Automatic Generation of Formal Properties for Logic Related to Clock Gating

Shuqing Zhao, Shan Yan
Broadcom
shuqing.zhao@broadcom.com, syan@broadcom.com

Abstract- **Clock gating is a common design technique used for reducing dynamic power of CMOS IC. To verify the correctness of the clock gating, the simulation methods to achieve thorough coverage and automatic checking has been a huge challenge. The main idea of our method is to generate low-level formal properties per flip-flop that uses any gated clock. This reduces the cone of influence for each property the formal tool has to process, thus increasing the probability a proof or counter-example can be produced within a reasonable amount of run time. Our scripts automatically extract properties by doing RTL design structural analysis of logic connecting to the clock gate cell(s) using a formal-proof tool's design-exploration API. For software-controlled clock gating, properties are generated to check that clocks for all gated flip-flops are parked at low when disabled. For hardware-controlled automatic clock gating logic, additional properties are generated. First, the clock gate cell is instrumented with functional coverage properties. Formal proofs of these properties ensure no gross design errors exist. The same coverage properties can also be used in simulation. Second, for each clock gate, our script performs an RTL analysis of the clock tree to report the number of flip-flops that the gate drives. Third, our script performs a comprehensive structural analysis of the sources and destinations of the clock-gated flip-flops. For each clock gate, the script extracts both the fan-in and fan-out flip-flops in the clock tree, and it automatically generates properties that can be used to check for error cases.**

I. INTRODUCTION

Reducing power consumption has become one of the key goals for IC design. In many applications such as mobile computing and data center computing it is as important as other IC quality metrics such as performance and area metrics. The power dissipation of a CMOS IC has two components - leakage power caused by unwanted subthreshold current in the transistor channel when the transistor is turned off, and dynamic power consumed by transistors switching states. Clock gating is a common technique used in many synchronous circuits for reducing dynamic power dissipation. It saves power by preventing the clock pin on registers (flip-flops and latches) and the clock-tree logic from toggling. Switching activities in downstream combinational logic and registers are also eliminated, further reducing the power dissipation. Our experience shows that just the clock-tree logic often consumes more than 40% of the total dynamic power.

The clock-gating logic is in the form of an ICG (Integrated Clock Gating) cell, as shown in Fig. 1, placed in the clock tree to prevent clock toggling from reaching specific registers. As listed in [1], there are several ways that clock-gating logic can be inserted into a design, either automatically by a tool or manually by RTL designers:

- The clock-gating logic can be added into the RTL code by inserting enable conditions that are automatically translated into clock-gating logic by the synthesis tools. This is a very fine-grain clock gating method that usually applies to a small piece of logic that maps to a particular process or always blocks in HDL.
- The clock-gating logic can be inserted into the RTL by automated clock-gating tools. These tools either insert ICG cells directly into the RTL, or add enable conditions into the RTL code as described above. The tools typically offer sequential clock-gating optimizations (see [2]). The granularity of this technique is usually at the level of a cluster of correlated blocks. Thus, its power-saving effectiveness could be better than the first method.

- The clock gating logic can also be inserted into the design manually by the RTL designers (typically as module-level clock gating) by instantiating library-specific ICG cells to gate the clocks of specific modules or registers. This is a coarse-grain clock-gating technique that can yield the most power saving. However, verification of the clock-gating logic using this technique is the most difficult to achieve, and is the focus of this paper.

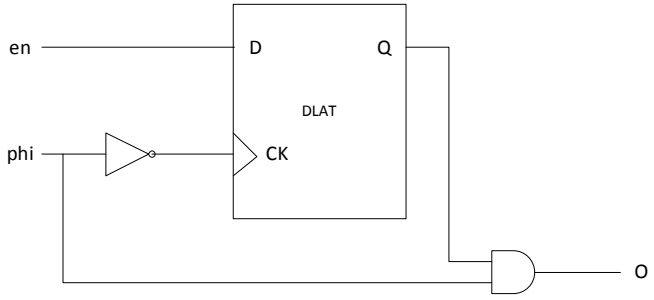

Figure 1. CLKGATE cell (Note: this is the ICG cell in our gate library)

Our mobile SoC chips support two types of clock-enable control for manually inserted ICGs. One uses a programmable register bit to drive the clock-enable signal. The other method is automatic clock gating, where hardware logic is introduced to detect whether there are any pending tasks, and to turn off a given clock if it is not needed. These two forms of clock-gating control can coexist for any given clock gate cell. For example, the internal bus-fabric logic might use automatic gating so that it is gated off until any bus master needs to use it, while the display controller might be more permanently gated off by system software until it is needed by the user, e.g. the user initiates a call by putting the phone on his/her ear.

Verifying the correctness of those two types of clock gating logic is a huge challenge for simulation methods to achieve thorough coverage and automatic checking. Thus the formal techniques are pursued to achieve better results. In this paper, we propose a method to verify clock gating logic by generating low level formal properties per flip-flop that uses any gated clock. This reduces the cone of influence for each property the formal tool has to process, thus increasing the probability a proof or counter-example can be produced within a reasonable amount of run time. We create scripts to automatically extract properties by doing RTL design structural analysis of logic connecting to the clock gate cell(s) using formal tool's design exploration API. For software-controlled clock gating, properties are generated to check that clocks for all gated flip-flops are parked at low when disabled. The details are discussed in Section III. For hardware-controlled automatic clock gating logic, additional properties are generated. First, the clock gate cell is instrumented with functional coverage properties. Second, for each clock gate, our script performs an RTL analysis of the clock tree to report the number of flip-flops that the gate drives. Third, our script performs a comprehensive structural analysis of the sources and destinations of the clock-gated flip-flops. The details of those three steps are discussed in section IV, V, and VI respectively.

