Reset and Initialization, the Good, the Bad and the Ugly

Ping Yeung
Design & Verification Technology
Mentor Graphics, Fremont, U.S.A.

Kaowen Liu
Design Technology Division
MediaTek Inc, San Jose, U.S.A.

Abstract- One daunting challenge of developing a low-power SoC design is how to verify its power-up, reset and initialization sequences. All of the different parts of the design must reset correctly, before the start of functional operation. In particular, design registers must initialize properly before they are used. Our initialization verification methodology classifies design registers into three types: GOOD registers (those that are initialized properly), BAD registers (those that are not initialized) and UGLY registers (those that are initialized, but are subsequently corrupted). This paper presents our methodology for verifying these three types of design registers.

I. INTRODUCTION

The initialization sequences of today's SoC design can be complex. To understand the effectiveness and to verify the correctness of the reset sequences, we need to examine the internal status of the design. Storage elements such as registers, latches, counters, FIFOs and memories provide a good snapshot of the design. Among them, registers are the most common and representative. Some registers in the design use asynchronous reset; some registers use synchronous reset. Some registers are equipped with both asynchronous and synchronous resets, but at the same time, some registers have no reset at all. There are a lot of reset signals in today's designs. Besides reset signals associated with each power domain, designs also have multiple sources of reset, such as power-on reset, hardware resets, debug resets, and software resets. It is a challenge to ensure that the sources of reset signal will propagate to their intended registers under different conditions.

Simulation has been the primary method used to verify the reset sequence, the reset propagation and the internal status of the design. However, with a large number of combinations of reset sequences, functional simulation alone cannot verified the reset sequence effectively, and simulation coverage is often insufficient. These factors result in power-up or initialization related bugs that may lead to costly, late-stage design changes and, in the worst case, silicon re-spins and time-to-market delays. Typically these types of bugs are very serious in nature, rendering the chip completely unusable. For example, an initialization bug may prevent the design from getting into the known good starting state, making its operation completely unpredictable.

In this paper, we define the different register status, the different phases of the initialization sequence, and the verification methodology. We explain how the problems can be identified through a variety of techniques including static analysis, simulation and formal verification.

II. The Register Status

In order to find out the status of the design deterministically, it is important to understand the status of the registers in the design. To keep the terminology simple, we called them the good, the bad and the ugly registers in the design.

1. The good registers are ones that have been initialized deterministically. Hence, they will drive meaningful values downstream to logics in their fan-out cone. In addition, they can be used to initialize other parts of the design.
2. The bad registers are ones that have not been initialized. Their initial values are unknown. In silicon, they are random 0s or 1s. These bad values will propagate downstream causing problems and unexpected results. To prevent these undesired effects, the outputs of the bad registers should be trapped or guarded to allow only valid data to be sampled.
3. The ugly registers are ones that have been initialized, but later corrupted by uninitialized logic in their fan-in cone. This situation should not have happened in normal functional mode. However, with complex gating conditions, it is important to verify that the initialized registers will not be overwritten unintentionally.

