

Addressing the Challenges of Reset Verification in SoC Designs

Chris Kwok, Priya Viswanathan, Ping Yeung
Design Verification and Technology
Mentor Graphics Corporation
Fremont, U.S.A.

Abstract – Modern system-on-chip (SOC) designs contain a high level of complexity in the reset distribution and synchronization circuitry. Verifying that a design can be correctly reset under all modes of operation presents a significant challenge.

In this paper, we present the common occurring challenges that are involved in reset tree verification and solutions to address them. We lay out a three part approach which combines static analysis of the design structure, RTL simulation with X-propagation and formal verification to build a complete solution. The solution was tested on a customer design. We presented the results.

Keywords – CDC, Reset Verification, Formal Verification, Simulation

I. INTRODUCTION

Today's SoC designs integrate many design IP blocks from various providers, each with their own implementation of reset. The reset architecture of a digital design can also be very complex. Designs have multiple sources of reset, such as power-on reset, hardware resets, debug resets, software resets, and watchdog timer reset. Even the basic reset decisions [1] [2] require special attention. It's a challenge to ensure that all sources of reset propagate to the intended destinations under all conditions. Errors in reset design can lead to metastability, glitches or other functional failure of the system. Furthermore, complex interactions can occur with the convergence of multiple resets, multiple clocks and multiple power domains. In many cases, this leads to a reset tree that is larger and more complex than the clock tree. Many of the concerns related to clock tree synthesis and load balancing now apply to the reset tree as well.

Simulation has been the primary method traditionally used to verify reset behavior, often with a heavy reliance on gate level simulation. However, simulation test coverage is incomplete and gate level simulation is only run late in the design cycle. These factors result in reset related bugs that cause costly, late-stage design changes and, in the worst case, silicon respins and time-to-market delays. Typically these types of bugs are of a very serious nature, rendering the chip completely unusable. For example, a reset bug may result in it being impossible to reset the design to a known good starting state, making its operation completely unpredictable. In more extreme cases, the design may consume too much power during assertion of reset, causing the device to overheat and be permanently damaged.

The purpose of this paper is to bring the awareness the complicated reset problems. In section II, we will describe common reset problems. In section III we will discuss how the problems can be identified through a variety of different techniques including static analysis, simulation and formal verification. Real life examples will be given in section IV to illustrate the problems.

II. COMMON PROBLEMS

In this section, we highlight some of the common problems related to reset verification. These are some of the problems that we have seen in our customers working on real customer products. We will separate it into two main categories: reset tree correctness and reset usage. In section IIA we will discuss issues related to correctness of reset trees and issues commonly occurred in the generation of reset. In section IIB, we will discuss issues related to the usage of resets, and issues common to the reset usage.

A. Reset Distribution Tree

The first set of reset problems in design is whether reset was implemented correctly. Figure 2 shows a typical reset tree. If the reset tree is incorrect, then the chip will not function properly. There is a long list of potential problems that we have seen, and we highlight 2 common problems here.

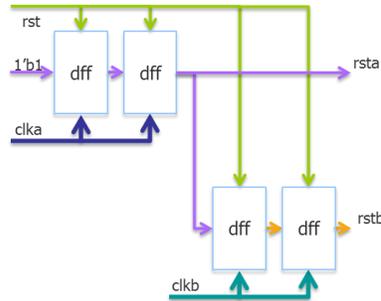


Figure 2

i. Resets used with mixed (asynchronous/synchronous) types

In general reset can only be defined as asynchronous or synchronous to a clock. Sometimes, it's being used as synchronous reset for clock1, and asynchronous for clock2. This usually indicates a misunderstanding of the reset in the system. The following RTL illustrates a clock being used as an asynchronous reset on register q1 and a synchronous reset on register q2.

```
always @(posedge clk or negedge rst_n)
    if (!rst_n)
        q1 <= 1'b0;
    else
        q1 <= d;

always @(posedge clk)
    if (!rst_n)
        q2 <= 1'b0;
    else
        q2 <= d;
```

ii. Registers with overlapping set and reset.