II. PREVIOUS WORK

One DVCon paper [3] proposed to use a formal sequential-logic equivalence tool for verifying clock gating logic. We performed evaluations using the same tool and approach. It did not show promising results for our designs mainly due to the classic convergence difficulty and constraint complexity.

III. SOFTWARE-CONTROLLED CLOCK GATING PROPERTY GENERATION AND PROOF

A. Clock requirement

Each subsystem of our SoC has a common microarchitecture as shown in Fig. 2. The clock and reset generator (a.k.a. *clkgen*) is separate from the core logic (a.k.a *core*). Our low-power design methodology requires all the retention flip-flop clock pins to be parked at low before going into retention state. The reason for this clock attribute is proprietary and beyond the scope of this paper. See Chapter 13 of [4] for more hints. Regarding the proof of this clock property, first impressions suggest it should be trivial to do if all clocks generated from *clkgen* use certain standard ICG cell(s) as mentioned before. As long as the clock-enable signal is driven low, the gated output-clock signal is guaranteed to be low by design.

However, in reality this clock requirement puts a huge burden on verification. The enable signal controlled by the software often does not drive the clock enable signal directly. There could be some logic in between for various reasons. The challenges to verify this are multifold:

- Counter logic are added to keep clocks running for a number of cycles once the clock-enable bit is set to 0 by the software. This allows the pipeline to be drained.
- Many design blocks may have their own clock-divider and/or clock gating logic.
- The requirements could apply to multiple subsystems with different retention requirements. For example, 1 million flip-flops in four subsystems (three require full retention, one requires partial retention capability). There is no solution to verify this using a simulation approach.
- Each subsystem could utilize up to hundreds of derived clocks all of which have to prove this property.
- An unknown large number of functional configurations/modes—clock-source selection, clock divider factor value, different clock gating techniques.
- Third-party IP designs with internal logic that is very hard if not impossible to change. Furthermore there may be no document about the internal design.

As you can conclude from these challenges, it is impossible to cover all cases in simulation. Actually findings from our formal verification effort were used to guide simulation to add test cases to hit the potential issues. We resorted to a formal tool [5] that includes an API to extract all the registers inside the design.

B. Problem statement

The problem is simplified to the following pseudoassertion:

*assert "~resetn & ~enable [*N] |=> $flop_clock == 0"*

C. Basic strategy

We performed an experiment trying to prove the above property directly. The run time was intolerable and also many assertions were not fully proved. Eventually, we adopted a two-step approach, leveraging the black-box abstraction technique, to improve run time and to increase likelihood of proof of convergence. The original assertion is divided into two parts:

The first part is to prove all the clock outputs of the generator satisfy the parking-at-low property. The pseudocode of this generic property is shown below. To prove this assertion, the *core* can be treated as a black box.

*assert "~resetn & ~enable [*N] |=> $root_clock == 0 ##1 $root_clock == 0"*

The second part is to prove that if the core input clocks are parked at low, the derived flop clocks also park at low. The pseudocode of this generic property is shown below. To prove this assertion, the *clkgen* can be treated as a black box.

assert "$root_clock == 0 |=> $flop_clock == 0"

Example TCL code for creating targeted assertions is shown in Fig. 3. The *assert_clk_off* is a TCL procedure using a heuristic method for finding root clocks by pattern matching of signal-name strings to traverse fan-in logic. The formal tool was not smart enough to recognize clock divider logic for example. The assertions once generated can be proven, visualized, or edited from the GUI without recompilation. This is a must-have feature enabling us to debug and experiment using hundreds of generated assertions.

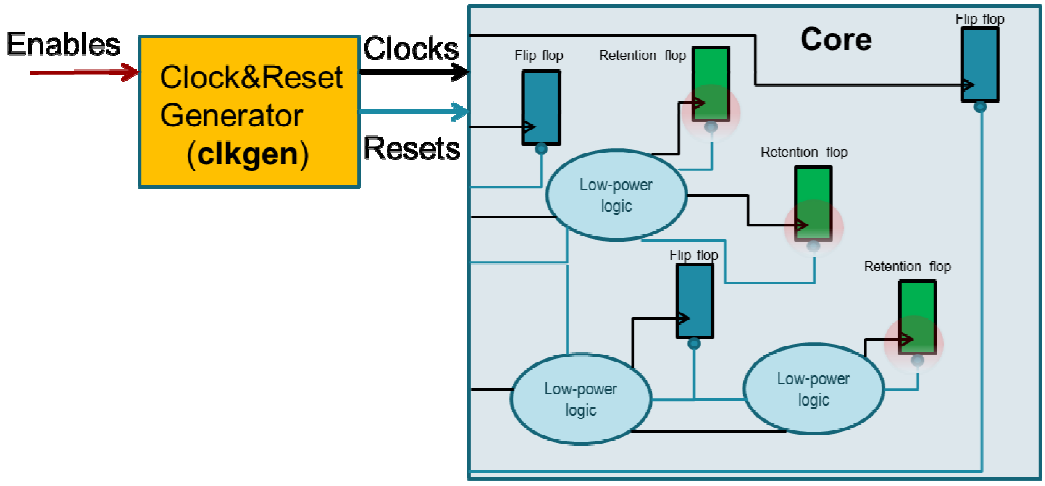

Figure 2. Subsystem common clocking micro architecture

```
foreach ff [get_design_info -instance u_kona_slaves -list flop -silent] {
  foreach sig [get_signal_info -bit_blast $ff -silent] {
    set clock_name [lindex [lindex [get_flop_info $sig -clock -silent] end] end]
    if {![info exists visited_dest_clocks($clock_name)]} {
      set src_clock_name [lindex [get_signal_info -equiv $clock_name] end]
      if {![info exists visited_src_clocks($src_clock_name)]} {
        set visited_src_clocks($src_clock_name) 1
          assert_clk_off $src_clock_name
      }
      set visited_dest_clocks($clock_name) 1
    }
  }
}
```

