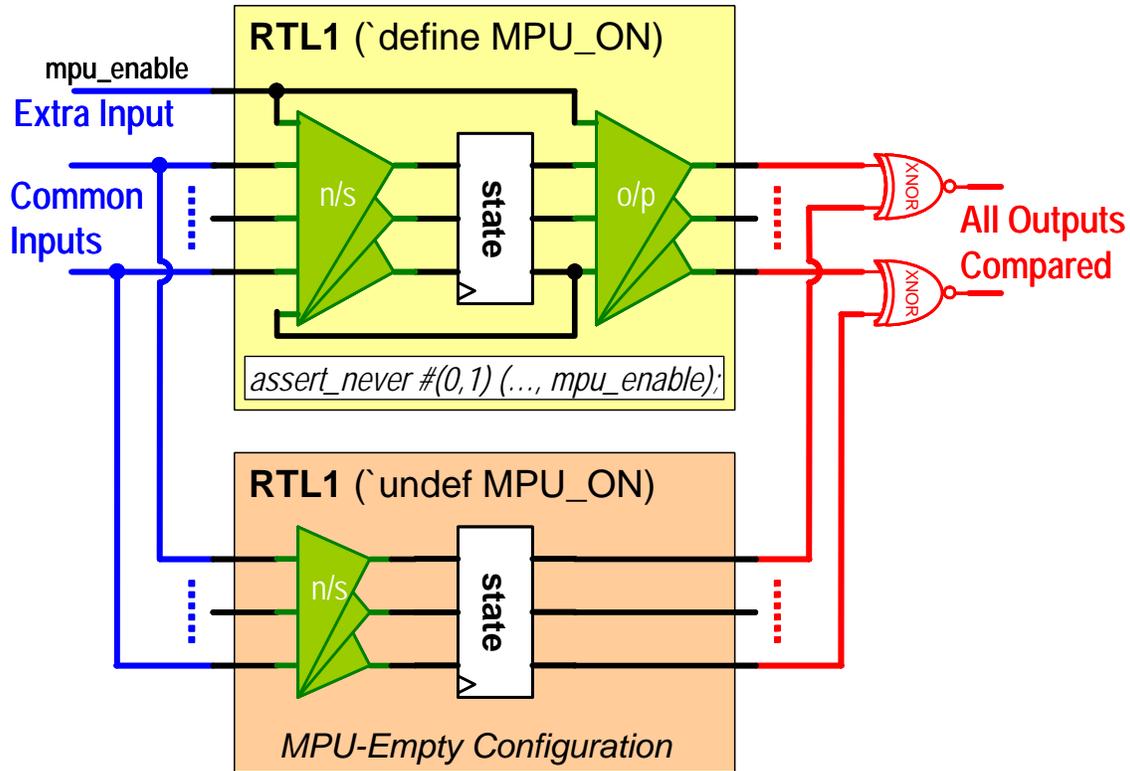


Figure 7 - Verifying IP Configurations

Even if all DFFs are equivalent in LEC, a Sequential comparison can sometimes show more differences than LEC in the Outputs. These extra non-equivalent outputs are due to dangerous Xs in the design (explained in section 4.0).

3.2.3 Complex Retiming

If a retimed pipeline has feedback or forwarding paths, as illustrated in Figure 8 below, then Logical Equivalence Checkers cannot generally prove equivalence. One exception is that Formality can handle a tight local feedback with a DFF and a multiplexer directly feeding the Q output back to the D input.