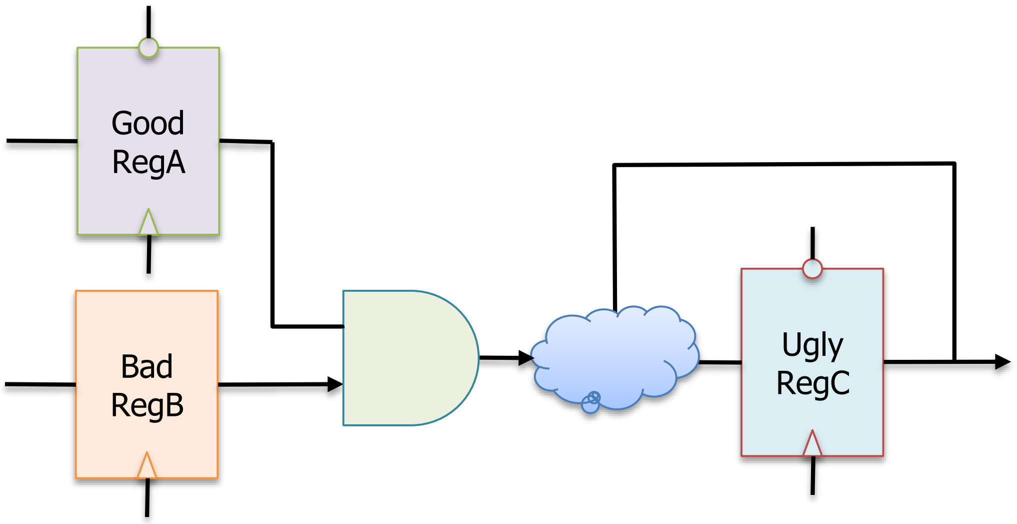

Figure 1. The Good, the Bad and the Ugly Registers

To illustrate, in Figure 1, the schematic shows a common structure in many designs. RegC is a finite state machine (FSM) that examines the data from RegB when needed. The RegA and RegC have explicit asynchronous reset signals. They are cleared when their reset signals are asserted. On the other hand, RegB does not have any reset signal; its initial value is undefined. During functional operation, RegA controls the sampling of RegB. It will only assert to allow meaningful data to pass from RegB to the FSM of RegC. However, things can go wrong in many ways:

1. The reset signals fail to reach RegA and/or RegC correctly. As a result, the initial status of the registers {RegA, RegB, RegC} will be {bad, bad, bad}. This is a reset propagation problem.

2. RegA fails to initialize before or at the same time as RegC. For instance, RegA may be controlled by a software reset while RegC is controlled by the power-on reset. In addition, RegA may power down when RegC is on. If RegA is not cleared before RegC, the undefined value of RegA will corrupt RegC. As a result, {RegA, RegB, RegC} will change from {bad, bad, good} to {bad, bad, ugly}. This is a reset domain crossing or reset timing problem[3].

3. Assuming RegA and RegC are reset correctly, RegA will control the sampling of RegB. If RegB has not be initialized properly when it is sampled by RegC, RegC will be corrupted. The biggest problem is: RegC is a FSM. Once it is corrupted, the feedback loop of the FSM will cause RegC to be corrupted continuously until it is reset explicitly. The status of the registers {RegA, RegB, RegC} are changing from {good, bad, good} to {good, bad, ugly}. This is an X propagation problem[1].
4. This circuitry is also common in low power designs where the output of RegA is the isolation signal. When the module containing RegB is powered down, the output of RegB will be undefined. As a result, the timing of the isolation signal from RegA is important. If it is late and ModuleB has already been powered down, the undefined output will likely corrupt RegC[2].
5. We have been assuming that the clocks to RegA, RegB and RegC are running continuously. However, to save power, it is likely that the clock to RegB is gated. Instead of loading continuously, RegB will be loaded only when new data is available. If the gating condition is off, RegB will not be initialized to any known value. Again, this will lead to the corruption of RegC.

We can summarize the potential problems in the table below. Basically, the problems are primarily caused by the connectivity and timing of the reset signals, the clock signals and the power control mechanism.

Problem	**Status of {RegA, RegB, RegC}**	**Status of {RegA, RegB, RegC}**
Reset Propagation Problem: reset signals failed to reach RegA and RegC correctly	{bad, bad, bad}	{bad, bad, bad}
Reset Timing Problem: reset signals to RegA and RegC did not meet protocol.	{bad, bad, good} RegA is not initialized correctly.	{bad, bad, ugly} RegC is corrupted by RegA.
X-propagaton Problem: reset signals to RegA and RegC assert correctly	{good, bad, good} RegB is not initialized correctly.	{good, bad, ugly} RegC is corrupted by RegB.
Power Sequence Problem: RegB is powered down	{good, bad, good} RegA fails to isolate it	{good, bad, ugly} RegC is corrupted by RegB
Clock Gating Problem: RegB is controlled by gated clock that is turned off	{good, bad, good} RegA fails to gate it	{good, bad, ugly} RegC is corrupted by RegB

Table 1. Summary of Problems and Changes of Register Status

III. The Initialization Sequence

From the example above, we can see that the status of RegC has moved from good to ugly. However, it did not happen when RegC is being reset; it happened after RegC has been reset. Hence, to understand how a register changes status, we need to examine the initialization sequence. In today's design, groups of modules are initialized when they are needed; blocks in different power domains are switching on and off; clocks are running at different ratio synchronous and asynchronous frequencies; registers are cleared with various hardware and software reset signals. As a result, the initialization sequence can be very complex and unpredictable in some cases. Conceptually, we can divide the initialization sequence into 3 phases: the power-up phase, the propagation phase and the reset-off phase. Traditionally, we assume the design is in a well-defined state after the propagation phase. Unfortunately, insufficient focus is put on the reset-off phase where some good registers may become ugly.