It is generally not a good idea to allow both asynchronous reset and set exist in a register and is not common in synchronous designs. In fact, many technology libraries do not provide a register that has both asynchronous reset and set. This problem is usually only a simulation problem and not synthesis issue. The RTL below may not simulate correctly because of the behavior of the simulation model. When both set and reset are both active asynchronously, the verilog always block may miss one of the reset or set states leading to race condition and there will be a mismatch between simulation and synthesis. An RTL example of overlapping set and reset is shown below.

```
always @(posedge clk or negedge rst_n or negedge set_n)
    if (!rst_n)
        q <= 1'b0;
    else if (!set_n)
        q <= 1'b1;
    else
        q <= d;
```

If `set_n` and `rst_n` are asserted together, there is a race condition. The resulting behavior of this register will depend on the physical implementation of this cell. One solution to avoid set and reset race conditions is to convert one of the asynchronous set or reset signals into a synchronous signal. The following RTL avoids a race condition by using an asynchronous reset and a synchronous set.

```

always @(posedge clk or negedge rst_n)
  if (!set_n) begin
    q <= 1'b1;
    if (!rst_n) q <= 1'b0;
  end
  else
    q <= d;

```

B. Reset Usage

The second set of reset problems is related to reset usage. To understand it we need to first define the terms reset domain and clock domains. A reset domain is characterized by the following attributes – a) Type – synchronous or asynchronous b) Polarity – active low or active high c) Value – set for 1 and reset for 0 and d) the top resetting signal. Multiple reset domains can be grouped together. A clock domain is associated with clock source signal and optional polarity. Multiple clock sources can be grouped together in a single clock domain.

In a design with multiple reset domains and clock domains, it is important to make sure resets are used properly. The reset tree annotates the sequential elements in the design with the reset domain information. The clock tree is generated and the clock domain information is annotated on every register. With every register in the design having a clock domain and reset domain – the usage of the resets in a particular clock domain can be determined. With this information, we can then identify asynchronous reset crossings across sequential elements in the same clock domain, as well as identify crossings between asynchronous and synchronous resets across registers in the same clock domain. These paths between resets are called Reset Domain Crossing (RDC) paths.

i. Missing Synchronizer for asynchronous reset

All asynchronous reset signals should be synchronized to the target clock domains before used. The problem of using raw(unsynchronized) reset is not the assertion of reset but with the de-assertion (release) of the reset. If the reset is de-asserted close to the next clock edge – it may violate the reset recovery time leading to metastability. When an asynchronous reset violates the setup and hold requirements for a register, this violation is likely to happen simultaneously on multiple registers and the register metastability will result in a random delayed release of reset on the metastable registers. Asynchronous reset synchronizers that ensure asynchronous assertion and synchronous de-assertion provide a reliable reset mechanism. Figure 3 shows the waveform for the asynchronous reset problem. When the active low reset is de-asserted very close to clock, Q becomes metastable.

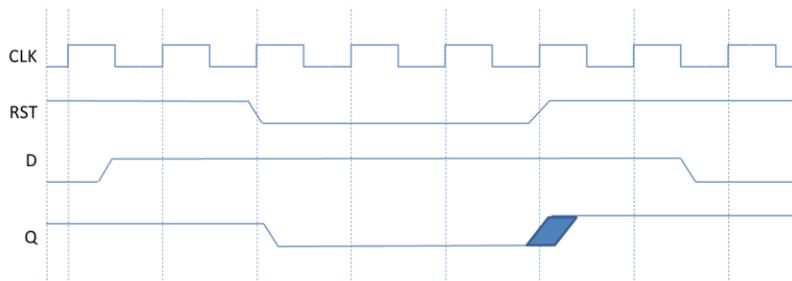


Figure 3

There are different ways to synchronize resets. Figure 4 shows typical reset synchronizers. The external reset signal, `RST1`, is active low. When it is asserted, the output of the reset synchronizer will be 0 immediately. When `RST1` is released, the output of the reset synchronizer will be high in the next or the cycle after. The release of the reset signal is synchronized to the clock, `CLK`. The synchronizer prevents metastability to be sampled when reset is de-asserted.