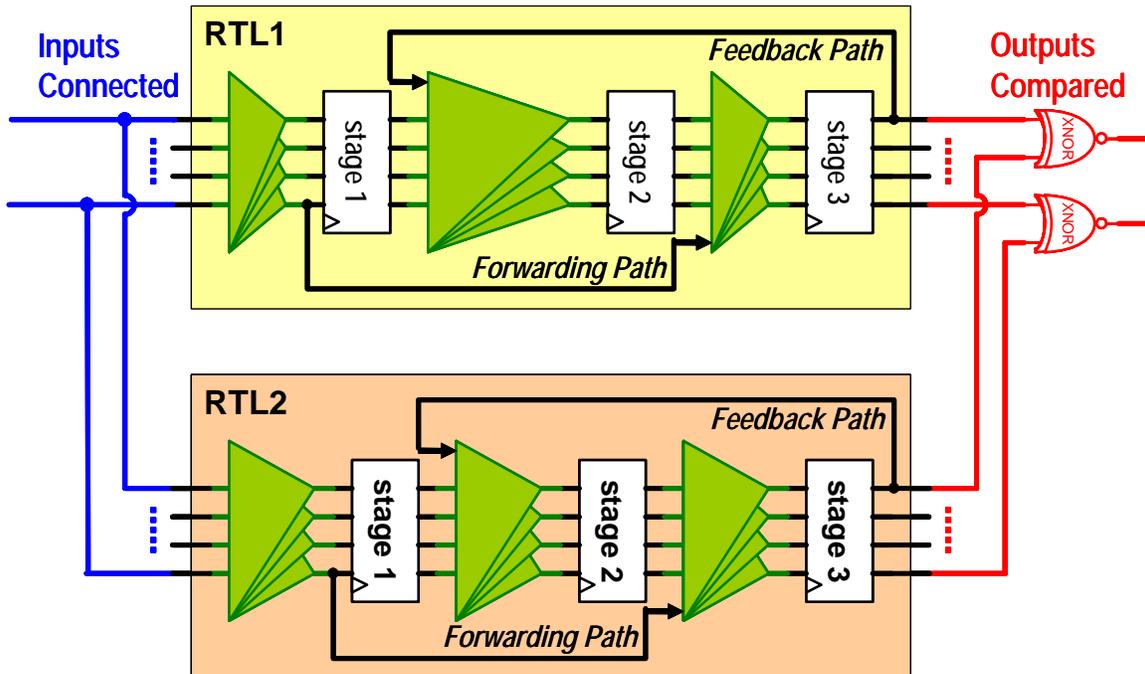


Figure 8 – Complex Pipeline Retiming

Forwarding paths can be checked by LEC tools that simply replace DFFs by wires (see section 2.2.3) but this approach can miss bugs (only shown by sequential analysis).

3.3 Complexity of Sequential vs. Logical

This section describes the different complexities of Logical Equivalence Checking and Sequential Comparison, and aims to help quantify the complexity of formal verification in general.

3.3.1 Logical and Sequential Cones

A combinatorial cone of logic is illustrated by Figure 9 below, which shows a request output called `req`. Fan-in to this cone can be design inputs and internal state (stored in DFFs) and complexity also depends upon the functionality of the signal itself (measured by the gate-count of its cone). A Logical Equivalence Checker has to consider 2^{I+S} permutations to the fan-in of such a cone (where I is the number of inputs and S is the number of states).

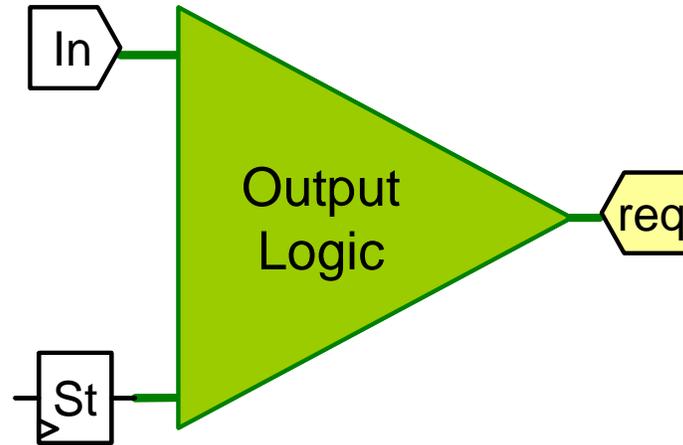


Figure 9 – Logic Cone for req output

A property can be represented by a sequential cone, as illustrated by Figure 10 below. For the property in Figure 10 there are 3 copies of the logic cone for the req output (for the 3 timesteps in the property). Inputs and internal states that fan-in to each cone can differ, which increases the complexity of the proof. Figure 10 is in fact a simplified view, as a sequential proof will also need to consider additional cones – those driving the state points. Complexity increases with the length of the property i.e. number of clock cycles, and in the proof method used e.g. prove_rst10 will add at least another 10 cycles to the start of the property (more if the reset sequence needs a number of cycles to reset all possible DFFs).

