A Formal Verification App Towards Efficient, Chip-Wide Clock Gating Verification

Prosenjit Chatterjee
NVIDIA Corp.
Santa Clara, CA
PChatterjee@nvidia.com

Scott Fields
NVIDIA Corp.
Santa Clara, CA
sfields@nvidia.com

Syed Suhaib
NVIDIA Corp.
Santa Clara, CA;
ssuhaib@nvidia.com

***Abstract*— Ensuring correct clock gating has been a major verification challenge. A chip can have dozens of clock gating islands, where each island has a control bit to enable/disable clock gating. We present an automated methodology for exhaustive clock-gating verification using Sequential Equivalence Checking (SEC) analysis. This automated methodology is enabled via an SEC formal verification "App". This "App" performs various optimizations automatically to achieve deeper proof bounds or even full proofs, in many cases, taking advantage of the symmetry of the setup. We apply this methodology across the chip to illustrate its usefulness. We found multiple clock gating bugs across many projects using this approach, where over half of these were found after supposedly high simulation coverage of the design.**

Keywords—clock-gating verification; low power; Sequential Equivalence Checking; SEC; formal; formal analysis; formal apps

I. INTRODUCTION

Advances in the mobile technology are causing a shift on how designs are being architected these days. Low power and high performance in complex systems and processors, especially those targeted toward mobile applications, is one of the biggest concerns surrounding the industry. A plethora of devices, such as mobile phones, tablets, laptop computers, as well as wearable digital devices, are some of the examples of technologies where either custom processors or off-the-shelf System-On-Chips (SOCs) are used and rely deeply on prolonged battery life. Hence, processors designed these days are focused highly on such use cases.

Clock gating is one of the common strategies being used for reducing dynamic power consumption of such designs, but it is not without risk [1]. A chip is traditionally divided into multiple functional islands (units) where each island may have one or more clock domains. The use of multiple clock domains does increase the complexity; however, it helps in reducing the power dissipation on execution.

Clock gating can be classified into functional clock gating and non-functional clock gating [1]. Functional clock gating refers to gating of the clock that turns off functionality. This type of clock gating is part of the functionality and is required for correct execution. Hence, turning this clock gating off would yield incorrect results. On the other hand, non-functional clock gating refers to the case where gating of the clock does not affect the functionality. In other words, the clock-gate logic is created in such a way that irrespective of the clock, the functionality is still correct. Usually, the clock enable of such clock-gater is designed in such a way that it is activated based on the presence of valid data, and in case of no-data, the output result is not affected. This type of clock-gater is solely used for power savings. In this paper, our focus is on verification of non-functional clock-gating.

Verification is known to be one of the biggest challenges in the design cycle of a chip. Ideally verification is started very close to the design phase. However, since clock gating is one of the late aspects of the functionality added into the design, verification of correct clock gating becomes even more burdensome. As a result, clock gating verification is left as one of the later things to do.

Traditional simulation and coverage closure techniques are necessary but insufficient due to difficulty in evaluating timing corner cases. Furthermore, missed terms in the logic of clock gating enable are invisible in code coverage. Writing a comprehensive set of functional coverage goals for clock gating is difficult, and exercising such goals can take a lot of time and become engineering resource intensive. Secondly, when bugs are found in simulation, debugging the source of failures becomes very time-consuming. On the other hand, traditional formal property verification tools can be used where one writes assertions on correct functionality irrespective of clock gating on or off. This requires a large effort in getting the necessary set of clean environmental constraints, which carry a risk of over-constrained assumptions [5,6].

Given the late verification push on clock-gating, bugs do get slipped into the design. Such bugs due to clock gating can result in flawed functionality, including the erroneous shutdown of whole regions of the chip. To circumvent the bugs and avoid costly ECOs, software work-arounds are implemented which un-necessarily increase the complexity of the software. Also, to get around the bugs, large sections of the chip may be turned-off, which may result in loss of functionality. Given the target market of the chip, a big loss on power budget in most cases becomes un-acceptable [7].