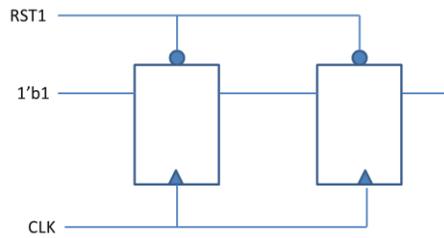


Figure 4

ii. Correct Polarity and Clock

Once resets are synchronized, we need to make sure that the reset signals are used properly. Two common problems are that resets are used with wrong polarity or wrong clock. Figure 5a shows that the downstream register using reset synchronizer is clocked in a different clock (CLK2) from reset synchronizer (CLK). CDC analysis [5] will detect a CDC crossing and the problem would be identified during CDC verification. Figure 5b shows that the register reset by the synchronizer is using active high reset, while the synchronizer is an active-low reset. Since this is not a CDC crossing, it will not be caught by CDC tool, and cannot be caught easily.

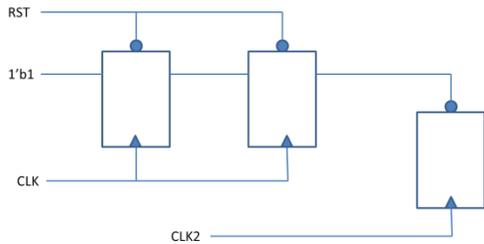


Figure 5a

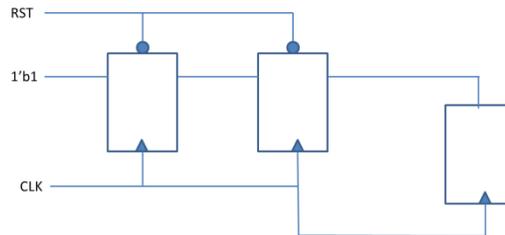


Figure 5b

iii. Reset Domain Crossing

Most modern chips employ multiple clock domains. Metastability in a design due to asynchronous clock domain crossing is a well-known issue [3] [4]. Advanced tools such as [5] are available to catch such structural and functional CDC issues. However, even if the data path is in the same clock domain, if the reset of the source register is different from that of the destination register – this asynchronous crossing path can still cause metastability at the destination register. This crossing involves understanding the use of reset domains and their interaction with clock domains [6] [7]. Figure 6a shows a circuit which could have metastability problems. If RST1 is asserted while RST2 is not asserted, DFF2 can sample asynchronous data. If the data transition is within the setup and hold window, then the DFF2 output will become metastable. The corresponding waveform is shown in Figure 6b.