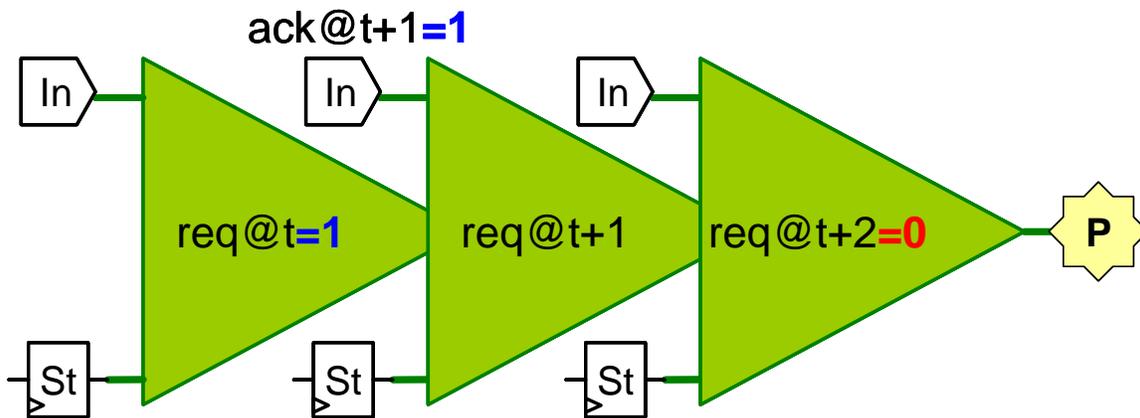


Figure 10 - Sequential Cone for the Property P: req ##1 ack |-> ##1 ~req

Sequential proofs are much more complex than combinatorial proofs, as illustrated by Figure 9 and Figure 10, and typically only applied at the block level. An amusing way to help remember this difference is: “A combinatorial proof only needs to consider cones but a sequential proof needs to consider the entire Christmas tree!”

Sequential proofs can have even greater sequential depths than the property itself, e.g. an inductive step adds an extra cycle because it’s proving: $P \mid\text{->} \#\#1 P$.

Solving Verilog X-issues by Sequentially Comparing a design with itself!

3.3.2 How Big is Exhaustive?

The complexity of verification that formal methods can cope with is staggering, and this section aims to help the reader quantify this complexity. Consider simulating a typical CPU design:

- 500k gates, 20k DFFs, 500 inputs
- 70 billion simulation cycles, running on 200 linux boxes for a week
- **How big: 2^{36} cycles**

Consider *exhaustively* verifying the entire design:

- Input sequences: $2^{(\text{inputs}+\text{state})} = 2^{20500}$ cycles
- What about X's: 2^{15000} (5,000 X-assignments + 10,000 non-reset DFFs)
- **How big: 2^{20500} cycles** (2^{15000} combinations of X is not significant here!)

That's a big number! So big in fact that it's hard to visualize unless given with some context:

- Cycles to simulate the 500k design: 2^{36} (70 billion simulation cycles)
- Cycles to *exhaustively* verify 32-bit adder: 2^{64} (18 billion billion)
- Number of stars in universe: 2^{70} (10^{21})
- Number of atoms in the universe: 2^{260} (10^{78})
- Possible X combinations in 500k design: 2^{15000} ($10^{4515} \times 3$)
- Cycles to *exhaustively* verify 500k design: 2^{20500} (10^{6171})

So, to say that formal verification is astronomical would not be an exaggeration – it would in fact be an understatement.

3.3.3 Increasing Capacity and Speed

It is possible to increase the capacity and speed of sequential comparisons in a number of ways including:

- **Hierarchical Comparison:** A divide-and-conquer approach where smaller blocks are verified before the modules that instance them. This is equally applicable to Logical Equivalence Checkers but is not usually required.
- **Hybrid Logical/Sequential:** Two designs will typically differ sequentially by only a small amount – the majority of the comparison can be done combinatorially as per existing LEC tools. One approach would be to first try and logically compare matching DFFs and remove any that differ (leaving the others in the following sequential comparison).

3.4 Commercial Tools

As described above, some LEC tools (including Cadence Conformal and Synopsys Formality) provide some limited support for sequential changes to a design. Up until recently there were no commercial tools that explicitly offered full sequential equivalence checking, but such a tool is

Solving Verilog X-issues by Sequentially Comparing a design with itself!

now available from a company called Calypto. In addition to providing RTL vs. RTL comparisons, Calypto allows SystemC vs. RTL comparisons.

Of course, it's also possible to apply the sequential comparisons described in this paper on a number of commercially available property checkers including:

- Averant's Solidify
- Cadence's Static Verifier
- Jasper Design Automation's Jasper-Gold
- Mentor's 0-In Confirm
- Synopsys' Magellan

4.0 Sequential Comparison for X-Analysis

This section describes how sequential equivalence checking can be used to find out which X's are dangerous.