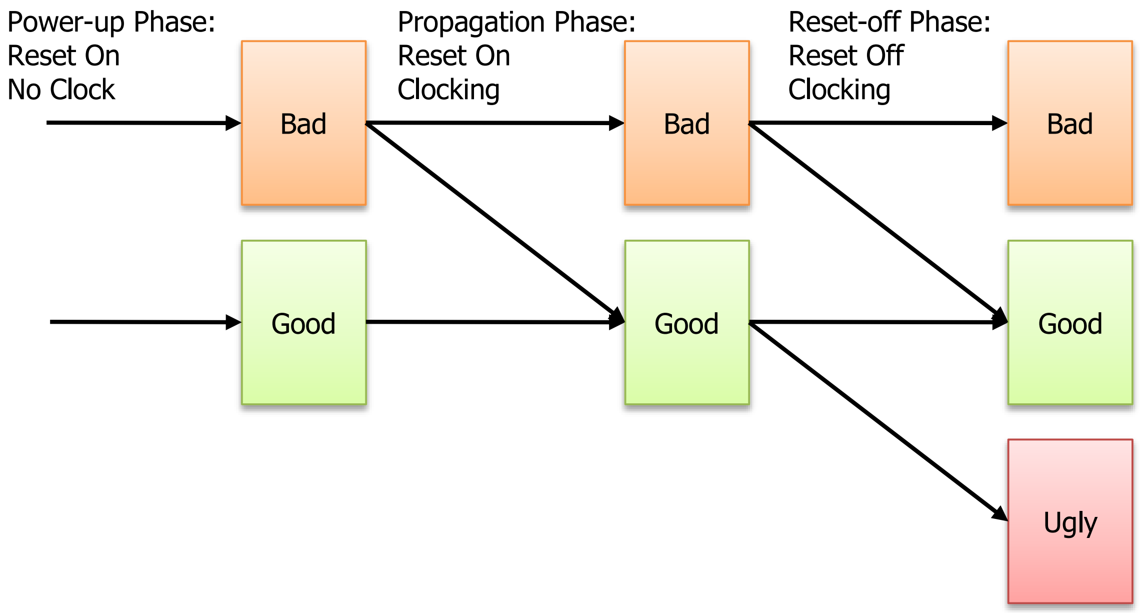

Figure 2. Initialization Phases and Change of Register Status

A. *The power-up phase*

On power-up, when the reset signals are asserted and the clocks are not running, registers with asynchronous or synchronous resets will be cleared. Hence, they are good. On the other hand, registers without any reset signal will be undefined. They are bad.

B. *The propagation phase*

During the propagation phase, the clocks start to toggle, but the reset signals are still asserted. Hence, the cleared registers will stay in their reset states. As the cleared values from the good registers propagate forward by the clocks to the un-reset bad registers, some of them will be cleared as well. In this phase, some bad registers turn good. This is the reason why we do not need to have reset signals on all registers. By understanding how good values will be propagated, the design can be initialized by a subset of resettable registers.

C. *The reset-off phase*

In this phase, the reset signals have been de-asserted. The design is being programmed to prepare for functional operation. Although most of registers have been initialized and are good, there are still a significant amount of registers that are bad. As the good registers start to operate and load new values, they will be affected by their fan-in cone. If some of the registers in their fan-in cone are bad, and the samplings do not guard against these bad inputs, the good registers will be corrupted.