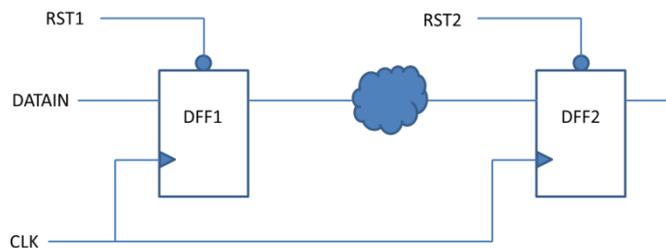


Figure 6a

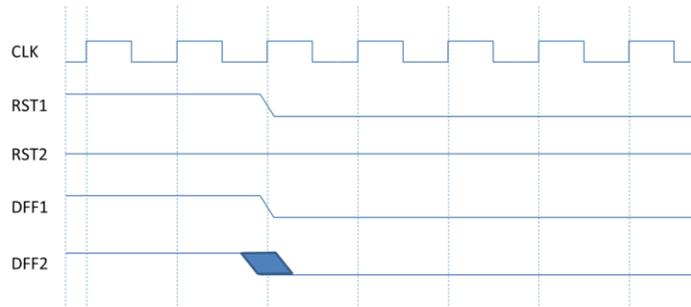


Figure 6b

iv. Incorrect Delay Sequences

In general, chips partition resets, through the use of delays in phases to avoid power surges which could burn the chip. This could lead to another problem. Figure 7 shows a simple example. The assertion of RST will prematurely cause DFF2 to reset while DFF2 is still in functional state. The inconsistent reset delay between adjacent registers causes incorrect data at the downstream logic. The reset structure assumes that RST is asserted for more than 3 cycles. If RST is only asserted 2 cycles, then when RST deasserts, DFF2 will be corrupted by DFF1 before DFF1 gets the reset signal.

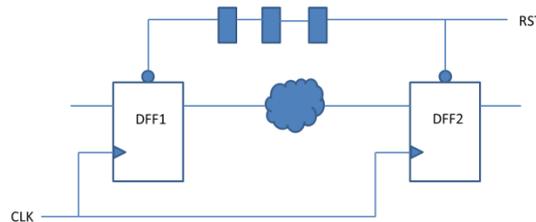


Figure 7

III. VERIFICATION SOLUTION

In this section, we propose a solution to solve the reset verification challenges. The solution involves multiple steps: static analysis, simulation and formal verification. Each step has its strengths and weaknesses. A complete solution needs to involve all 3 steps.

A. Static Analysis

The first step of the verification is Static Analysis. Static Analysis only requires RTL. The tool automatically analyzes the design, usually synthesizes the design and detects obvious design errors. The advantage is that it does not require any other information such as testbench stimulus or assertions.

Determining the reset tree can be done through static analysis. Static analysis is performed on the RTL. Resets can be identified by through synthesis, then resets are further classified as either synchronous or asynchronous resets, as well as active high or active low. Top reset signals can be recognized in the design by identifying the structural templates for synchronous and asynchronous reset pins of a register and their respective polarities. The top resets can be used to build the reset tree by propagating the reset signals through sequential logic till the destination registers. The top base resets can be classified into groups in the design depending on the source – the resets could be primary inputs, gated or soft resets or generated by a reset synchronizer. Every reset domain is characterized by its type – asynchronous/synchronous, polarity, its use as a set or reset. The combinational and soft resets need to be analyzed to resolve the participating control signals which may be constants or belong to a particular reset and clock domain. The designer can be allowed to group resets which are known to have the same source and are not asynchronous to each other. Once the reset tree is built – it can be annotated with multiple layers of design information including clock, power and delay levels.

The generation of the detailed reset tree enables structural checks on the reset architecture. Results of the static analysis vary from simple informative checks to complex functional checks. Some examples of informative checks are:

- a) Resets which are used as active low and active high in the design.
- b) Resets which are used asynchronously and synchronously which is not recommended in reset design.
- c) Conflict in usage of resets between specification and usage.
- d) Find resets used as data

The static analysis also performs functional checking for many of the issues in section II above. Some examples are:

- a) Asynchronous reset domain crossings
- b) Reset synchronizer structures and their reconvergence
- c) Reset delay protocols
- d) Glitch detection

In a complex SoC design with multiple reset sources and large number of registers – there could be millions of violations including all the categories of reset related problems as described above. A sophisticated debug methodology and environment is essential to help users manage the violations, as well as understand and debug the real issues.

The disadvantage of static analysis is that, in many cases, it can only show that it has potential problems. To show that the problem actually exists in the chip, we need to rely on RTL simulation or model checking formal engine. This requires users to write SVA assertions. In some cases, industry tool can also generate the assertions to be used by RTL Simulation and Formal Verification.