4.1 Problems with X

There are different semantics of X, which can be described as follows:

- **Simulation** semantics of X as "unknown". Can give optimistic and pessimistic results.
- **Synthesis** semantics of X as "don't-care", allowing it to be minimized.
- **2-State Sequential** semantics of X can be either 0 or 1 (modeled by some formal tools)

The fact that there are different semantics compromises the results of RTL simulation and Logical Equivalence Checking, as described in reference [Turpin 2003] - which gave some automated property checking techniques for finding potentially dangerous X's from two sources:

- X-assignments that are reachable
- X-storage in non-reset DFFs

However, it's not always clear if reachable X's are dangerous. [Turpin 2003] argued that even a reachable X might be safe if it's not read when it is at X (e.g. an X stored in a DFF is safe if it's never read). Distinguishing reachable-but-safe X's can be very difficult.

4.2 2-State Sequential Semantics of X

Formal tools do not have to adhere to the semantics of X in RTL Verilog, as described in [IEEE 95], but can instead fail the verification by stressing the design with X=0 and X=1 settings for *every* X (a much better representation of any possible silicon). Unlike Logical Equivalence Checkers, formal property checkers consider the *sequential* behavior of a design – allowing them to track possible values of X's through internal registers. This is termed *2-State Sequential* in this paper (and in [Turpin 2003]), but can also be thought of as the *Silicon Semantics* of X as it's a better reflection of what Silicon would actually do (at least from a *static* viewpoint).

Solving Verilog X-issues by Sequentially Comparing a design with itself!

Strict *2-State Sequential* semantics can be performed by many formal property checkers including those described in section 3.4 (although Synopsys' Magellan also considers *Unknown* simulation semantics via VCS)

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. 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.

The rest of this section assumes that the property checker being used for sequential comparison is using *2-State Sequential* semantics of X.

4.3 Sequential Comparison of a Design with itself!

There are some built-in sanity checks in ARM's `seq_compare` script to avoid inadvertently comparing the exact same design twice, either the same file location or an exact copy. However, *sequentially* comparing a design with itself is not a waste of time as it can show a difference if:

- An X-assignment can affect an output
- An X stored in a non-reset DFF can affect an output

Comparing a design with itself would pass in `unix diff`. It would also pass in a Logical Equivalence Checker which considers both 0 and 1 values of every X but, crucially, matches pairs of X's in the two designs and ties them together. Note that Formality does not match pairs of X-assignments (it treats them as separate unknowns), but it does match DFFs (which can store Xs, e.g. on power-up).

Comparing a design with itself would not necessarily pass in a sequential comparison as the DFFs (and X-assignments) are not matched and so the X can be 0 in one copy and 1 in the other. This approach is illustrated in Figure 11 below. The X's themselves can be from X-assignments in the logic cones driving the DFF, or from X's in non-reset DFFs (e.g. during reset). The formal property checker will try every combination of X, differently in each design copy, to show a non-equivalent output.

Sequential equivalence checking provides a simple and elegant solution to finding dangerous X's, as it can determine if potentially dangerous X's can affect the block's outputs. Conversely, it can be used to increase the number of non-dangerous X's in order to produce an optimal design (X assignments for synthesis don't-cares and smaller non-reset DFFs).

Solving Verilog X-issues by Sequentially Comparing a design with itself!

4.3.1 Usage

ARM's seq_compare script is run for X-analysis as follows:

```
seq_compare -x RTL1.v
```

The -x flag will override the sanity-checks and do a formal comparison of RTL1 with itself. The formal property checker will try every combination of X differently in each design copy, to show a non-equivalent output.

