Finding Reset Nondeterminism in RTL Designs – Scalable X-Analysis Methodology and Case Study

Hong-Zu Chou[†], Haiqian Yu[*], Kai-Hui Chang[‡], Dylan Dobbyn[*], and Sy-Yen Kuo[†]
[†]Electrical Engineering Department, National Taiwan University, Taipei, Taiwan
[*]Teradyne Inc., North Reading, MA, USA
[‡]Avery Design Systems, Inc., Andover, MA, USA
sykuo@cc.ee.ntu.edu.tw

***Abstract*—Due to increases in design complexity, routing a reset signal to all registers is becoming more difficult. One way to solve this problem is to reset only certain registers and rely on a software initialization sequence to reset other registers. This approach, however, may allow unknown values (also called X-values) in uninitialized registers to leak to other registers, leaving the design in a nondeterministic state. Although logic simulation can find some X-problems, it is not accurate and may miss bugs. A recent approach based on symbolic simulation can handle Xs accurately; however, it is not scalable. In this work we analyze the characteristics of X-problems and propose a methodology that leverages the accuracy of formal X-analysis and can scale to large designs. This is achieved by our novel partitioning techniques and the intelligent use of waveforms as stimulus. We applied our methodology to an industrial design and successfully identified several Xs unknown to the designers, including three real bugs, demonstrating the effectiveness of our approach.**

I. Introduction

Due to the dramatic increase in design complexity and the miniaturization of transistors, routing the reset signal to all registers is becoming more difficult [9]. A commonly used approach to address this problem is to reset important registers only and rely on software sequences to initialize the rest of the registers. Some registers, such as those used to hold intermediate values in an arithmetic unit, may even be left uninitialized. Those uninitialized registers will have unknown values, often denoted as Xs, which can be either 0 or 1 in real hardware. If those Xs are not handled correctly, they may leak to important registers and leave the design in a nondeterministic state, causing serious problems. Therefore, it is important to to find those Xs and make sure they do not affect design correctness. Because Xs have many other applications like enabling synthesis tools to perform better optimizations [3], [10], such X-related problems will become more serious in the future.

Since engineers design circuits at the Register Transfer Level (RTL), it is desirable to find and fix all X-problems at this level. The most commonly used method to find X-problems in an RTL design is to set uninitialized registers to Xs and then perform logic simulation to check whether those Xs will propagate to important registers. Although this approach is fast and easy to implement, the result is often inaccurate due to X-optimism and X-pessimism in logic simulation. Another method is to run gate-level simulation and compare the results with RTL simulation. However, gate-level simulation is much slower. Furthermore, once a bug is found, identifying the root cause of the problem in the RTL code may be challenging [4]. To address this problem, Chou et al. [5] proposed to use symbolic simulation and SAT solvers to find X-problems at the RTL. Although this approach is effective, scalability remains an issue due to the formal nature of their analysis and the lack of suitable methodologies to apply their method. In addition to the lack of appropriate X-analysis techniques, there are several practical issues in X-analysis that remain unaddressed in existing literature. For example, little has been mentioned on how to determine what to check in X-analysis. Since Xs can exist in the design, reporting all the Xs in registers will undoubtedly result in too many false alarms. Asking engineers to write an assertions for every register that needs to be checked may also be infeasible because the number of such registers may be large.

To address these issues, we analyzed the characteristics of X-problems and proposed an novel methodology called "eXact". We adopted Chou's formal X-analysis method due to its accuracy, and we applied several partitioning techniques to increase its scalability. Since our partitioning forms under-constrained inputs to the block under test, no bug will be missed. Our second innovation is an approach that only reports X-problems missed by logic simulation, i.e., those that cannot be observed in waveforms. In this way, we can reduce X-analysis effort and let designers focus on real problems. We applied our methodologies on a six-million gate industrial design and successfully identified several X-problems unknown to the designers – including three real bugs. This case study shows that our techniques and methodologies are useful and can be applied to practical designs.

The rest of the paper is organized as follows. In Section II, we briefly describe some circuit reset methodologies and their X-problems. Analysis of X-problems and current solutions are provided in Section III. In Section IV, we describe our eXact methodology for finding X-problems in real-life industrial designs. Case studies are shown in Section V, and Section VI concludes this paper.