B. RTL Simulation with X-propagation

Today, RTL simulation is still the main verification vehicle. RTL simulation with silicon-accurate X-propagation semantics can be used to find issues related to uninitialized design elements, which cannot be found by static analysis. X-propagation addresses X-optimism by propagating X values forward in time. It involves if statements, case statements, and conditional assignments. For example, when a conditional expression has the value of X, the X-propagation enhanced simulator changes the language semantics to propagate X values. These values can be observed in simulation waveforms and the downstream logic is affected by the propagated X values. If the X's are not blocked or handled correctly in the design, the simulation could fail. In particular, the design's silicon implementation could be subject to random failures, which are completely missed by normal RTL simulator.

C. Formal Verification

Although simulation-based verification is capable of detecting many design bugs, it requires a proper testbench stimulus. Model checking technology can be used to supplement traditional simulation methods. For example, formal verification can be used to verify these reset tree properties: (a) connectivity of all sources of resets to their intended destinations (b) X-pessimism and X-optimism (c) corruption of correctly reset storage elements with unknown logic values. Most of the reset problems resulted in X-states and eventually resulted in silicon bugs. Turpin [8] showed how X-bugs can be missed by simulation. Liu [9] discussed how Formal techniques can be applied to verify reset & X-state problems.

IV. RESULTS

We ran Questa[®] Reset Check on 3 designs of varying complexities and functionalities: a prototype block, a functional controller and a networking design unit. Each of the designs exhibits different interactions between the generated clock domains and the inferred reset domains leading to domain crossing issues. In general, users can customize the analysis by assigning explicit values to the control signals, or by grouping the clock and reset signals explicitly with directives. Table 1 shows the properties of the designs run as well as issues found. We found a lot of RDC in design 2. This was due to high number of reset domains as well as tight interactions of the data paths.

Design complexity	Design 1	Design 2	Design 3
Number of registers	305	47016	43622
Number of latches	0	592	0
Number of RAMs	2	0	64
Number of Gate-level modules	0	88	20

Reset/Clock domains information	Design 1	Design 2	Design 3
Total number of reset domains	3	36	48
Total number of clock domains	9	5	12
Number of clocks crossing reset domains	2	3	6
Number of resets crossing clock domains	2	5	5

Number of (RDC) reset domain crossings	Design 1	Design 2	Design 3
Data path across same clock domain	3	25766	225
Data path across different clock domains	0	2618	42

Results of static analysis	Design 1	Design 2	Design 3
Missing asynchronous reset synchronizer		✓	✓
Unexpected gate in reset tree	✓	✓	
Reset signal used as asynchronous and synchronous			✓
Good asynchronous reset synchronizer		✓	

Table 1 Summary of Static reset verification

Other static reset issues found in the designs included:

- Unused user specified resets
- Conflicts between specification and reset usage
- Combinational logic before asynchronous reset synchronizer

The reset domain crossing across data path registers in same clock domain is a RDC violation. Typical report of such a violation indicates the start and end registers with the clock and reset domain information.

Figure 9a shows a simple RDC violation. The start register 'mask' reset asynchronously by a signal 'rst' driving the end register 'rx_masked_data' reset asynchronously by a different signal 'clr'. Both the registers are clocked by the output of a scan mode multiplexer thus belonging to the same clock domain.

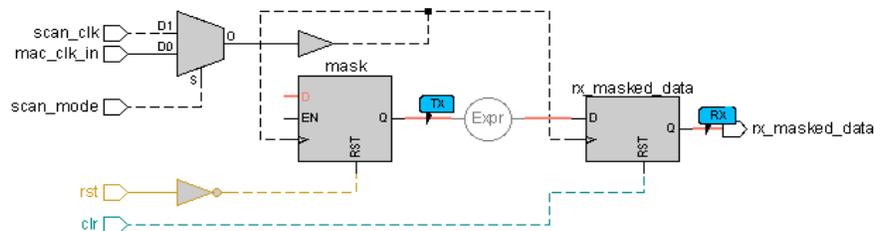


Figure 9a

Figure 9b shows another example of a RDC violation. The green and pink colors indicate the different reset domains for the start and end registers marked in blue.