Find all flop signals
Expand to all bits
Get clock name
Visit each clock once
Trace to wire source
Visit each source once
Gen assertion or do fanin analysis

Figure 3. TCL code for creating targeted assertions

D. Results

The clock-parking-low proof process was carried out on four subsystems including three subsystems employing retention for all flops and the other subsystem employing retention for a subset of the flops. Third-party IPs were integrated into the subsystems. Many clock logic bugs/issues were uncovered.

- As a byproduct of doing formal proof, combinational loops were found inside a few blocks.
- We found some clocks were gated but toggling for a number of cycles during reset. It turned out this is a feature not a bug.
- Some clocks were found to take too long to stop. For example, one 2.4 MHz clock took a minimum of 6.5 us to stop.
- The clock for a random-value generator block was not guaranteed to park at low. This was a don't-care case because the corruption was on the random value. This finding led to the potential optimization of the full retention design into a partial retention design to save area and power. Similarly, another block was found to be unable to satisfy clock requirements but did not need retention.
- One third-party IP had an internal clock divider that was not guaranteed to park at low (requiring a software driver fix).
- Another third-party IP from a different vendor had about 7 problematic internal clocks belonging to these categories:
    - Clock divider: Below is an example of the clock-divider code. The waveform in Fig. 4 illustrates a counter example (CEX) showing that when the input clock stopped at 0, the divided output clock parked at high.

```
always @ (posedge utmifs_clk48 or negedge hreset_n)
  begin
    if (~hreset_n)
      begin
        count            <= 2'h0;
        utmifs_clk6      <= 1'b0;
        utmifs_clk_ls_sel <= 1'b0;
      end
    else
      begin
        count <= count +1;
        if(count == 3'b011)
          utmifs_clk6 <= 1'b1;
        else if(count == 3'b100)
          utmifs_clk_ls_sel <= (wpc_xcvrselect == 2'b10);
        else if(count == 3'b111)
          utmifs_clk6 <= 1'b0;
      end
  end
```

    - OR gate clock gating: Instead of using a standard ICG cell for clock gating, the OR gate was used for clock gating purposes. In this case, the clock is parked at high when disabled.

- PLL clock recovery from serial data: There were IP internal clocks, for example the USB PHY clock was recovered from serial USB RX data using a local PLL. This is not controllable from the common *clkgen* block.

In conclusion, all the potential issues found would be very difficult if not impossible to catch in simulation.

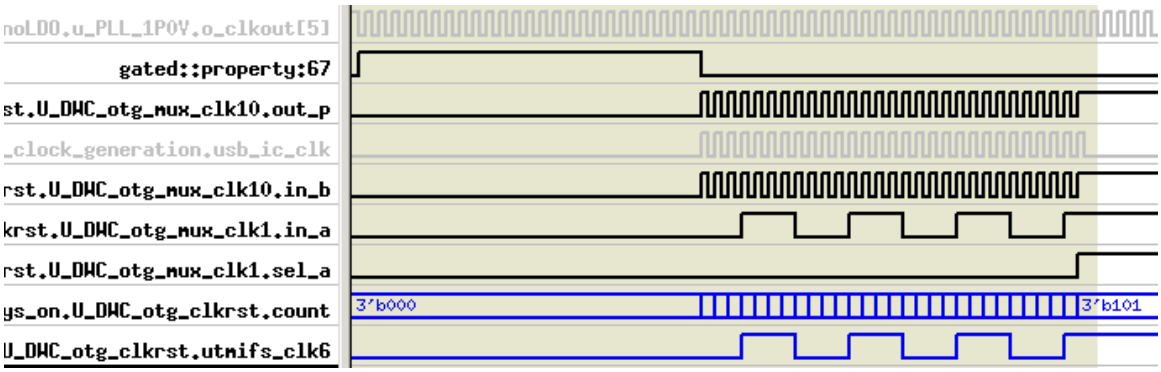

Figure 4. CEX of divider clock output parking at low

IV. HARDWARE-CONTROLLED CLOCK GATING PROPERTY GENERATION AND PROOF

A. Introduction

Many clocks in our design support a mode whereby they are gated by hardware logic automatically depending on the idle/active conditions generated from involved hardware blocks. The hardware clock gating logic uses an equation conceptually as follows:

$$CLKGATE.enable = \sim idle = \sim(idle_1 \ \& \ idle_2 \ \& \ ... \ \& \ idle_n)$$

where term $idle_i$ is the idle condition of subblock i sourcing its clock from this CLKGATE output. Sometimes the equivalent of the above equation is used instead in the RTL design.

$$CLKGATE.enable = active_1 \mid active_2 \mid ... \mid active_n$$

The potential bugs in this logic can be classified into one of the following categories:

1. The idle is always 1, which causes CLKGATE.enable to be 0, gating the clock. This issue is usually very easy to detect by simulation assuming there are test case(s) exercising this block and there are checks to detect the misbehavior once the problem occurs.
2. The idle is always 0, which causes CLKGATE.enable to be 1, never gating the clock. In other words, the gated clock is always free-running. How could this happen? One case we found is that sometimes $idle_j = \sim idle_k$. This is usually a harder problem to detect in simulation because it will not cause any data-integrity issue visible to the test bench checkers.
3. The idle is 1 sometimes when it should be 0. There could be many reasons for this to happen. One case could be that the idle equation is missing an idle condition term for a particular subblock. For example, let's use a simple 3-stage processor pipeline as an analogy. The instruction fetch subblock is idle and the execution subblock is also idle but there is still an instruction in the decode block being processed. In this case, the processor clock should not be gated.