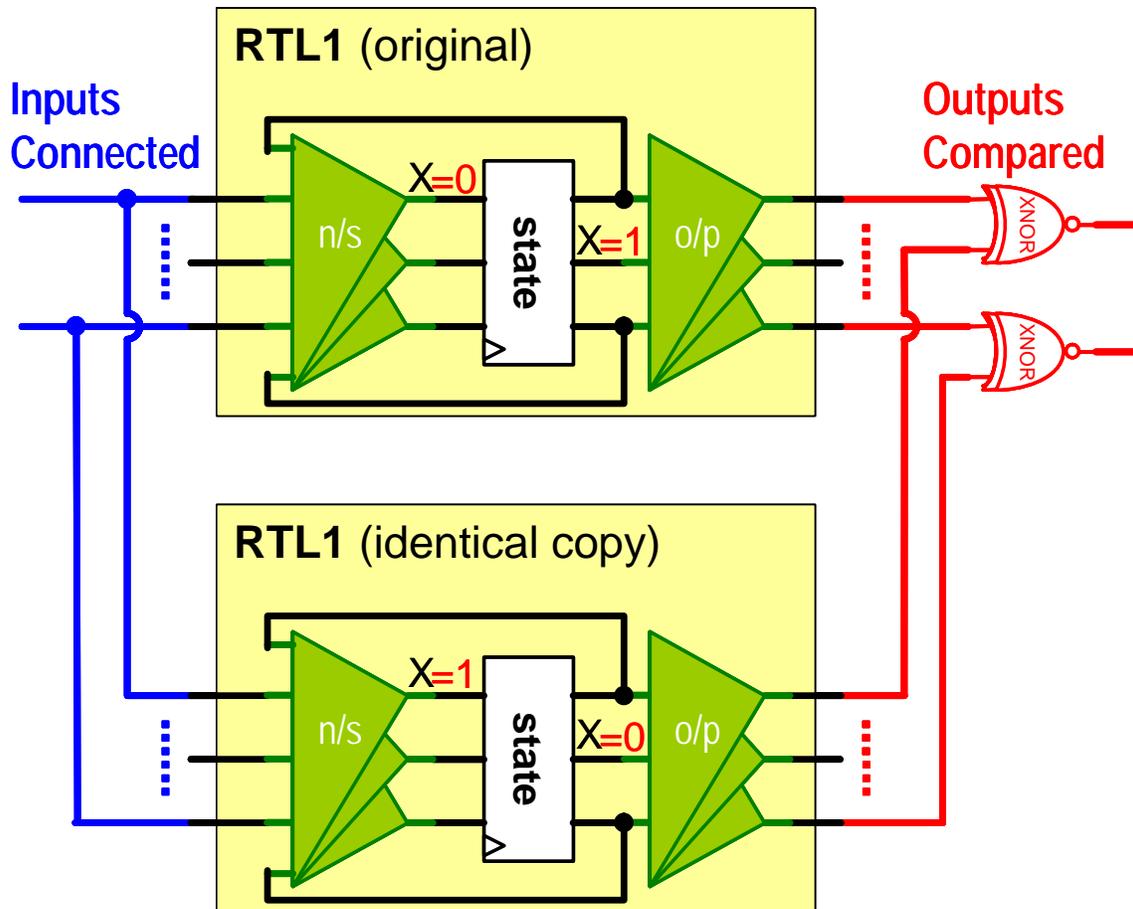


Figure 11 - Sequentially Comparing a Design with Itself for X-Analysis

This comparison is illustrated in Figure 11 above. As DFFs (and X-assignments) are not matched, a property checker is free to choose different values of X to fail an equivalent output assertion.

4.3.2 Additional Bug Hunting

If your comparison has a number of partial proofs, e.g. proven for 50 cycles from reset but no further (due to timeouts) then you can try bug-hunting by setting all X-assignments to 0 (and resetting all non-reset DFFs to 0) in one copy and to 1 in the other. A pass on such a mapping is non-exhaustive, but a failure is real.

4.4 Debugging Non-Equivalent Outputs

This section describes some techniques to debug non-equivalent outputs for a sequential comparison of a design with itself.

4.4.1 Identifying Reachable and Deadcode X-assignments

[Turpin 2003] describes some automated property checking techniques that tell you which X's are *deadcode* (i.e. unreachable) and which are *reachable*. Any X-assignments reported as deadcode can be left in the design (to improve minimization during synthesis). Any X-assignments that are reported as reachable are potential reasons why the design is not equivalent to itself – removing them could make more outputs equivalent.

4.4.2 Identifying Reset and Non-Reset DFFs

[Turpin 2003] describes an automated property checking technique that tells you which DFFs are not reset – each one a potential source of a dangerous X. Recently this has been improved by a new technique that provides information about:

1. Which DFFs are not reset (potential source of X).
2. Which DFFs are reset to 1 (as opposed to 0). Can indicate bugs or one-hot vectors.
3. Which DFFs are *directly* reset, asynchronously or synchronously.
4. Which DFFs are *subsequently* reset, due to pipelining from directly reset DFFs.
5. How long the reset should be applied to get all possible resets.