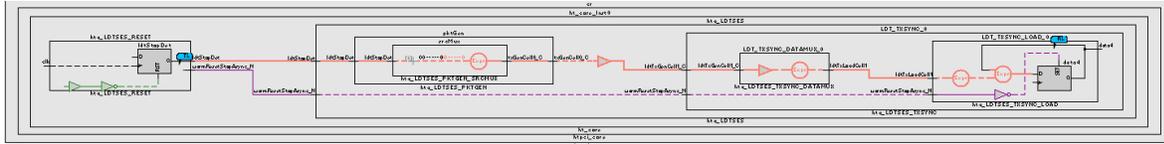


Figure 9b

Static reset issues are also reported to users. Figure 9c illustrates a reset used synchronously and asynchronously in a design. The highlighted path shows the top primary reset input `rst_HT_L` resetting the first register asynchronously and the second register on the right hand side through a synchronous reset mux select.

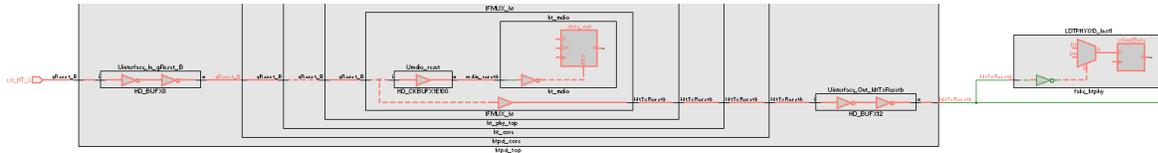


Figure 9c

One of the important debug features is to understand the interaction between clock domains and reset domains. It can be represented in the form of a table. Figures 9d and 9e illustrate the clock and reset mapping.

Clock Groups								
Name	rst_HT_L	..vPreMux_N	..DFFR_51.q	..p_rxenable	..reAsync_N	..etAsync_N	..SyncSized	rst_PCIBR_L
clk_PCI1CLK_H								R: 3, L: 0
clk_ht100_or_200_H		R: 23, L: 0					R: 17, L: 0	R: 23, L: 0
clk_pcbir66_H								R: 9, L: 0
h_extxclkln	R: 179, L: 0							
h_extxclkln	R: 167, L: 0							
h_mdc	R: 262, L: 0							

Figure 9d

Reset Groups								
Name	ldtRxWinkClk	txclrx2p	ldtTxWinkClk	h_extxclkln	h_extxclkln	h_mdc	..or_200	..pcib
...t_inloop_inlp_rxenable			R: 3, L: 0					
...loop_rst_inlp_rxenable	R: 1, L: 0							
...mux.htext_ldtRxSync				R: 1, L: 0				
...ht.ht_mux.rxsync_det				R: 6, L: 0				
...SYNC_ACTL_0.reset_N							R: 23, L: 0	
...RxWinkRstPreAsync_N	R: 644, L: 0							
...0_ldtRxWinkSyncSized	R: 7, L: 0						R: 17, L: 0	
...SIZER_0.resetAsync_N	R: 151, L: 0							
...TXSYNC_RESET_0.rst			R: 87, L: 0					
...ch.UDP_U_DFFR_51.q	R: 3, L: 0							
...RESET_ldtRxSyncArm								

Figure 9e

Once the static results were all validated, we identified several potential problems that needed further investigation. One of them is related to FIFO. We wanted to verify that FIFO was functioning correctly. We ran

Questa® Formal [10] on the design. The entries of the FIFO were initialized to 0 after reset. By manipulating the two request signals, reqa & reqb, Questa® Formal found a way to enqueue the FIFO with corrupted data. The X propagation path is shown in red in Figure 10a. And the waveform is shown in Figure 10b.