4. The idle is 0 sometimes when it should be 1. The keyword here is "should be". How do we know when the clock should be gated unless there is a cycle-accurate and signal-accurate specification? We have never seen such specification for our SoC chips. Therefore, we claim this issue is a design choice or power-optimization issue not to be verified as part of functional verification.

B. Clock Enable Stuck-at Property Generation and Proof

We proposed a method that uses formal cover properties of CLKGATE.enable (stuck-at-0 case) and ~CLKGATE.enable (stuck-at-1 case) to detect issue #1 and #2 mentioned above using the formal tool. The two SystemVerilog cover properties are put inside a module and bound to the CLKGATE library cell module. In this way, all the instances of CLKGATE have these properties when the design is compiled and elaborated in the formal tool. The unreachability proof of the former exposes issue #2 and the latter exposes issue #1, respectively.

C. Coverage Model for Simulation

The coverage code used for formal proof can be used for simulation as well. The functional coverage data, collecting when a clock gate cell's enable signal is high and when it is low, provides information on how much the clock gating logic is exercised in our simulation regression suite. Below is a snapshot of the clock gate functional coverage report after we run our subsystem or SoC regression.

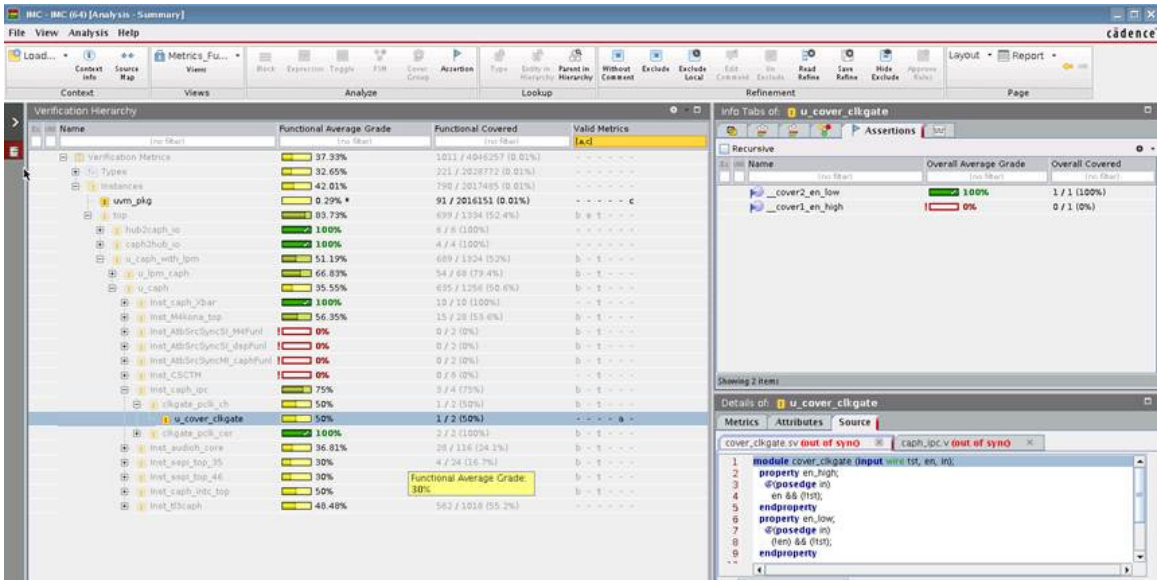

Figure 5. Clock gate functional coverage report

D. Results

Surprisingly many issues were found by just proving the stuck-at clock enable properties using the formal tool. Normally there should not be CLKGATE.enable stuck-at-0 case in the RTL design assuming the simulation testing achieves good coverage. The case we found is that the designer tied the clock of one of the unused blocks to 0, expecting the synthesis tool to optimize it away. Regardless of whether the result is correctly done by the synthesis tool or not, we think this is not a good practice and should be discouraged. One of the consequences is a low code-coverage number and time wasted figuring out what happened to this unused block. A parameter or ifdef macro should have been used instead to remove the code from being part of the DUT.