II. Background

In this section, we first briefly describe constrained-random simulation and the concept of under-constrained and over-constrained testbenches since they are important for understanding the benefits and limitations of our eXact methodology. We then explain X-optimism and X-pessimism problems in logic simulation.

A. Constrained-Random Simulation

Constrained-random simulation is a design verification method that automatically generates random but legal patterns. The advantage of this method is that it can generate input

978-3-9810801-6-2/DATE10 © 2010 EDAA

scenarios that designers did not think of, thus increasing verification quality. However, writing constraints for designs under verification can be challenging because the testbench may be over-constrained or under-constrained. Under-constrained testbenches produce illegal inputs that may result in false alarms, leading to the analysis of irrelevant states and increasing verification effort. Over-constrained inputs cannot cover the whole valid input space and may miss bugs, thus reducing verification quality. Since it is often difficult to write testbenches that are precisely-constrained, proper methodologies must be used to ensure verification correctness.

B. X-Pessimism and X-Optimism in Logic Simulation

Logic simulation is the most widely-used verification technique and may be able to find some X-problems in a design. However, X-handling in logic simulation is often inaccurate due to X-optimism and X-pessimism [2], [10]. Take the simple code shown in Fig. 1(a) as an example, if "*a*" is 1'bx, "*out*" can be either "*b*" or "*c*". However, due to X-optimism in logic simulation algorithms, only one branch is considered and "*out*" will be equal to "*c*". Fig. 1(b) shows another example where signal "*out*" should not be affected by "*a*" but is assigned X erroneously due to X-pessimism.

(a) **X-optimism**	(b) **X-pessimism**

a = 1'bx;
if (*a*) *out* = *b*;
else *out* = *c*;
result: *out* = *c*;
	a = 1'bx; *b* = 1'b1; *c* = 1'b1;
	out = (*a* & b) | (∼*a* & *c*);
	
	result: *out* = 1'bx;

Fig. 1. A simple example to show X-optimism and X-pessimism problems in logic simulation.

III. Analytical Study of X-Problems and Current Solutions

In this section we first analyze the characteristics of design nondeterminism problems (X-problems) and then describe current solutions that try to solve the problem.

A. Characteristics of X-Problems

Theoretically, Chou's [5] formal X-analysis method can find all the X-problems in a circuit if used correctly. However, it is highly unlikely that one can symbolically simulate a circuit for thousands of initialization cycles to obtain the correct analysis. In order to devise practical solutions for the design-nondeterminism problem, we performed an analytical study on X-problems in RTL designs. We found that such problems have the following characteristics:

(1) The source of Xs is often localized and can be traced back to just an if-then block or a case statement. As a result, once the bug is found, fixing it is often simple. On the other hand, the fact that Xs are highly localized also means they can be easily masked in logic simulation, making their detection much more difficult by RTL logic simulation.

(2) When an X does exist in a design, it tends to propagate out and affect many other registers. The positive side of this characteristic is that one may be able to catch X-problems by observing just a few registers. However, since there may be Xs everywhere at that point, finding the root cause of the problem may be extremely challenging.

(3) In a design where Xs are allowed after initialization, it is often difficult for engineers to figure out whether the Xs found in the design are acceptable or not unless the purpose of every register is known. As a result, knowing what to check becomes challenging because if a tool just reports all the Xs, there will be too many false alarms, rendering the tool ineffective. One solution is to write an assertion for every key register; however, this approach is tedious and can be error-prone.

B. Current RTL X-Verification Methods

Recently, Kaiss et al. [8] proposed a SAT-based method for computing reset sequences that can be utilized to prevent undesirable X propagations. Haufe and Rogin [6] proposed a technique that utilizes automatic RTL code transformation to avoid unexpected X-propagation. However, their method is based on templates and cannot handle all the RTL syntax. Another way to detect X-problems is to run gate-level simulation by assigning random values to Xs and then compare the results with RTL simulation. The major advantage of this approach is that simulation is easy to use and existing verification infrastructures can be used. However, setting up gate-level simulation still needs time, and gate-level simulation is slow. In addition, once a bug is found at the gate level, finding the root cause of the problem at the RTL can be difficult.