We present a methodology implemented by a formal verification "App" that enables clock gating verification of non-functional clock gaters in an effective and timely manner. We show how this methodology can be applied at the chip-level or even for larger blocks by using divide and conquer approach. The methodology is based on comparing two copies of the RTL, where in one instance, the clock is un-gated and in the other instance the clock is gated. Basic combinatorial logic equivalence checking tools, such as LEC, Formality, etc., are inadequate since our clock gating schema creates different state elements in the "new" clock gated design vs. the original un-gated design. As a result, major control path registers do not match between the clock-gated version and un-gated version of the design. Furthermore, combinational equivalence checking tools do not support transactional equivalence [8]. Transactional equivalence in this scope means that a given transaction between the two instances of the RTL provides the same results.

A clean recipe to achieve a sequential equivalence checking [9] between a clock-gated RTL and an un-gated version of RTL requires the following: (1) Ability to express temporal properties (SVA), (2) Ability to express constraints at the primary inputs as well as internal signals, (3) Ability to abstract initial state of the DUT for a subset of registers, (4) Ability to abstract logic functions, (5) Clean approach to automatic assume guarantee reasoning, (6) Ability to express transactional equivalence and (7) Ability to automatically populate the mismatching traces between the two instances of the RTL and point to the root cause of the mismatch, which speeds up the debugging capability of the design. The combination of above recipe requires that we need a sequential equivalence checking capable tool with formal property verification (FPV) features. For this methodology, we use JasperGold [11], which has the capabilities for formal property verification, along with an "App" that understand/creates the environment suitable for clock gating verification.

In this paper, we show how this methodology is implemented and highlight key features of the "App" that helps in effective clock-gating verification. We show how we implement this methodology at the chip level, and we provide results of the case-studies. One thing to note is that this process helps in verifying the correctness of non-functional clock-gating only.

The paper is organized as follows: Section II describes the ports and relevant information of a common clock gater. In Section III, we discuss the verification setup and methodology. Section IV discusses how this approach is applicable to large designs and techniques to handle large state space issues associated with formal verification. Section V discusses the application of this methodology as well as results, followed by conclusion and future work in Section VI.

II. Clock Gater

The interface of a traditional clock gater consists of four ports as shown below:

```
module cg_ul ( clk_ul, gclk, func_en, cg_cya_disable );
```

where *clk_ul* is the gated clock, which is output of the module, *gclk* is the un-gated global clock, *func_en* is the enable of the clock signal, which is used as enable signal with the *gclk*, and *cg_cya_disable*, which is the CYA disable that turns off clock gating. The *cg_cya_disable* signal equates the *gclk* to *clk_ul*. As a fail-safe, special defeature bits (controlled by software) are put in place to protect against the late bugs due to clock gating. These defeature bits are connected to *cg_cya_disable* ports. In some cases, the *cg_cya_disable* port is connected to 0 for non-functional gaters, which results in clock gating always being enabled and may become troublesome if/when bugs arise in the logic. The source of the bugs is usually in the logic driving the *func_en* signal.

III. Setup

A chip consists of multiple units with various functionalities that fulfil multiples tasks. Furthermore, each unit can potentially have multiple clock gating domains which can be enabled or disabled based on availability of inputs. Recall that in this paper, we address verification of non-functional clock gaters. Hence, as a first step, for each unit all non-functional clock gaters are identified. Ideally, non-functional clock gaters should be defined by a different module name than the functional clock gaters such that the usage is clearly expressed. A formal tool can be used to list out all the instances of such non-functional clock gaters. As mentioned in the previous section, the *cg_cya_disable* bit controls the enabling or disabling of the clock gating. We also look at the signals connecting the corresponding *cg_cya_disable* port for all such non-functional clock gaters because, as mentioned earlier, in some cases this bit mistakenly gets connected to 0. Ideally, there is a defeature bit connected to the *cg_cya_disable* port.