IV. METHODOLOGY

The objective is to classify the registers in a design and to verify their correctness. A methodology is defined to ensure good registers are initialized correctly, to monitor the usage of the bad registers and to catch any ugly register that has been corrupted. The flow consists of:

A. The good registers

During the initialization phases, the good registers have been initialized explicitly by synchronous or asynchronous reset signals, or implicitly by loading known values from their data inputs. However, if any of the clock or reset signal is X, the output of the register will be X. To identify and to verify the integrity of the good registers, the following steps are performed:

1. Static verification is used to derive the clock and reset trees of the design. As clock gating and reset enabling are common in today's low power designs, extra care is taken to extract all the control signals to the clock and reset trees. Assertions are generated to ensure these signals are well behaved in various simulation environments.
2. Registers with explicit reset pins can be identified easily, but finding all registers that are initialized implicitly by their data inputs are not. X-pessimism introduced by simulation will mask some good registers. Hence, formal verification is used instead to identify all the good registers in the design.

B. The bad registers

For registers that have not been initialized, they are bad. They are potential X sources. If their unknown values are allowed to propagate, they will corrupt the rest of the design. Hence, the fan-out destinations of these registers are examined:

1. If it only fans out to other bad registers, it is ignored.
2. If it fans out to any output port, assertion is generated.
3. If it fans out to any critical control logic, such as FSMs, assertion is generated.
4. If it fans out to any good register, it is handled in the next step.

For these bad registers, they should be loaded with fresh data before use. To monitor them, X propagation is enabled during functional simulation. It will warn us if any of these bad values have been propagated and caused corruption in various design elements such as clock signals, select-expressions, registers, and FSMs.

C. The ugly registers

For the good registers that have been initialized, they can potentially be turned ugly. A good register can be corrupted by its fan-in logics when unknown values are allowed to propagate and load into it. It is potentially very harmful as downstream logic may assume the data from these registers are well defined. As a result, for all good registers in the design, assertions are generated to monitor them. The properties are written to ensure that once the registers are initialized, they will stay good and will not be corrupted. These assertions are included in the simulation regressions. But more efficiently, with formal verification, we are able to stress the design early and uncover corner-case scenarios that will cause corruption later on.