We found several cases of the CLKGATE.enable stuck-at-1 issue. One scenario is the enable signal is tied to 1 on purpose. It could be due to the fact that the IP-level clock gating is disabled to allow the clock to be controlled by the upper level. Another reason is simply that designer felt the autogating logic is too risky to be used and abandoned correcting it before tape out. There could also be true bugs with the idle or active-detection logic due to contradictory terms. Below is a real bug that exists in a third-party IP that had supposedly undergone thorough verification. The CLKGATE.enable signal is driven by edac_se_en which is defined as below. Obviously the *or* result of edac_se_live_not_zero and edac_se_live_is_zero is always 1.

```
// Enable signal for the SE and BE outputs
assign edac_se_en =
        edac_se_live_not_zero |      // Indicates that there are requests that were not granted yet
            sp_req_set        |      // Request set signal
            edac_ready;              // Ready signal after all bytes were granted by the internal arbiters
// Ready signal after all bytes were granted by the internal arbiters
assign edac_ready = edac_se_live_is_zero |      // All requests were granted
                    ~sp_req_r;                  // No request is active
```

V. CLOCK GATING LOGIC STRUCTURAL ANALYSIS

During the debugging of the clock gating property proof, we found that some clock gate cells drive very few flip-flops. This raises a concern of how efficient the clock gating logic design should be. To further analyze the issue, a script was created to analyze the RTL clock tree of each clock gate and to report the number of flip-flops it drives. The report is helpful for designers reviewing the effectiveness of their clock gating design.

Fig. 6 is a snapshot of the report generated by the script. The second column in the spreadsheet prints out the number of flops driven by the clock gate output listed in the first column. The designer can review it to decide whether this is intentional. We found cases where some clock gates were driving only a few flip-flops, revealing the potential for the clock gate design overhead to be more than its power saving. One third-party IP vendor told us they had a rule to put a clock gate in if it drives more than four flops but we found they violated their own rule.

1	u_kona_slaves.u_clkgate_axi.out	12114	u_kona_slaves.u_axislave_switch.upl301_a3bm_protocol_con	u_kon
2	u_kona_slaves.gen_clkgates.gen_clkgate_inst[0].u_clkgate_apb.out	2		
3	u_kona_slaves.gen_clkgates.gen_clkgate_inst[1].u_clkgate_apb.out	2		
4	u_kona_slaves.gen_clkgates.gen_clkgate_inst[2].u_clkgate_apb.out	2		
5	u_kona_slaves.gen_clkgates.gen_clkgate_inst[3].u_clkgate_apb.out	2		
6	u_kona_slaves.gen_clkgates.gen_clkgate_inst[4].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.u_ssp_0.u_sspi_core.pclk	
7	u_kona_slaves.gen_clkgates.gen_clkgate_inst[5].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.u_ssp_1.u_sspi_core.pclk	
8	u_kona_slaves.gen_clkgates.gen_clkgate_inst[6].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[6]	
9	u_kona_slaves.gen_clkgates.gen_clkgate_inst[7].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[7]	
10	u_kona_slaves.gen_clkgates.gen_clkgate_inst[8].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[8]	
11	u_kona_slaves.gen_clkgates.gen_clkgate_inst[9].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[9]	
12	u_kona_slaves.gen_clkgates.gen_clkgate_inst[10].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[10]	
13	u_kona_slaves.gen_clkgates.gen_clkgate_inst[11].u_clkgate_apb.out	2		
14	u_kona_slaves.gen_clkgates.gen_clkgate_inst[12].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[12]	
15	u_kona_slaves.gen_clkgates.gen_clkgate_inst[13].u_clkgate_apb.out	1	u_kona_slaves.u_kona_apb1_top.apb_clk[13]	
16	u_kona_slaves.gen_clkgates.gen_clkgate_inst[14].u_clkgate_apb.out	765		
17	u_kona_slaves.gen_clkgates.gen_clkgate_inst[15].u_clkgate_apb.out	78		
18	u_kona_slaves.gen_clkgates.gen_clkgate_inst[16].u_clkgate_apb.out	66		

Figure 6. Clock gating logic structural analysis result

VI. END-TO-END CLOCK GATING STRUCTURAL PROPERTY GENERATION AND PROOF

A. Introduction

Even with forementioned coverage properties proved or covered in simulation, we found there were still cases in which clock gating logic bugs can escape. The type of design where problems could exist is conceptually depicted in Fig. 7. Two RTL blocks communicate to each other not necessarily in a unidirectional data pipeline fashion but more in a master and slave style. *BLK1* is the master that initiates the transaction and *BLK2* is the slave that responds to complete the transaction. The clocks *CLK1* and *CLK2* for each block are independently controlled or gated but are sourced from the same clock *CLK*. There are two scenarios where a bug can occur: One is when *BLK1* is passing transaction information to *BLK2* but *CLK2* is gated. The other is *CLK1* getting gated prematurely, before a transaction is completed in *BLK1*. We generalized these two problems as the design in question not satisfying two formal properties. The details and the method of generating these assertions are explained in section VI B and VI C.