Next, we create two instances of the same RTL, where in one instance (calling it un-gated version), we force the *cg_cya_disable* port of all non-functional clock gaters to 1 (to disable clock gating). This can be done by adding a cut-point to each *cg_cya_disable* that makes it free to take any value, and then force the value 1 (disable) by adding an assumption. On the other hand, in the clock gated instance of the RTL, we force the signal to be random and stable throughout the proof. This would give us the ability to ensure that all values in the clock gated instance are exhaustively verified. Again, this can be done by adding a cut-point, and an assumption which ensures that the value remains stable. Figure 1 illustrates this setup, where we have two instances of RTL (sfv0 and sfv1), where sfv0 is the un-gated instance and sfv1 is the gated instance.

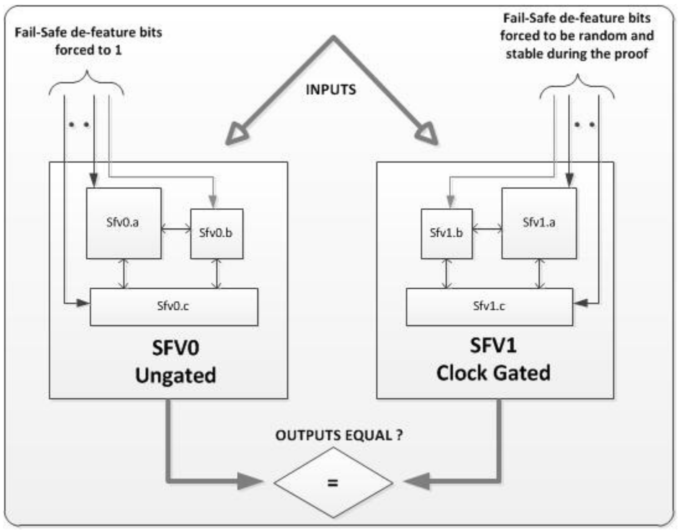
Fail-Safe de-feature bits forced to 1
Fail-Safe de-feature bits forced to be random and stable during the proof
INPUTS
Sfv0.a
Sfv0.b
Sfv0.c
SFV0 Ungated
Sfv1.b
Sfv1.a
Sfv1.c
SFV1 Clock Gated
OUTPUTS EQUAL ?
=

Figure 1. Formal Verification Setup

The output signals can be classified into control and data signals. For each of the output control signals, assertions are added which state that signals in one instance are equal to the signals in the other instance.

Below is an example assertion that is used for control signals:

```
asrt_CORE_unit_control_output_ctrl_a: assert property
    (sfv0.output_ctrl_a == sfv1.output_ctrl_a);
```

For each output data signal, we append the appropriate control signals as preconditions (as applicable). Note that the corresponding data signals between the two instances are allowed to be different as long as the corresponding control signals are low. Below is an example assertion that is used for data signals:

```
asrt_CORE_unit_data_output_signal_a: assert property
    sfv0.output_control_a |->
        sfv0.(sfv0.output_data_a == sfv1.output_data_a);
```

Another important factor to consider is the reset values of the flops. The reset sequence can be configured in various forms: (1) Run the proofs with design initialized flops, and keep the uninitialized flops as they are. This is the ideal case, were we would catch not only clock-gating bugs, but also bugs due to x-propagation from un-initialized flops. (2) Run the proofs with design initialized flops, and force all un-initialized flops to be equal in both the instances. In this case, we only focus on clock-gating bugs. (3) Run the proofs with design initialized flops, and force the un-initialized flops to be 0 or 1. This is a subset of previous case; however, it might help with the depth of the proof runs. (4) Run the proofs with all un-initialized but equal flops. This allows exploration of more states; however, one might end up debugging false failures.

IV. Approach

Our methodology is based on formal verification techniques, and it has the issue of state space explosion. Given this issue, a complete chip setup with millions of gates and flops would not be tractable for formal tools. Hence, the chip is divided into smaller units which act as functional partitions of the design. These partitions are physical as well as logical decompositions of the chip, resulting in self-contained units. Some of the larger units of the design are further broken down into smaller sub-units based on functional partitioning. The partitioning is done based on the segregation of the functional behaviors across the sub-units. Ideally, one must ensure not to create very small "unnecessary" partitions that would result in the addition of many assumptions. Furthermore, these extra partitions would result in unwanted setups, requiring additional engineer and compute time and resource overhead.