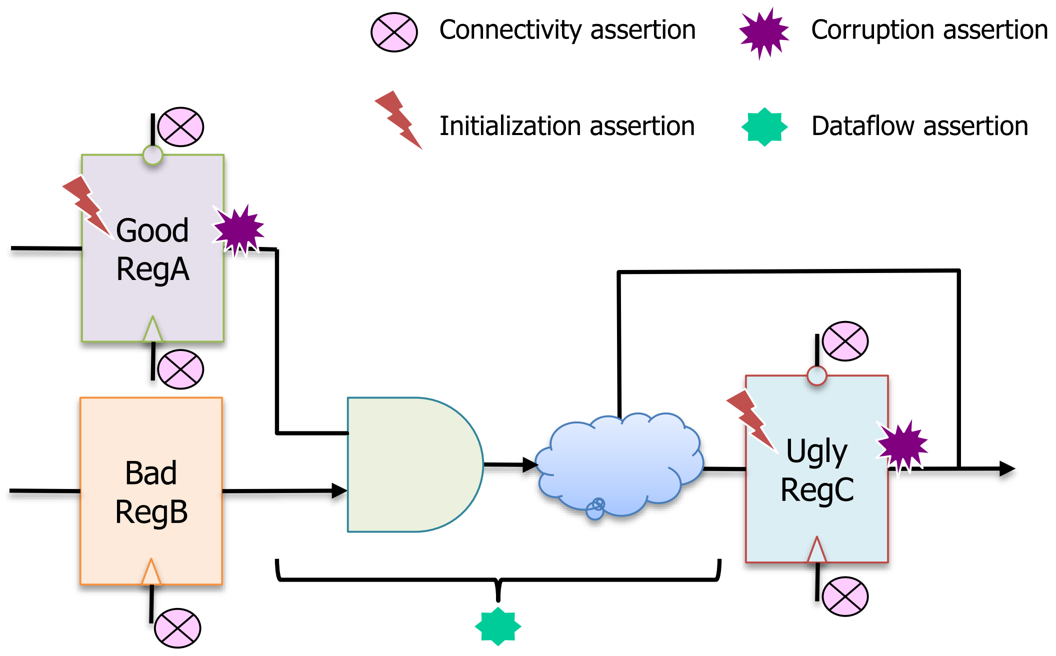

Figure 3. Summary of Assertions for Different Registers

Register	**Condition**	**Assertion**
All Registers	Clock is connected correctly from top to blocks/registers	Connectivity assertion e.g. (dut.a.enable) |-> (dut.clk === dut.a.b.c.gclk)
Good registers	Reset is connected correctly from top to blocks/registers	Connectivity assertion e.g. $rose(dut.rst) |-> ##[1:3] (dut.a.b.rst)
Good registers	Register is being initialized correctly	Initialization assertion e.g. (dut.a.b.rst) |-> (dut.a.b.rega === `IVALUE)
Bad registers	Uninitialized value is being guarded from propagating	Dataflow assertion e.g. $isunknown(dut.a.b.regb) |-> ! $isunknown({dut.a.b.fanouts}) $isunknown(dut.a.b.regb) |-> ! (dut.a.b.regb_out)
Ugly registers	Good registers should not turn ugly	Corruption assertion e.g. (! dut.a.b.rst) |-> ! $isunknown(dut.a.b.regc)

Table 2. Summary of Conditions and Assertions

V. Results

The methodology was implemented using products from the Questa Formal Verification tool suite, in particular, Questa ResetCheck® and XCheck®. Although formal verification may not have sufficient capability to handle a whole SoC design, with application specific tools such as ResetCheck and XCheck, we were able to apply them successfully on large IP blocks. The methodology was applied on 3 designs of varying complexity including a bridge block, a functional controller and a networking design unit.

Design complexity	**Design 1**	**Design 2**	**Design 3**
Number of register bits	305	47016	43622
Number of latch bits	0	592	0
Number of RAMs	2	0	64
Number of asynchronous resets	3	13	16
Number of synchronous resets	2	33	35
Number of clocks	3	5	12

Register Status information	**Design 1**	**Design 2**	**Design 3**
Good registers	38%	50%	34%
Bad registers	58%	9%	66%
Ugly registers	4%	41%	<1%

Table 3. Summary of Register Status after initialization and formal verification

From Table 3, we can see the percentage of the good, the bad and the ugly registers in the design. Design 1 represents a common situation where 1/3 of the registers are initialized during the reset sequence. However, majority of the registers are not initialized. And formal verification finds a small percentage of registers that can be corrupted and turned ugly. In one situation, the CRC calculation was corrupted. Formal verification had found a way to generate a read request to an uninitialized FIFO. As the egress value was unknown, the subsequent CRC value was undefined as well. This would be a problem in silicon. If the undefined CRC result becomes true occasionally, it will cause the bridge to sample bad data.

In Design 2, we see a general problem. Although there are a lot of good registers, a lot of the registers have potential to be corrupted (turned ugly) when stressed by formal verification. After some analysis, there are two primary reasons:

- There are a lot of register files in the design. When the incoming data is unknown, the registers will be corrupted one after another.
- To lower power consumption, most of these register files are driven by gated clocks. When the gating condition is undefined, it corrupts the clocks and subsequently the registers.

For Design 3, we were debugging the potential ugly registers from formal verification, we had found a few hurtful situations. Fortunately, they were found before RTL code freeze; weeks before the functional simulation environment is ready.

- 1 clock signal can be corrupted under some gating conditions,
- 4 warm reset signals can be unknown if one of the configuration register is not setup correctly,
- 45 good registers can potentially be corrupted under some specific scenarios.

Two situations are shown below. In figure 4, a good register with asynchronous reset was turned ugly.