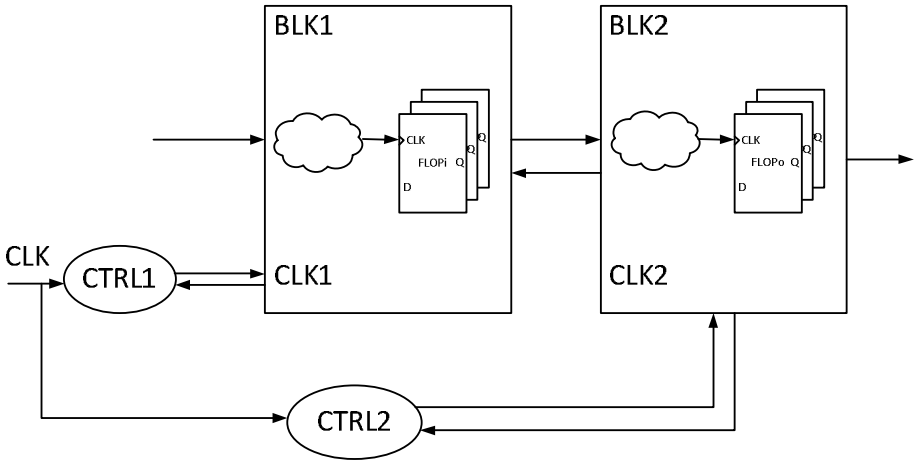

Figure 7. Master/Slave blocks separately clock-gated

B. Structural Analysis of Registers with Gated Clock

A script was created to automatically extract the structural relationship between the flip-flops in the CLK1 domain to the flip-flops in the CLK2 domain. This is similar to the clock domain crossing (CDC) analysis feature offered by commercial EDA tools but here the two clocks are actually synchronous to each other. No synchronizers are necessary between these two clock domains. Fig. 8 is a snapshot of the spreadsheet generated by the register fan-out analysis. The first two columns show all the "launch" registers and their clock names. The next two columns show the "capture" registers and their clocks, which are different from the "launch" clock. Not all flip-flops going from *CLK1* to flip-flops on *CLK2* are of interest to us. For example the data signals crossing the clock domain are don't-care since their validities are indicated by other control signals. Unfortunately there is no good automatic way to distinguish data signals from control signals. Therefore, a manual filtering process was used to screen the control signals for further property generation and proof step explained next.

C. Control Signals Clock Gating Property Generation and Proof

Once a filtered spreadsheet with all source flip-flops and destination flip-flops for the control signals is available, the next step of property generation and proof can be done. Our script parses each row of the spreadsheet to generate

two properties as shown below for the formal tool to verify. The two properties are meant to detect the two clock gating problems mentioned previously. The basic idea is that once the *D* input to the flip-flop using *CLK1* changes, *CLK1* and *CLK2* should be gated together in the next two clock cycles.

Property 1: *not ($changed(FLOPi.D_on_CLK1) && $changed(CLK1) ##1 $stable(CLK1) && $changed(CLK2) ##1 $changed(CLK2))*

Property 2: *not ($changed(FLOPi.D_on_CLK1) ##1 $changed(CLK1) && $changed(CLK2) ##1 $changed(CLK1) && $stable(CLK2))*

	LAUNCH FLOP	LAUNCH FLOP CLK	CAPTURE FLOP	CAPTURE FLOP CLK
2	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_3.u_dmac_gator.uaxi_xact_count_Inst.xact_count	u_lpm_fabric.axi_dma_clk
3	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_3.u_pl330_dma_kona_dmac.upl330_engine.upl330_axi.upl330_r	u_lpm_fabric.axi_dma_clk
4	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_3.u_pl330_dma_kona_dmac.upl330_engine.upl330_axi.upl330_r	u_lpm_fabric.axi_dma_clk
5	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_3.u_pl330_dma_kona_dmac.upl330_engine.upl330_axi.upl330_r	u_lpm_fabric.axi_dma_clk
6	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_3.u_pl330_dma_kona_dmac.upl330_engine.upl330_axi.upl330_r	u_lpm_fabric.axi_dma_clk
7	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_3.u_pl330_dma_kona_dmac.upl330_engine.upl330_axi.upl330_r	u_lpm_fabric.axi_dma_clk
8	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.axi_blocker_pcie_master.rd_beat_cnt_r	u_lpm_fabric.master_axi_5_4_clk
9	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.axi_blocker_pcie_master.rxact_cnt	u_lpm_fabric.master_axi_5_4_clk
10	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_ful	u_lpm_fabric.master_axi_5_4_clk
11	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_ful	u_lpm_fabric.master_axi_5_4_clk
12	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_ful	u_lpm_fabric.master_axi_5_4_clk
13	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_ful	u_lpm_fabric.master_axi_5_4_clk
14	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_ful	u_lpm_fabric.master_axi_5_4_clk
15	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_ful	u_lpm_fabric.master_axi_5_4_clk
16	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_fw	u_lpm_fabric.master_axi_5_4_clk
17	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_fw	u_lpm_fabric.master_axi_5_4_clk
18	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_rev	u_lpm_fabric.master_axi_5_4_clk
19	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_amib_sys_sw.u_r_master_port_chan_slice.u_rev	u_lpm_fabric.master_axi_5_4_clk
20	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_maskcntl.rd_cnt	u_lpm_fabric.master_axi_5_4_clk
21	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_maskcntl.reg_mask_r	u_lpm_fabric.master_axi_5_4_clk
22	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.full	u_lpm_fabric.master_axi_5_4_clk
23	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_asel	u_lpm_fabric.master_axi_5_4_clk
24	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_ids[0]	u_lpm_fabric.master_axi_5_4_clk
25	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_ids[1]	u_lpm_fabric.master_axi_5_4_clk
26	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_ids[2]	u_lpm_fabric.master_axi_5_4_clk
27	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_ids[3]	u_lpm_fabric.master_axi_5_4_clk
28	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_ids[4]	u_lpm_fabric.master_axi_5_4_clk
29	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_ids[5]	u_lpm_fabric.master_axi_5_4_clk
30	u_kona_fabric.u_kona_fabric_5_3.axi_blocker_apps2hub.arid_r	u_lpm_fabric.axi_switch_clk	u_kona_fabric.u_kona_fabric_5_4.u_mst_sw2.u_asib_pcie.u_asib_pcie_rd_spi_cdas.spi_valid	u_lpm_fabric.master_axi_5_4_clk

Figure 8. Register fanout analysis report

D. Results

We applied the above techniques on several subsystems and IP blocks in our SoC chip, which exposed bugs that would otherwise be hard to find. Two example scenarios that violate Property 1 and Property 2 respectively are shown in Fig. 10 and Fig. 11. The design logic that contains the bug is captured in Fig. 9.