A wrapper in the form of a formal verification "App" is created containing the two instances of the RTL with the inputs connected, and assertions generated for each of the output signals (as described in the previous section). This process is further simplified by the SEC "App" that takes in RTL and automatically creates the wrapper with connections surrounding the two instances of the RTL. The "App" also generates the necessary assertions on the output signals. In this case, the "App" is part of the Jasper Gold verification environment. Hence, it uses the environment's existing proof engines, which take advantage of the symmetry between the two instances of the DUT and help in achieving better proof convergence [11].

Furthermore, various techniques are employed to circumvent the state-space issue of the formal verification problem. Abstraction is one of the key techniques used to address the state space issue. We identify and abstract out sub-units in the design that have no clock gating. This is done by black-boxing them in the setup. The following steps are used: (1) Identify subunits within the design which are oblivious to clock-gating. (2) Black-box these sub-units in both the instances of the setup. This will make the outputs from these black-boxed sub-units as free signals. (3) Add assumptions on the output signals from black-boxed sub-units to be equal. This ensures no false failures are seen downstream. (4) Add assertions on the input signals to the black-boxed sub-units to be equal. This will ensure that the input signals are equivalent, and no bugs arise due to clock gating. Figure 2 illustrates this abstraction. One important thing to note is that if the sub-unit being black-boxed is driven from a gated clock, then care must be taken to ensure that none of the enable signals to the black-boxed unit must be high when the gated clock is low.

Another helpful technique used with this methodology is assume-guarantee approach. This is done by adding intermediate cut-points at locations, where the signal in the un-gated version and clock-gated version of the RTL are equal. There are two steps to this procedure: (1) Add an assertion to ensure that these points are equal for the upstream logic. (2) If (1) is proven true, then add assumption that both signals are true for the down-stream logic. The SEC "App" easily enables this in the same environment internally. Furthermore, addition of cut-points at intermediate points helps to prune away the driving logic. Furthermore, cut-points can be added in one or both instances. The ideal choice would be adding cut-points to both instances; however, this may result in false failures. One of the reasons might be that the downstream logic was dependent on the behavior that got abstracted out as a result of the cut-point. Hence, a single side cut-point may help get away from this issue. Alternately, one may add helper assertions for each of the proven intermediate point that may also help in getting better proof convergence.

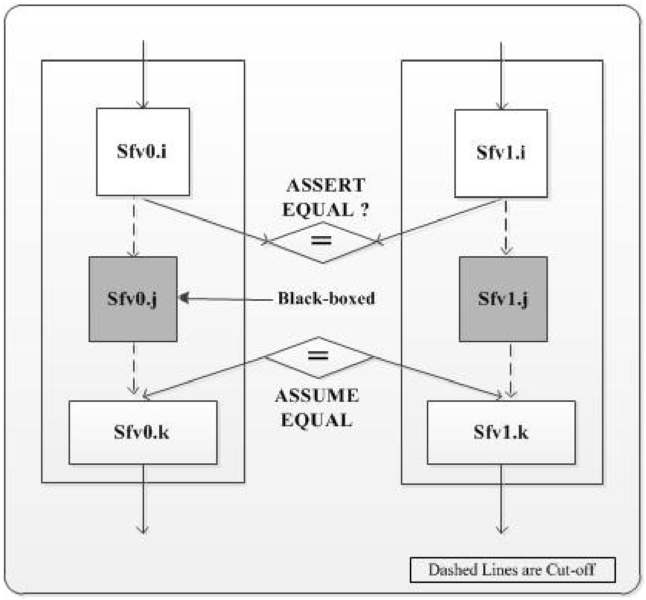

Figure 2. Black-box Setup

When multiple cut-points are added in the design, ordering the evaluation of cut-points helps with better proof convergence. The idea here is to prove the easiest assertions first. These assertions usually have the smallest cone of influence (COI) compared to other properties in the design. Once proven, these helper assertions are used to prove the next set of properties. This ordering of assertions to enable an incremental proof convergence is referred to as "levelizing". This is done based on Topological sort starting from output signals and measuring the depth to the corresponding primary inputs. The SEC "App" automates this process internally, and helps automatically with proof convergence with less user interaction.