Point 4 above is particularly interesting as it avoids adding reset lines to DFFs that are reset as part of a pipeline from directly reset DFFs. You just need to ensure that the reset is asserted for enough cycles to reset all possible DFFs (see point 5 above).

Achieving points 4 and 5 above requires an iterative proof sequence, which can be computationally expensive as all non-reset DFFs will be tried as the reset depth increases. To improve efficiency, vector-wide reset proofs are performed as described in Table 1 below (for asynchronous resets). Half of the proof results in Table 1, the *None* and *All* results, are conclusive for all bits in the vector and proofs for individual DFFs are only required for the *Some* results. Using vector-wide reset proofs means that large non-reset datapath registers don't take long to prove. Another efficiency is that at the start of each reset depth (required for point 5 above) a vector is created for all remaining DFFs, so on the last step only two proofs are performed (showing that no remaining DFFs can be reset-to-0 or reset-to-1).

Asynchronous Reset Assertion (in SVA)		Proven	Failed
<code>~reset_n</code>	<code>-> &(dffs[31:0]) == 1'b0;</code>	Some reset-to-0	None reset-to-0
<code>~reset_n</code>	<code>-> &(dffs[31:0]) == 1'b1;</code>	All reset-to-1	Some reset-to-0, or Some still-at-X
<code>~reset_n</code>	<code>-> (dffs[31:0]) == 1'b0;</code>	All reset-to-0	Some reset-to-1, or Some still-at-X
<code>~reset_n</code>	<code>-> (dffs[31:0]) == 1'b1;</code>	Some reset-to-1	None reset-to-1

Table 1 Vector Wide Proofs

Solving Verilog X-issues by Sequentially Comparing a design with itself!

4.4.3 Examining Fan-in that drives the Cone

Given a non-equivalent output, you can examine the fan-in (inputs and state) that drives its cone of logic – to see which X-assignments and/or non-reset DFFs can affect the cone. The user can then experiment with removing such X's and re-running equivalence.

4.4.4 Modified Comparison

Given a data output, e.g. of 64-bits wide, it may be that X's can occur on this output for certain clock cycles but should not be X when a *valid* signal is high. The OVL assertions for comparison can be modified to factor in this enabling term e.g. using `assert_implication`.

4.4.5 Matching DFFs for Software Initialization

It might sometimes be necessary to match a small number of non-reset DFFs in order to avoid failures due to X in the hardware that are prevented by software. Consider the register bank of a processor, which is not reset as they are datapath registers (so power up with X's stored in them). If any instruction was allowed after power up, then a load from a register could be performed before it has been initialized – causing a difference when comparing the design with itself. Matching such DFFs is better than forcing them to be reset as it allows all permutations of X to be checked, with a constraint that they will be the same in both copies of the design being compared.

5.0 Conclusions and Future Directions

This paper has described how a property checker can be used to perform sequential comparisons. It has also given some useful applications of this technique, which cannot be compared using traditional Logical Equivalency Checkers.

This paper has also described a novel solution to the problem of dangerous X semantics, by sequentially comparing a design with itself! This technique is an improvement to the approaches described in [Turpin 2003]. It enables designs to be efficiently implemented, with smaller non-reset DFFs or X-assignments used to improve minimization during synthesis, without compromising safety. It can also improve the results of formal property checking in general, which can fail due to X's in a design.

ARM will continue to use these techniques in its IP, and develop this methodology further (maybe a hybrid solution of Sequential and Combinatorial comparisons) until they become widely available in EDA tools.

6.0 Acknowledgements

I would like to thank Mike Bartley, from the SNUG Technical Committee, for doing a great job of reviewing this paper. From Synopsys I'd like to thank Lisa McIlwain and Phil Moorby for their comments on this paper, and Dan Benua and Alessandro Fasan for their comments on this technique. I'd also like to thank Jeremy Sonander from Saros Technology, along with Ramin Hojati and Adrian Isles from Averant, for their support in implementing the techniques described

Solving Verilog X-issues by Sequentially Comparing a design with itself!

in this paper. I'd also like to thank several colleagues at ARM for giving feedback on this technique, in particular Simon Craske and Samin Ishtiaq.

7.0 References

- [Bening 99] "A Two-State Methodology for RTL Logic Simulation", Lionel Bening, DAC 1999
- [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
- [Turpin 03] "The Dangers of Living with an X (bugs hidden in your Verilog)", Mike Turpin, Boston SNUG 2003