As shown in Fig. 9, the blocks in red are controlled by *aclk* and correspond to *BLK1* in Fig 7. The blocks in green are controlled by *PCLK* and correspond to *BLK2* in Fig. 7. *aclk* and *PCLK* are from the same source clock i_func_clk, but they are gated independently by different CLKGATE cells and control logic.

In the CEX shown in Fig.10, a violation of Property 1 is captured. An AXI write transaction was triggered by the AXI master to execute the write to slave ipc_caph_wrapper in *BLK2*. Flip-flop psel_i[3]'s input signal nxt_psel[3] was asserted at time 91 to reflect this transaction. Flip-flop psel_i[3]'s input clk aclk stopped at time 93 for one cycle after nxt_psel[3] changed value. As a result, psel_i[3] was not asserted at time 93. Then nxt_psel[3] was also deasserted at time 93. Flip-flop psel_i[3] didn't get asserted after its aclk came back at cycle 95 because nxt_psel[3] was already deasserted. On the other side, PCLK was still running. However, the APB slave ipc_caph_wrapper missed one transaction due to the stop of aclk. APB signals such as PCLK, PSEL, and PENABLE at the ipc_caph_wrapper boundary never got asserted.

The other CEX shown in Fig. 11 is the scenario where aclk (*CLK1*) didn't stop but PCLK (*CLK2*) stopped two cycles after nxt_psel[3] was asserted. In this case, psel_i[3] and PSEL in ipc_caph_wrapper caught the change of nxt_psel[3] because of the running of aclk. However, due to the stop of PCLK, pSEL_csr_dly didn't get asserted and

thus the gated clk PCLK_csr_gated never ticked even when PCLK started running again. As a result, the transaction wasn't caught by ipc_caph_wrapper either.

These types of bugs are very hard (if not impossible) to catch by simulation. Using our results as guidance, dedicated simulations were generated to duplicate such CEX scenarios and the bugs were verified by simulation and confirmed by the designers. Moreover, sometimes these types of bugs are also hard to fix. For the above-illustrated bug, the designers couldn't find an appropriate solution within the project schedule to fix the issue. As a workaround, a suboptimal software-controlled clock-gating strategy was used instead to prevent the bug scenario from occurring in the real world.