In order to accurately handle Xs at the RTL, Chou et al. [5] proposed a technique based on symbolic simulation which treats Xs as symbols and produces Boolean expressions in terms of symbols. Since each symbol represents an arbitrary value like real Xs in hardware, the Xs can be handled accurately. Take Fig. 1(a) as an example, since the X in "*a*" is treated as a symbol, both conditions can be considered and the correct Boolean value for signal "*out*" can be produced as shown below:

$$out = a \ ? \ b \ : \ c \ ;$$

To check whether the X in *a* can propagate to output *out*, Chou's method first duplicates the design and then performs symbolic simulation. In the duplicated version, variables with Xs are replaced with new symbols, but other symbols remain the same. Next, a miter is used to check whether the outputs can be different. The built Boolean expression is shown below. A SAT solver is then used to solve the problem: if a solution can be found, then the X can propagate to the output.

$$\begin{array}{l} out \ = a \ ? \ b \ : \ c \ ; \\ out' = a' \ ? \ b \ : \ c \ ; \end{array} \Rightarrow \text{solve } miter(out, out')$$

Due to the accuracy of this technique, it is adopted in our methodology for X-analysis. However, scalability remains an important issue. To cope with such a problem, in the next section we will describe several methods that allow us to apply this technique to large designs.

IV. Our X-Analysis Methodology

The goal of this work is to design a practical methodology that can find X-problems in industrial-size circuits. To achieve this goal, the methodology should fit in the simulation-based verification flow that most people use, and it must scale to large designs. In this section, we first formulate the X-analysis problem, and then discuss how our eXact methodology addresses several practical issues in X-analysis, including how to generate stimulus, what to check, and how to enhance scalability. Finally, we outline the flow of our scalable X-analysis methodology.

A. Problem Formulation

Given a design in which Xs may still exist after reset, we seek to find X-problems that corrupt important registers. In other words, we want to make sure that all key registers [1] which have known values in logic simulation after reset are indeed X-free. In our work, we assume that logic simulation has been carried out to the extent that most simple bugs, including some X-problems, are already found. Therefore, the rest of the bugs are those masked by logic simulation due to X-optimism. Note that X-problems are not necessarily design errors and can be caused by incorrect software reset sequences or misinterpreted specifications. In this work we only focus on X-problems in designs.

B. Generating Stimulus for X-Analysis

Constrained-random simulation is a commonly-used verification technique. Therefore, constrained-random testbenches are readily available for most designs for stimulus generation. If one has properly-constrained or under-constrained testbenches that can generate all possible reset sequences, and the symbolic simulator can simulate enough cycles that cover all the sequences, then the X-analysis problem can be solved perfectly. However, this is unlikely to happen in practice.

Since many designs, including the one used in our case study, only have a limited number of possible initialization sequences, we propose to use simulation waveforms as stimulus for the design block under X-analysis. Using waveforms for X-analysis has the following advantages:
(1) It allows us to partition a design into smaller blocks. Since it is easy to dump waveforms for any part of the design, we can partition the design and then perform X-analysis on smaller blocks. In this way, the formal analysis technique that we adopt will be able to handle the blocks due to their smaller sizes. Parallel processing also becomes easy due to partitioning.
(2) It fits into verification flows whose testbenches cannot be symbolically simulated. In our case, the testbenches are written in C++, SystemC and Verilog, and they cannot be symbolically simulated. By dumping waveforms for X-analysis, our verification environment does not need to be changed.

Fig. 2 shows our X-analysis flow. First, we partition the design and then perform logic simulation of the initialization sequence to dump a waveform for each design block. Next, we perform X-analysis on each block by replacing all the Xs in uninitialized registers with symbols. We then symbolically simulate each block using the waveform as its stimulus. When Xs are encountered in the waveform, we replace them by new symbols at every input timestep. Finally, we use Chou's formal X-analysis technique to find Xs at observation points. In the next section we will describe how we determine observation points. Note that if a design has multiple reset sequences, all of them need to be verified to make sure no bug is missed.