V. Application and Results

The SEC "App" and the approach were applied successfully to various chips across Nvidia. In this paper, we will discuss the results of two different chips, one from GPU and another one from Tegra, which is a mobile processor [12]. Note that we abstract out the details of the chip due to confidentiality.

In the GPU chip, this approach was applied to multiple units where each unit was broken into up to 5 different sub-units based on the functional partitions. In total, there were 250 different testbench setups. The break-down was such that the setup size was amenable to the formal verification tool. For each of the sub-units, all FIFOs and RAM modules were black-boxed. Inputs to these were proven equivalent and primary outputs of the subunits were proven under the precondition that output of the black-boxed modules were proven. The FIFOs were proven separately. The initial values of the counters and major finite state machines were abstracted out to help achieve deeper proof depth. A wrapper was created in an internal macro based language where two instances of the sub-unit were declared, and the primary input signals were automatically connected. The setup time for each sub-unit was 0.5 days on average. Around 1-4 weeks were spent on iterating over constraints and RTL fixes. OVL assertions per primary output were generated in the following form:

```
wire cond_1x = sfv0_vld[0] == sfv1_vld[0]
assert_always #(0,0,"equiv") equiv_bit_1x (clk, reset_, cond_1x);
```

Table 1 illustrates sample results for some of the sub-units of a unit in GPU. It lists the approximate size of each unit along with maximum pipe stages as well as the clock gating domains in it. The table also lists the percentage of assertions that saw full proofs and the lowest cycle bounds for the worst case for assertions with bounded proofs.

Table 1 Partial Results of SEC Application to GPU

Name	Flops	Pipe Stages	Clock Gate Domains	Proofs (full)	Cycles	Bugs
Unit A	7k	2	3	83%	16+	4
Unit B	40k	5	4	7%	16+	0
Unit C	150k	13	8	5%	19+	6
Unit D	56k	15	7	9%	6+	1

A total of 32 bugs were found related to clock-gating where simulation found 50% of the bugs, and other 50% were found by SEC approach. The simulation had started much earlier than SEC and continued until DV signoff. From a post-mortem analysis, simulation could have found 50% of the bugs found by SEC, but they would have appeared in the form of post FNL or post silicon bugs. The remaining 8 bugs found by SEC were very difficult–to-hit corner cases and would have required an additional 9 man-months of effort.

In the Tegra project, 52 setups were created, which were spread across 20 verification and design engineers. All the setups were up and running within the week. The smaller units had around 5k flops, whereas the larger units were over 250k flops. Hence, larger blocks were broken down into smaller sub-sets to become friendlier for formal tool. A lot of features from the SEC "App" were used to speed up the verification time. The SEC "App" allowed quick and automatic mapping of the inputs and outputs. As some of the setups were created much earlier than the tool, a special mode was developed in the "App" called the "attached" mode which could read in existing setups and automatically map the inputs/outputs. Table 2 illustrates a subset of results from SEC "App" vs running the setup in FPV. For some of the designs, much improvement was realized, and this was due to the fact of using automated abstraction along with "levelizing", which proved easier properties first, then used them as assumptions to prove downstream properties. For some of the difficult models, the cut-points were applied to all the flops and evaluated for every level. Note that an automated tcl script was written that removed cut-points for the failing properties. After removal of these cut-points, the script would automatically rerun the proofs until all the properties were proven or bounded proven. For each of the units, we had a different acceptable bound point dependent on the number of cycles it took to hit the deepest cover property.

