Clock Gater Verification Using Formal Property Checking

Alfonso Urzua

Advanced Micro Devices

Boxborough, MA, US

ABSTRACT

Clock gating is a common design technique for reducing dynamic power dissipation, by temporarily disabling the clocks of certain flops, while preserving the functional intent. Clock gater verification is oftentimes addressed via assertions within a simulation environment. Formal verification offers an exhaustive way to verify the design intent, becoming a powerful complement to simulation. In this paper, we propose a clock gater verification solution that uses formal property checking. First, we use VC Formal property checker by running unconstrained proofs on existing clock gater assertions, starting at the lower levels of the design hierarchy. The unproven assertions are re-run at higher levels of the design hierarchy, further constraining the formal proofs. Using formal property checking allowed us to exhaustively verify an average of 80% of the assertions within 3 hours and without constraints.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Many of today's portable electronic devices depend on low power IC designs in order to extend their battery life. Power dissipation can generally be divided into two: static and dynamic. Static power dissipation, also called leakage power, relates to power consumed while signals are not transitioning. Dynamic power dissipation relates to power consumed while a circuit is in operation with signal activity that causes capacitances to charge and discharge.

Clock gating is a common design technique for reducing dynamic power dissipation by temporarily disabling the clocks of certain flops, while preserving the design functional intent. The verification of clock gaters is oftentimes addressed indirectly via functional checks and assertions, attempting to catch potential clock gater design bugs. Following this approach has the drawback of a larger debug effort. For instance, clock gater bugs could be observed downstream from the root cause, depending on where the assertions catch the unexpected behavior, making debug less efficient. The other disadvantage is the fact that clock gater bugs could go undetected, since stimulus created by random testing does not guarantee exhaustive verification and directed testing is not practical. Ideally, each clock gater should have an assertion that checks for its intent, allowing for potential bugs to be more efficiently debugged and making sure we have an explicit verification target.

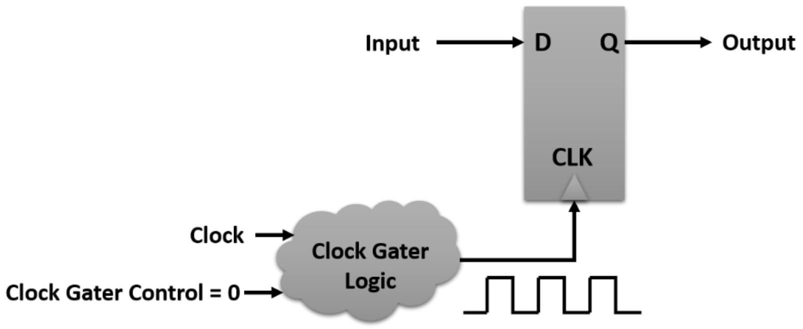
Formal verification offers an exhaustive way to verify the design intent and can be a practical complement to simulation. In this paper, we propose a clock gater verification solution using formal property checking. First, we use VC Formal property checker by running unconstrained proofs on existing clock gater assertions, starting at the lower levels of the design hierarchy. The unproven assertions are re-run at a higher level of the design hierarchy, where additional RTL can constrain the proof by reducing illegal state space. This clock gater verification approach allowed us to get proofs for an average of 80% of the assertions and without constraints. Most of the assertions converged within 3 hours.

2. Overview of Clock Gater Design

Clock gaters are used to limit the dynamic power dissipation by disabling clocks over selected flops, while preserving the design functional intent. Clock gaters are usually implemented by the RTL team, who follow certain local power requirements as part of an overall power specification.

In general, there are two types of clock gaters: power and functional. Power clock gaters are those that gate the clock when the input to a flop is constant, while letting the clock through, when the input changes. On the other hand, functional clock gaters are those that will gate the input to a flop when we want to prevent the downstream logic from consuming any new input values. In other words, power clock gaters do not change the functional behavior of the design, while functional clock gaters do.

Figure 1 presents a block diagram of a flop with its corresponding clock gater logic. The clock gater control signal is in charge of enabling or disabling the clock gater logic. When the clock gater control signal is asserted, as shown in Figure 1, the clock input to the flop is gated. On the other hand, when the signal is deasserted, the clock input to the flop becomes a free-running clock.



This can be observed in

Figure 2.