1 Partition the design into smaller blocks;
2 Perform logic simulation of the initialization sequence and dump a waveform for each design block;
3 For each design block, perform X-analysis using logic values from waveform as initialization sequences (Xs in the waveform are replaced with symbols during symbolic simulation);

Fig. 2. X-analysis flow using waveform as stimulus.

[1] Key registers are usually identified by users according to the specification.

Since logic simulation is inaccurate when handling Xs, our use of waveforms will create stimulus that may be both over-constrained and under-constrained. It may be over-constrained if there are X-problems in the block that fans out to the current one and the Xs are masked due to X-optimism, and it may be under-constrained because we treat each X as independent symbols. An under-constrained example is shown in Fig. 3(b), and an over-constrained example is shown in Fig. 3(c).

Our use of waveforms as inputs may produce false alarms but will not miss bugs. False alarms may exist because we treat each X as an independent symbol. As the example shows, doing so reduces the chance to eliminate the Xs. From our case studies, however, this is not a problem because we can eliminate most of the false alarms by having the designers quickly inspect the report. Bugs may be missed because over-constrained conditions in waveforms are caused by X-optimism when simulating the block that has X-problems. Therefore, as long as the problematic block is also checked, we will find unexpected Xs at its registers or outputs and discover the problem. Since in our methodology we check all the blocks that may have Xs, we will be able to find X-problems if they do exist and will not miss bugs. In the example, the X in B2's output may be missed because simulating B1 produces 0 instead of X at its output, making the input signal in B2 0 instead of X. Since simulating 0 produces 1 on B2's output in symbolic simulation, the X will be missed. However, the source of the problem is actually in partition 1, and the X-problem will be caught when verifying partition 1. As a result, no bug will be missed because our methodology verifies all the blocks. After the engineer fixes the problem in B1, the X-problem in B2 will also be fixed.

C. Deciding What to Check

Most industrial designs contain a large number of registers. For designs that allow Xs after initialization, designers need to specify which registers to check in order to avoid false alarms. This can be achieved by writing assertions using SystemVerilog's *$isunknown* which returns true if any bit of the expression is X or Z. If this information is available, our methodology will only check those assertions and can have the most accurate results. However, it may not be practical to specify everything that needs to be verified because the number of such registers may be large. Therefore, it is desirable to have a way for the tool to figure out what to check.

To address this problem, in our methodology we propose to check only the registers that do not have Xs in logic simulation. The major reason is that if registers already have Xs in the waveform, engineers are probably aware of them already. Therefore, there is no need to check those Xs again. By only checking registers that do not have Xs in the waveform, we can identify potential problems that are masked due to X-optimism in logic simulation. In this way, any X found by our methodology will be new to the engineer and is worth looking at. In our experience, we found that this approach can reduce the number of false alarms and point out problems that catch designers' attention right away.

D. Enhancing Scalability

To further improve the performance of our methodology, we perform temporal partitioning of our X-analysis by creating