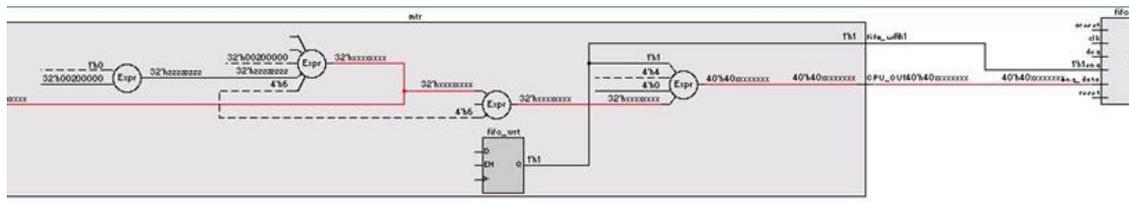


Figure 10a

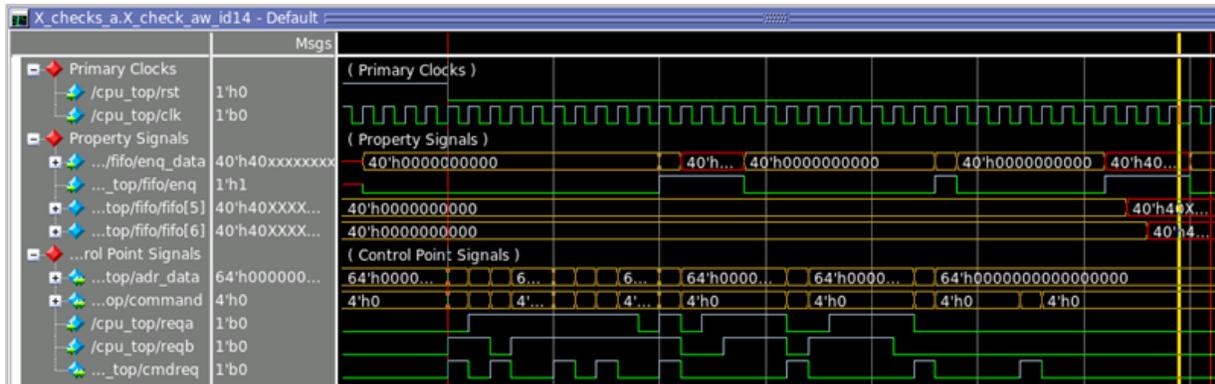


Figure 10b

V. CONCLUSION

In this paper, we presented several common reset problems, the reset verification challenges, and a proposed comprehensive solution to address the problems. We have also demonstrated the solution through verifying several customer designs, and presented the issues discovered. We showed that many of these issues are hard to catch bugs that cannot be caught easily with simulation or traditional verification techniques. Complete verification requires a full suite of methods involving static analysis, simulation and formal analysis.

REFERENCES

- [1] C. E. Cummings, et al., “Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?”, Synopsys User Group (SNUG) 2002
- [2] C. E. Cummings, et al., “Asynchronous & Synchronous Reset Design Techniques – Part Deux”, Synopsys User Group (SNUG) 2003
- [3] R. Ginosar, “Metastability and Synchronizer, A Tutorial”, IEEE Design and Test 2011
- [4] C. Kwok, et al., “Using Assertion-based Verification to Verify Clock Domain Crossing Signals”, DVCon 2003
- [5] Questa® CDC User Guide, version 10.3d, 2014
- [6] A. P. Chowdhury, “Dealing with SoC metastability problems due to Reset Domain Crossing”, www.embedded.com, Nov 2013
- [7] A. P. Chowdhury, “SoC tool flow techniques for detecting reset domain crossing problems”, www.emdedded.com, Aug 2014
- [8] M. Turpin, “The Dangers of Living with an X (bugs hidden in your Verilog),” version 1.1, October 2013
- [9] K. Liu, et al., “Using Formal Techniques to Verify System on Chip Reset Schemes”, DVCon 2013
- [10] Questa® Formal User Guide, version 10.3d, 2014