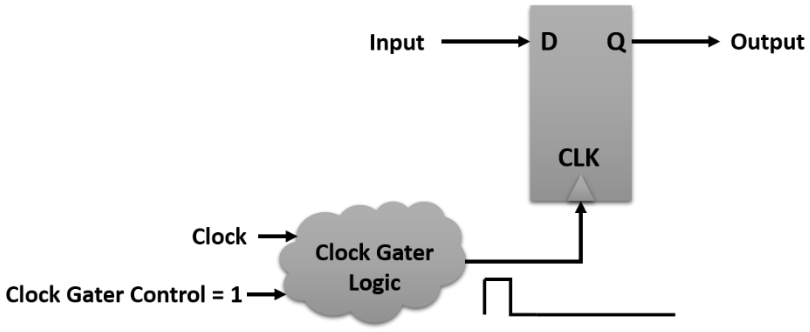


Figure 1. Flop with gated clock

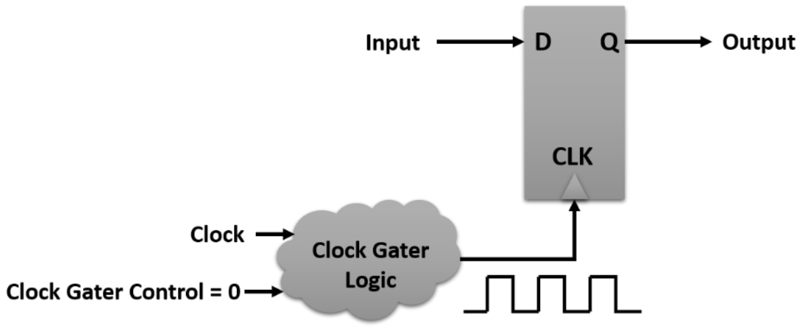


Figure 2. Flop with free-running clock

3. Challenges of Clock Gater Verification

The goal of verification is to qualify the implementation of a design against its corresponding design intent. The implementation of a design is written in RTL, while the design intent is usually expressed in a written specification. The verification and design teams plan together to ensure the design implementation follows the specified intent. This plan includes, among other things, the development of stimulus as well as checking infrastructure, the latter in the form of assertions and higher level checks.

The verification of clock gaters can be seen from various perspectives, each presenting a different challenge. First, we need to make sure the local power requirements are met. Second, we need to make sure the design functional intent always holds true in the case of functional clock gaters, and the overall functional intent is preserved in the case of power clock gaters. The remainder of this paper will only refer to the verification of power clock gaters. Verifying local power requirements or functional clock gater intent is beyond the scope of this paper.

Clock gater verification is seldom considered as part of the verification plan. Instead, it is addressed via existing assertions and higher level checkers combined with directed and/or random stimulus and coverage. Due to the typically large number of clock gaters in the design, developing directed tests for each of them would be impractical; hence we heavily rely on random stimulus.

With random stimulus we improve controllability to target clock gaters. However, we still have an observability issue, where some areas of the design could be exposed to non-observable bugs. For example, a bug can remain hidden if it can only be detected when incorrect behavior is propagated to an observable point, where we don't have an assertion in place.

The observability issue can be addressed by adding assertions that directly target clock gater functionality. However, the number of clock gaters existing in recent designs is very large, making it almost impossible to be addressed independently like other aspects of the design functionality. Instead, we need an automated solution that will create assertions for each clock gater and will verify those assertions always hold true.

This paper does not discuss the automatic generation of clock gater assertions. Instead, we assume that assertions are available for each clock gated flop, and their naming follows an established guideline. Having a naming guideline, will allow us to identify a large number of assertions using regular expressions.

An instance of an assertion that checks for correct clock gater functionality looks like this:

```
// Instance of a clock gated flop
FFG Flop(.C(CLK),.R(RESET),.D(IN),.Q(OUT),.CLKEN(CLKEN));


// Clock gater assertion
asrt_cg: assert property(IN != OUT |-> CLKEN == 1'b1);
```

Figure 3. Clock gater assertion example

The above implication will make sure that any change on the flop input will be propagated to its output by properly enabling the clock.

3.1 Speeding up the Verification Effort with Formal

Formal verification is a systematic process of ensuring, through exhaustive algorithmic techniques, that a design implementation satisfies the requirements of its specification. All possible executions of the design are mathematically analyzed without the need to develop simulation input stimulus or tests [1]. On the other hand, formal verification that involves a large state space will not always be able to complete the proofs within a reasonable time, making simulation the only viable option. Formal verification can still become a practical complement to simulation when the target functionality is well defined.

This paper presents the use of *model checking*, also known as *property checking*, in the verification of power clock gaters. *Model checking* is a formal technique that can be applied to the verification of sequential logic designs. This technique needs some sort of *specification* in addition to the design *implementation*. The *specification* is presented in the form of properties made of temporal logic expressions. *Model checking* tools provide algorithms to traverse the properties as well as the design implementation, ensuring that the intent specified through those properties always holds true; otherwise, counter-example traces are generated [2].