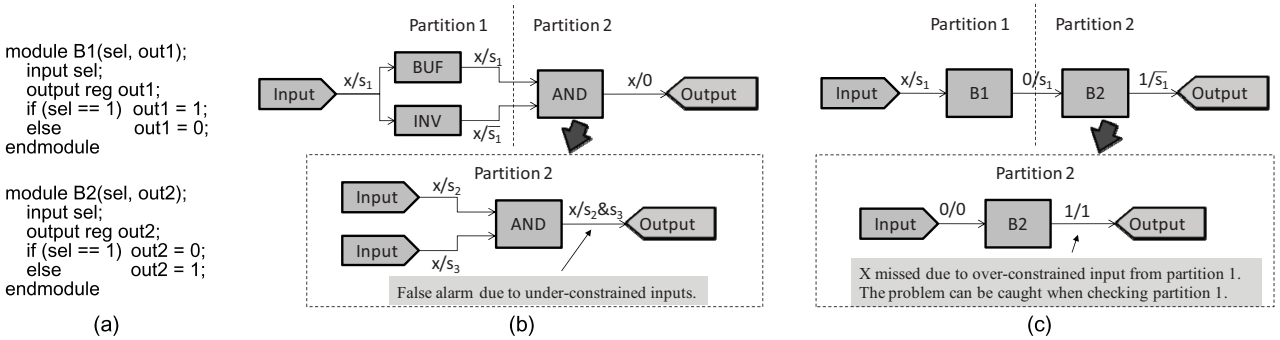

Fig. 3. An example to show over-/under-constrained characteristics using stimulus from waveforms and design partitioning. In symbolic simulation, Xs in waveform are replaced by symbols (denoted as s_n). Logic and symbolic simulation results are shown using "logic/symbolic". (a) RTL code for blocks B1 and B2. (b) False alarm at AND gate's output after partitioning due to under-constrained inputs caused by replacing Xs with new symbols. (c) Missed X at B2's output after partitioning due to over-constrained inputs caused by X-optimism in B1. This X-problem will be caught when verifying partition 1.

checkpoints in the initialization sequence and then execute formal analysis in intervals. The process of X-analysis after temporal partitioning is shown in Fig. 4. We first execute symbolic simulation to a checkpoint and then perform formal X-analysis. At the checkpoint, if any X is found, designers should check whether the X is acceptable or not. If the X is not acceptable, then a bug is found. If it is acceptable, then the non-X value in the design register should not cause any problem in the future, so we can simply execute logic simulation to the current checkpoint. Next, we perform abstraction [1], [7] by injecting new symbols for all the registers that have Xs at the current checkpoint, switch to symbolic simulation, and then simulate until the next checkpoint. X-analysis is then performed again at the next checkpoint. This process repeats until the whole initialization sequence is verified. It should be noted that checkpoints in the initialization sequence are all independent, and there is no need to carry any information between checkpoints.

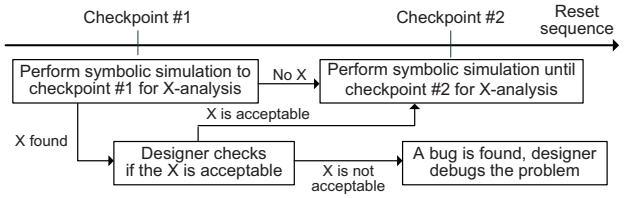

Fig. 4. Verification flow after temporal partitioning.

The main advantage of temporal partitioning is that formal X-analysis can be performed for a shorter period of time because new symbols are used to replace the Xs in registers at each checkpoint, thus reducing the complexity of symbolic simulation. However, since such Xs are now free symbols instead of complex Boolean expressions, temporal partitioning creates under-constrained conditions for the next interval. As a result, no bugs will be missed, but there may be false alarms.

E. Overall Methodology

Given a design and a reset sequence that includes both hardware and software reset, our eXact methodology, shown in Fig. 5, works as follows:

(1) Partition the design so that each block can be verified efficiently by the formal X-analysis engine.

(2) For a block, select checkpoints for temporal partitioning. Run logic simulation using the reset sequence to dump a waveform. X-free registers at those checkpoints are also identified.

(3) Use waveform as stimulus to perform X-analysis on X-free registers for every interval according to the method described in Section IV-D. Potential Xs will be reported.

(4) Repeat step 2 and 3 for all the blocks.

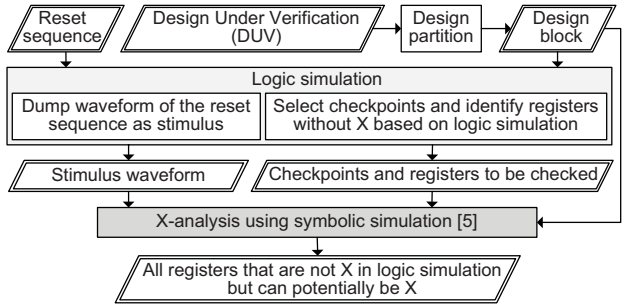

Fig. 5. Flow chart of the eXact methodology. By design and temporal partitioning, eXact can scale to large designs.

Our methodology utilizes design and temporal partitioning to reduce X-analysis complexity. In our experience, partitions with 200-300K gates are appropriate for X-analysis. While smaller sizes can reduce runtime, more blocks will need to be verified. If a computer farm exists for parallel processing, one can run several blocks in parallel to reduce throughput time.