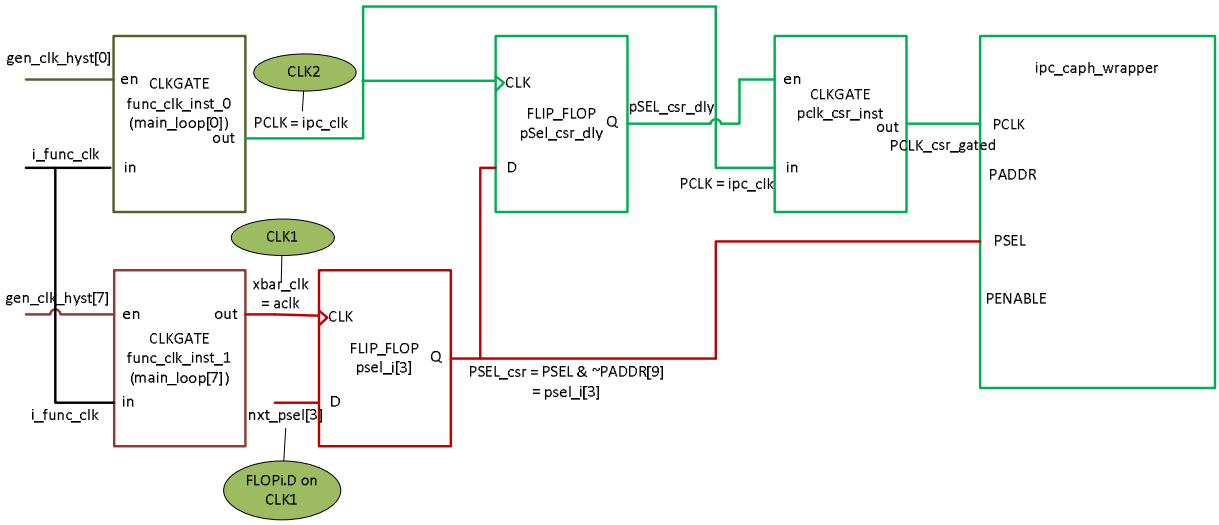

Figure 9. Design logic violating property 1 and 2

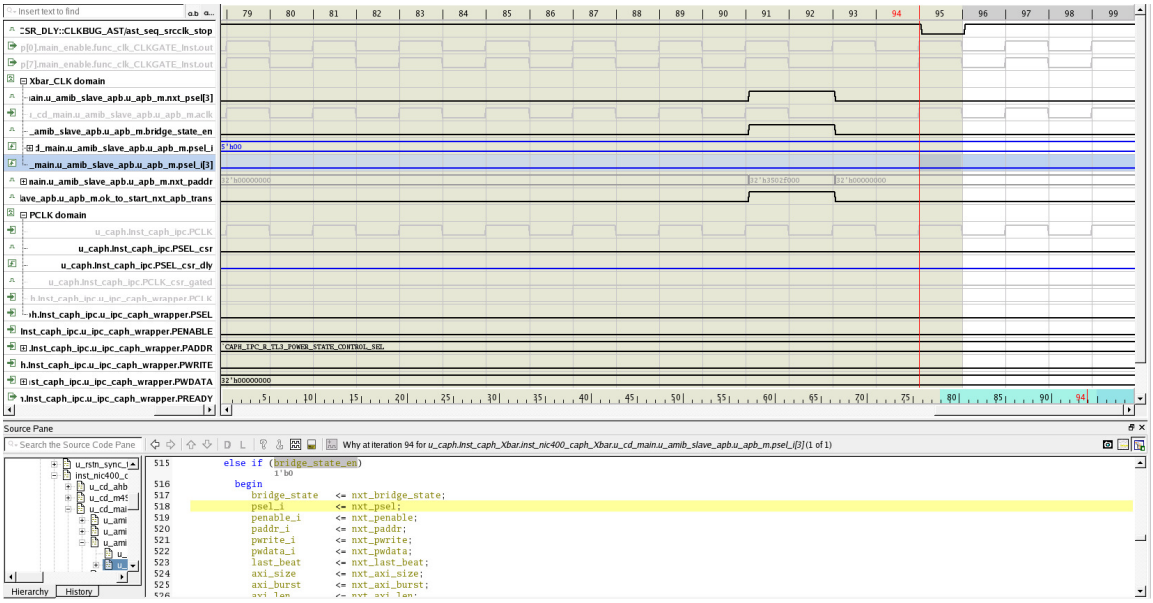

Figure 10. CEX for violation of property 1

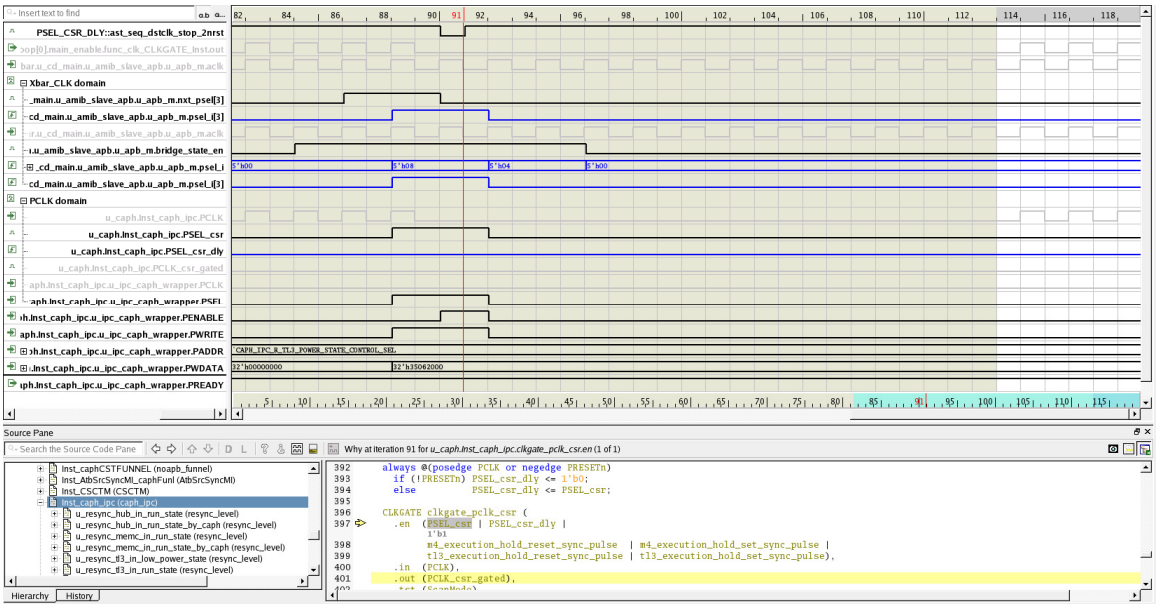

Figure 11. CEX for violation of property 2

VI. SUMMARY

In summary, to maximize dynamic power reduction, our SoC design adopted an aggressive clock-gating design methodology using manually inserted ICG as close as possible to the root of the clock tree. As a result, this posed great challenges to traditional verification methods. We created a systematic clock-gating verification methodology, combining simulation, structural analysis, and formal property proof. The checks we did for each type of clock-gating logic are summarized in table I. Interesting results were discovered using our method. Further improvements are needed to reduce formal tool run-time and to reduce "noises" generated from the automated flow.

TABLE I. CLOCK GATING VERIFICATION SUMMARY

Clock-Enable Control Type	Verification Methodology
Software controlled	Check clocks for all gated flip-flops are parked at low when disabled
Hardware controlled	Check enable inputs for all clock gate cells are not stuck-at-0 or stuck-at-1
	Analyze number of flip-flops that each clock gate drives
	Do structural fan-in and fan-out analysis of each clock-gated flip-flop and automatically generate properties to check cross-domain errors

ACKNOWLEDGMENT

We would like to thank Jennifer Hwang, Jing Li, Kang Xiao for their support and encouragement to finish this paper. We thank Jing Li for spending time to proof-read the drafts and provide valuable feedback.

REFERENCES

[1] http://en.wikipedia.org/wiki/Clock_gating.
[2] http://electronicdesign.com/power/reduce-power-chip-designs-sequential-clock-gating
[3] Syed Suhaib, Scott Fields, Prosenjit Chatterjee, "A Formal Verification App Towards Efficient Chip-Wide Clock Gating Verification", *DVCon,* 2014.
[4] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, Kaijian Shi, "Low Power Methodology Manual", *Springer*, 2007.
[5] Cadence JasperGold, http://www.jasper-da.com/.