Table 2. Subset of Results from SEC "App" vs FPV "App"

Name	Flops	Clock Gate Domains	FPV Proofs (SFV)	SEC "App" Proofs (SFV)
Unit A	25k	4	30%	100%
Unit B	35k	4	100%	100%
Unit C	25k	8	70%	100%
Unit D	35k	9	40%	60%
Unit E	45k	14	25%	65%

Debugging capabilities of the "App" were very beneficial. It allowed viewing two instances of the RTL side by side and quickly identifying the mismatches. A script was developed which would automatically plot the mismatching signals in a breath-first search manner. This help save about 8-10 minutes of debugging time per failure. Over 40+ bugs were found via this approach where more than half were discovered after many months of simulation.

To improve upon the verification time, black-boxing of various unwanted sub-units was easily done. The below commands would automatically populate the assertions on all inputs of the black-boxed module and add assumptions on all its outputs.

```
check_sec -map -auto -spec sfv0.a.ram -imp sfv1.a.ram -type
bbox_input -tag ram_input
check_sec -map -auto -spec sfv0.a.ram -imp sfv1.a.ram -type
bbox_output -tag ram_output
```

Furthermore, cut-points were added as discussed in Section IV to improve the proof depth. These were added by identifying which flops were to be equivalent. Below is a sample command used to add the cut-points.:

```
check_sec -map -spec sfv0.b.large_counter -imp sfv1.b.large_counter -
tag countdown_count -with_attr cutpoint
```

We were able to achieve full proofs for many of the properties using these techniques.

There were various types of bugs found. Most of the bugs fell into one of the three buckets:

1. Missing terms: A common bug across many sub-units was that a term was missed from the *func_en* term which was used as the enable for the clock gater. Given the following example,

   ```
   clk_en1 = vld_1 || vld_2 ~~|| vld_3~~
   func_en = defeature_clk_en1|| clk_en1
   ```

   vld_3 was mistakenly omitted from *clk_en1* which was driving the logic related to *vld_3*.

2. Bad clocks: Incorrect clock enable was hooked in to *func_en* of the clock gater by mistake.

   ```
   @(posedge clk_en_a1) vld_a1_data1 <= a1_data1;
   @(posedge clk_en_a~~2~~) vld_a1_data2 <= a1_data2;
   ```

3. Hang Case: Clock enable was stuck due to bad logic driving it. As a result, valid data was not getting propagated.

   ```
   @(posedge clk_en_stuck) vld_a3 <= vld_a2;
   ```

Each of the units had a signoff checklist. The checklist had a variety of items, and was not limited to the following:

- Resolve all counter-examples
- Re-confirm that all relevant defeature bits are accounted for and exercised in the setup file.
- Re-confirm that all outputs have equivalence properties.
- Ensure that reset states for un-initialized flops is random to catch X-propagation issues.
- Increase the per-property runtime to at least 10 hours
- Review the input constrains with the designer or enable them in simulation.
- Prove all assertions to an acceptable depth

VI. RELATED WORK

Various approaches to clock gating verification have been applied in the industry. [4] presents a solution where a sequential equivalence checking of clock gating can be reduced to a combinational equivalence checking problem based on specific preconditions of the design. They formulate a set of theorems for equivalence checking and transform the clock-gating in sequential space to combinational space. With huge industrial designs, each clock-gating point would add loops to the design making it difficult to use their approach. Furthermore, this transformation can be often time-consuming. Our work targets specific clock-gating (non-functional) however is very efficient and does not require any transformations. Similarly, [10] also looks at RTL transformation and using combinational equivalence checking for proving clock gating correctness.

On the other hand, [8] describes a framework for sequential equivalence checking across arbitrary design transformations. Their framework targets transformations that arise from changes for performance, power, etc. for a design. Their framework takes in two gate-level design representations, generated from traditional synthesis process, and checks equivalence based on their outputs. Our approach is different from theirs in the way that they run a redundancy removal algorithm which identifies equivalence classes of gates that contain a representative gate for each of those classes and construct a speculatively reduced model. They create a mitre over each gate and its representative and prove that miter is unreachable. Once proven that mitre is un-reachable, they replace and merge the gates to reduce the model. If a mitre is proven to be reachable, then they have to select a new representative gate, and repeat the steps. Hence, their process is very iterative and time-consuming.