In order to determine the number of cycles between checkpoints for temporal partitioning, one must find a balance between runtime and false alarms. Shorter intervals will make symbolic simulation faster, but there may be more false alarms to analyze. Although one can partition the sequence into equal-length intervals, it is recommended to consider signal activities for better performance. A heuristic is that shorter intervals can be used during hardware reset because there will be lots of signal activities and X-analysis can be slow. Longer intervals can then be used for software sequences to reduce false alarms since Xs in many registers have been removed already.

V. Case Studies

In this section, we first describe the characteristics of the design blocks used as our case studies and then show our X-analysis results. Finally, we describe what real bugs were found. Our experiments were performed using a commercial symbolic simulator called Insight [12] running on a Linux server farm. The machines in the server farm have Quad-Core Xeon processors with frequency ranging from 2.93 to 3.16 GHz, and they have memory between 16G and 128G.

TABLE I
CHARACTERISTICS OF DESIGN BLOCKS.

Block Name	Register Count		Gate Count	Lines of RTL	Description
	RTL	Gate-level			
alp_pcm	367	9247	30632	7802	Program counter memory which stores vectors used for digital sourcing.
alp_mpg	6739	109882	224986	42937	Memory pattern generator which can automatically generate test pattern for memory testing.
alp_cmem_eng	3614	13148	44291	35115	Capture memory engine which can store captured data.
alp_per_gen	331	5324	11011	5636	Period generator which can generate user period according to users requirement.
alp_lvm	15481	89848	234938	28523	Large vector memory which can be used as extra storage for vector data, capture data, etc.

A. Description of Design Blocks

In our case study, we applied the eXact methodology to a six-million gate high-speed tester design. The initialization sequence was composed of two phases of hardware reset and two stages of software reset. The hardware sequences were only a couple cycles long. The first stage of software reset (called mapping sequence) was approximately 20,000 cycles long, and the second stage (called pre-pattern sequence) was roughly 500 cycles long. The total length of the initialization sequence was approximately 40,000 cycles long because there were idle cycles in the sequence. The whole reset sequence was prepared by the hardware team and approximately 70% of design registers were initialized. Obviously, it will be difficult to write assertions to specify all the registers that need to be checked. We used C++ to generate the transaction level stimulus for the initialization sequence, and used a SystemC interface to pass the stimulus to the Verilog testbench. Since the design was too large for formal X-analysis, partitioning was necessary. To select appropriate block sizes for partitioning, we first picked a block and ran symbolic simulation for a few cycles. Next, we measured the run time of symbolic simulation. If the runtime seems to be reasonable (at least 20 cycles per hour in our case), then the block is suitable for formal X-analysis. In this work, we found that 200K-300K is the maximum number of gates that symbolic simulation can handle efficiently. We then applied our methodology on five blocks that the verification engineer was interested in. The characteristics of the blocks are shown in Table I.

B. X-Analysis Results

After we selected the blocks to verify, we then chose checkpoint intervals according to simulation speed. Our goal was to finish X-analysis of each block in at most two days with as few checkpoints as possible. Since the verification engineer was more interested in the correctness of the pre-pattern sequence, we performed X-analysis only on the sequence.

The X-analysis results of the blocks are shown in Table II. As our results show, most blocks could be verified without temporal partitioning. However, block alp_cmem_eng took more than one week to run without temporal partitioning and did need 129 checkpoints to be able to finish in two days. We analyzed the reason why this block had especially long runtime even though it was not the largest block. We found that there were several FIFOs inside the block that were implemented using memory. Since memory is more difficult for symbolic simulators to handle , runtime is also much longer. Apparently, it is impractical to handle such complex X-analysis problems using brute-force methods, showing that our methodology is useful for handling realistic designs.