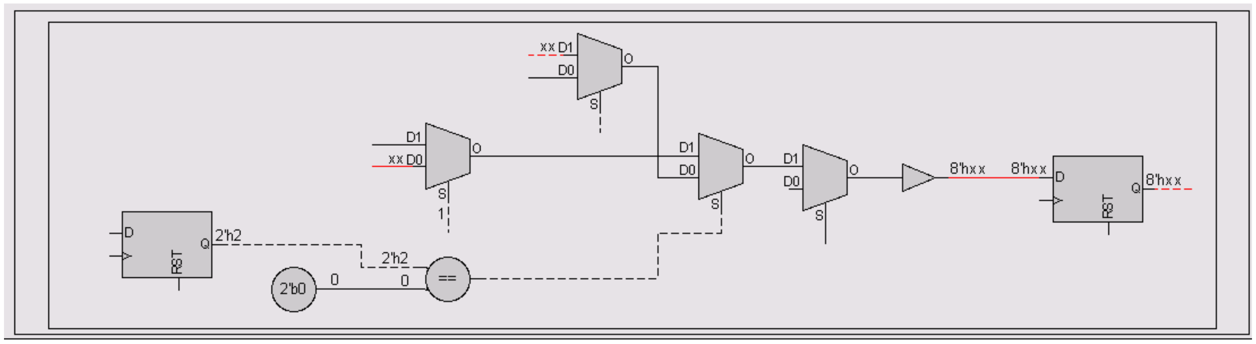

Figure 4. A good register was corrupted by an X source

The X source was originated from the X assignment inside the default case branch. Formal verification was able to find a corn-case situation that activates the default branch. As a result, even though the RX register was initialized correctly, it was corrupted quickly when the case condition was not matched by any branch. In figure 5, a good register with synchronous reset was turned ugly by a group of bad registers. The TX registers were not enabled. As a result, they were holding their corresponding un-initialized values. When the reset to the RX register was de-asserted, the RX register was corrupted quickly by the un-initialized TX registers.

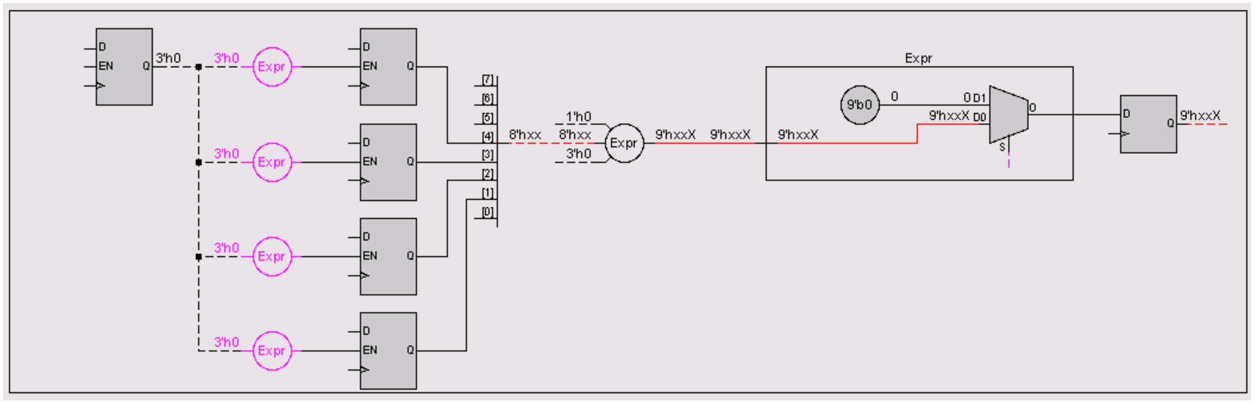

Figure 5. A good register was corrupted by a group of uninitialized bad registers

VI. SUMMARY

In this paper, we have classified design registers into three types: GOOD registers (those that are initialized properly), BAD registers (those that are not initialized) and UGLY registers (those that are initialized, but are subsequently corrupted). By doing so, sets of assertions can be generated to verify each types of registers and catch UGLY registers early with formal verification.

VII. REFERENCES

[1] Kaowen Liu, Penny Yang, Jeremy Levitt, Matt Berman, Mark Eslinger, "Using Formal Techniques to Verify System on Chip Reset Schemes", DVCon 2013.

[2] Ping Yeung, Erich Marschner, Kaowen Liu, "Multi-Domain Verification: When Clock, Power and Reset Domains Collide", DVCon 2015.

[3] Chris Kwok, Priya Viswanathan, Ping Yeung, "Addressing the Challenges of Reset Verification in SoC Designs", DVCon 2015.