Model checking based sequential clock-gating is proposed in [5]. In their approach, they use a model-checking based approach to identify sequential clock-gating opportunities. Using model-checking they identify intra-register relationships across clock boundaries and map this relationship to temporal properties. Based on the results of the model-checker, they identify potential sequential clock-gating. This approach is useful when clock-gating is unknown and one needs to identify where it can be applied. In most industrial cases, clock-gating is already added during the design cycle. Furthermore, on larger designs this can be challenging and time-consuming.

VII. Conclusion and Future Work

In this paper, we present a methodology for verification of non-functional clock gaters that are used for dynamic power saving of the chip. These non-functional clock gaters can also be referred to as “second-level clock gating” as they are instantiated in the RTL and gate downstream non-gated clock nodes, which are used to gate off circuits that have low utilization and are not being used during functional execution. This methodology is applied at the chip-level by dividing the chip to smaller units/sub-units that can be read by the formal tool. This approach requires fewer assumptions than the traditional formal property verification approach because assumptions are only needed in case of mismatches (if they are not real clock gating bugs.) between the two instances of the RTL. Every assumption increases the risk for over-constraint, in which an incorrect output in one instance is made to match the incorrect output in the other instance. This is one of the primary reasons for having a limited number of assumptions.

Automated abstraction techniques were used, including black-boxing and cut-points, which resulted in full-proofs for many of the sub-units. Furthermore, the approach was very efficient and required on average 2-3 weeks per unit, and it was very effective in finding bugs. All these features were brought together in the SEC “App”, which was easy to use by designers who were not experts in formal verification.

This approach can be extended to block-level clock gating as well, where an entire unit is turned off when the unit is not active. This is highly desirable for power saving; however, complexity rises due to controller logic that turns off the clock of the unit/sub-unit and which is part of the verification environment. In the future, our efforts are to apply similar approach to block level clock gating.

References

[1] F. Emnett and M. Biegel, “Power Reduction Through RTL Clock Gating”,http://www.eng.auburn.edu/simvagrawal/COURSE/E6270Fall07/PROJECT/LUO/snug2000.pdf

[2] A. Raghunathan, N. Jha, and S. Dey, “High-Level Power Analysis and Optimization. Norwell”, MA, USA: Kluwer Academic Publishers, 1998.

[3] A. Kuehlmann and C. van Eijk, Combinational and Sequential Equivalence Checking, in Logic Synthesis and Verification. Kluwer Academic Publishers, 2004.

[4] H. Savoj, D. Berthelot, A. Mishchenko and R. Brayton, Sequential Equivalence Checking for Clock-Gated Circuits, in IWLS 2012.

[5] S. Ahuja, and S. Shukla. "MCBCG: Model checking based sequential clock-gating." *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*. IEEE, 2009.

[6] J. Brandt, et al. "The model checking view to clock gating and operand isolation." *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*. IEEE, 2010.

[7] B. Bentley, Bob. "Validating the Intel (R) Pentium (R) 4 microprocessor." *Design Automation Conference, 2001. Proceedings*. IEEE, 2001.

[8] J. Baumgartner, et al. "Scalable sequential equivalence checking across arbitrary design transformations." *International Conference on Computer Design,* ICCD 2006.

[9] M. Mneimneh, and K. Sakallah, "Principles of sequential-equivalence verification." *IEEE Design & Test of Computers,* 2005.

[10] C. Manovit, S. Narayanan and S. Subramanian, “Design and Verification Challenges of Observability Don’t Care (ODC)-based Clock Gating”, Poster Session, Design Automation Conference, 2011.

[11] Jasper Design Automation, http://jasper-da.com/

[12] Nvidia Tegra, http://www.nvidia.com/object/tegra.html