From Table II, we can see that there are still X-problems in the design even though the design has been heavily verified. The percentage of registers with X-problems, however, is still small, suggesting that our under-constrained methods did not create large amounts of false alarms. It is interesting to note that a relatively large number of Xs were found in alp_mpg and alp_cmem_eng after software reset. More analysis shows that most Xs were from a couple modules that were instantiated several times. After ruling out repeated ones, only 71 Xs in alp_mpg and 69 Xs in alp_cmem_eng really needed to be checked. The designers quickly pinpointed 5 registers that were worth looking at in alp_mpg, and they turned out to be false alarms due to temporal partitioning. The rest of the Xs were mostly real Xs but would not affect design correctness.

In Column 4 of Table II we provide the percentage of registers that need to be checked according to the method described in Section IV-C. It is observed that by focusing only on the registers without X in logic simulation, we can reduce the number of registers that need to be analyzed. Take alp_mpg for example, we only need to analyze 90.3% of the 6739 RTL word-level registers. If gate-level simulation were to be used for X-analysis, then 109,882 bit-level registers needed to be checked, which would be even more inefficient. Note that we started X-analysis at the second software initialization sequence where a large number of Xs have already been eliminated. For designs that allow more Xs after initialization, this heuristic will provide even more benefits.

TABLE II
VERIFICATION RESULTS USING EXACT METHODOLOGY.

Block Name	# CKPTs.	Runtime	Checked/Total Registers(%)	# Found Xs
alp_pcm	1	7hr5m	79.0%	2
	58	6m	75.3%	43
alp_mpg	1	49hr12m	90.3%	793
alp_cmem_eng	129	53hr16m	85.3%	848
alp_per_gen	1	45m	98.8%	4
	2	29m	93.8%	16
	4	3m	94.1%	18
	8	3m	93.1%	18
alp_lvm	1	4hr49m	78.5%	1
	2	4hr30m	78.5%	3

To examine the effect of different number of checkpoints on X-analysis runtime and the number of false alarms, we varied the number of checkpoints for alp_per_gen, block alp_lvm and alp_pcm. As shown in Table II, when the number of checkpoints increased, the runtime for X-analysis decreased, but the number of Xs increased. This trend is consistent with what we predicted earlier — shorter intervals will make X-analysis faster, but it could create more false alarms. The runtime of block alp_lvm, however, did not improve significantly with more checkpoints. The reason is that it took almost four hours for logic simulation to reach the pre-pattern reset sequence and formal X-analysis only took less than one hour. As a result, although formal analysis time was reduced by approximately 40%, the overall improvement was less obvious.

To check how well our methodology can handle long traces, we also tried running alp_per_gen for the whole initialization sequence that was almost 40,000 cycles long. Without temporal partitioning, we could not finish symbolic simulation in a week. By partitioning the trace to 19,148 checkpoints, X-analysis of the whole trace finished in 43 hours and 51 minutes, and 32 Xs were reported. This result shows that the temporal partitioning technique scales well to long traces.

The memory usage of our case studies was under 2G for most design blocks. One exception is alp_lvm, which used 10G because it contained large memory models. Note that this memory usage is an over estimation because the symbolic simulator that we used does not free symbolic traces. However, in our methodology the traces can actually be freed after we finish a checkpoint. As a result, the reported memory usage can be much larger than necessary.

C. Bugs Found and Discussions

In order not to miss bugs, the designer has to check all the found Xs and decide which Xs should be analyzed. In our case study, although designers found that most Xs were OK because they did not affect circuit operation, they confirmed that most of the Xs were real and they were not aware of the Xs in the past. In our experience, approximately 20% of the Xs were false alarms caused by under-constraining conditions, 70% of the Xs did not need to be analyzed because the designers knew right away that the Xs were not important, and only 10% of the Xs needed to be further analyzed. Using eXact, we found 3 bugs that escaped initial verification. One of them is illustrated in Fig. 6. As we analyzed in Section III-A, the source of X is often localized; therefore, if we remove irrelevant RTL code, the structures of found bugs all look like simple if-then blocks as shown in the figure. In this case, register *r*1 is supposed to have a known value that controls the value of *enable_o*. However, since *r*1 is not initialized properly and *enable_a* != *enable_b*, *enable_o* can have an unknown value in the circuit, leading to a unexpected result.

```
// r1 = 1'bx; enable_a != enable_b;
if (!r1)    enable_o <= `DELAY enable_a;
else        enable_o <= `DELAY enable_b;
```

Fig. 6. A simplified example of bugs found by eXact. Register enable_o has nondeterministic values due to X in r1.

The X-problems found in the design were serious because they could break the data bus and left the chip in a nondeterministic state. This may eventually break all the functionality associated with the chip. Once the bug was found, however, fixing the problem was easy – we routed the reset signal to the problematic register.

Actually, one of the bugs was also found by a traditional method that compares the results of gate-level and RTL simulation. However, it took the verification team more than one man-month to set up gate-level simulation due to numerous transactors used in behavioral and RTL code. In addition, delta delay caused a lot of race conditions, which was very difficult to solve in gate-level netlists. Once simulation mismatches between RTL and gate-level designs were found, it took the team another man-month to find the corresponding RTL code that caused the X-problem. However, it only took us three hours to set up the environment for each block, and the X-problem is found by our methodology within 3 minutes. In addition to the huge reduction in bug-finding time, fixing the bug is also considerably easier because the eXact methodology works earlier in the design cycle on the RTL code directly. Therefore, the designer can easily identify the problematic code and fix it. This experience suggests that our eXact methodology provides a practical solution for finding reset nondeterminism problems in RTL designs.

VI. Conclusion

In this paper we proposed the eXact methodology to find nondeterminism problems, also called X-problems, caused by uninitialized registers in RTL designs. We utilized the formal X-analysis method by Chou [5] due to its accuracy, and we devised design and temporal partitioning techniques to improve its scalability. Furthermore, we discussed several important issues when applying X-analysis to real designs, including how the testbench should be designed and what to check. By utilizing waveforms as inputs and checking only variables without X-values in logic simulation, our methodology can be easily adopted into most simulation-based verification flows and find potential X-problems quickly. Our case study using an industrial design shows that our eXact methodology can find X-problems quickly and accurately, thus significantly reducing design verification time and improve design quality.

Acknowledgment

This research was supported by the National Science Council, Taiwan under Grant NSC 97-224-E-002-216-MY3; Excellent Research Projects of National Taiwan University,95R0062-AE00-05; and Small Business Innovation Project of Ministry of Economic Affairs, Taiwan, 1Z960401.

References

[1] V. Bertacco, "Scalable Hardware Verification with Symbolic Simulation," Springer, 2005.
[2] D. Brand, R. A. Bergamaschi and L. Stok, "Be Careful with Don't Cares," *DAC'95*, pp.83-86.
[3] R. A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, and S. Prakash, "Efficient Use of Large Don't Cares in High-level and Logic Synthesis," *ICCAD'95*, pp. 272-278.
[4] E. Cheung, X. Chen, F. Tsai, Y.-C. Hsu and H. Hsieh, "Bridging RTL and Gate: Correlating Different Levels of Abstraction for Design Debugging," *HLDVT'07*, pp. 73-80.
[5] H. Z. Chou, K. H. Chang, and S. Y. Kuo, "Handling Don't-Care Conditions in High-Level Synthesis and Application for Reducing Initialized Registers," *DAC'09*, pp. 412-415.
[6] C. Haufe and F. Rogin, "Ad-Hoc Translations to Close Verilog Semantics Gap," *workshop on DDECS'08*, pp. 1-6.
[7] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix, "Hybrid Verification Approach: Getting Deep into the Design," *DAC'02*, pp. 111-116.
[8] D. Kaiss, M. Skaba, Z. Hanna, and Z. Khasidashvili, "Industrial Strength SAT-based Alignability Algorithm for Hardware Equivalence Verification," *FMCAD'07*, pp. 20-26.
[9] M. D. Moffitt, J. A. Roy, and I. L. Markov, "The Coming of Age of (Academic) Global Routing," *ISPD'08*, 148-155.
[10] R. Ranjan, Y. Antonioli, A. Hunter, and O. Petlin, "Formal verification enables safe X handling," Dec. 2008. (available at http://www.scdsource.com/article.php?id=324)
[11] M. Turpin, "The Dangers of Living with an X," SNUG, 2003.
[12] Avery Design Systems Inc., http://www.avery-design.com