As shown in Figure 4, *model checking* can be applied by embedding assertions, similar to the one in Figure 3. Each of these assertions represents the reference specification for each clock gater. A model checking tool makes sure that the intent specified through the assertions always holds true; otherwise, a counter-example trace is generated, showing how specific input conditions violate the assertion intent.

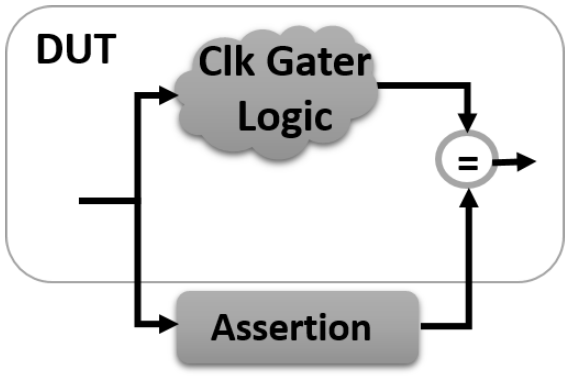


Figure 4. Model checking

4. Approach

Our methodology to verify clock gaters is based on *model checking*, making use of the Synopsys VC Formal property checker tool.

In order to achieve formal proofs within a reasonable time, we start by dividing the device under test (DUT) into sub-modules. For each sub-module, we run unconstrained proofs with VC Formal property checker on each the clock gater assertion. By running without input constraints, we can achieve proofs with an exhaustive set of inputs values. Also, by running the proofs at the sub-module level, we can potentially achieve shorter proof times.

After running unconstrained proofs on every sub-module, we might end up with a group of unproven assertions and their corresponding counter-example traces. These counter-example traces could indicate false failures, due to the lack of input constraints, or they could indeed point to a RTL bug. However, instead of debugging those counter-example traces, the same unconstrained proofs are re-run within a larger design. The neighboring sub-modules, now present on the larger design, work as new input constraints for the proofs.

The remaining unproven assertions can be further refined, in order to understand the nature of their failures. This remaining effort would require a deeper understanding of the underlying design implementation.

5. Clock Gater Formal App

In order to automate the various steps to run the VC Formal property checker, we create a Ruby script to orchestrate the top level flow, calling various Tcl scripts as described in the following diagram:

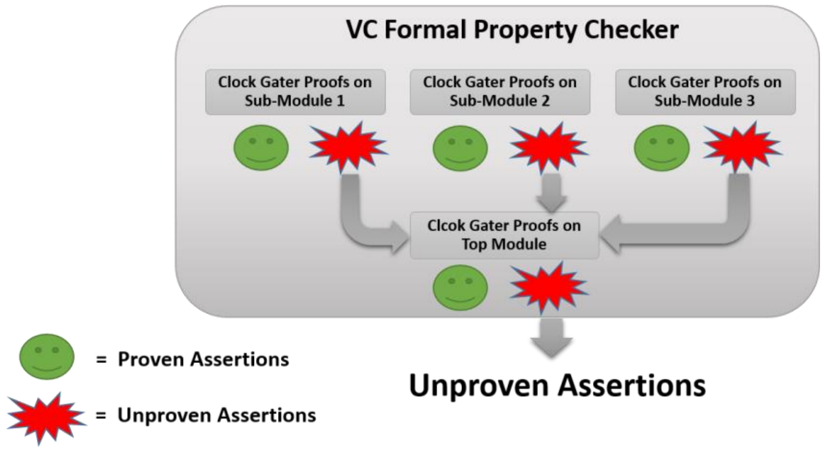

VC Formal Property Checker
Clock Gater Proofs on Sub-Module 1
Clock Gater Proofs on Sub-Module 2
Clock Gater Proofs on Sub-Module 3
Clcok Gater Proofs on Top Module
= Proven Assertions
= Unproven Assertions
Unproven Assertions

Figure 5. High level flow for clock gater verification

First, threads are spawn, each executing a VC Formal property checker Tcl script. Every script performs unconstrained proofs on each clock gater assertion, within a sub-module. A list of unproven assertions is generated. The list is fed to a subsequent Tcl script in charge of re-running the proofs at the top DUT level. The intent of running at the top level, is to increase the number of exhaustively verified clock gaters.

5.1 Scripting the Property Checking Flow

In general, all Tcl scripts running the VC Formal property checker contain the following abbreviated steps:

Run as a property checker with fml_mode_on set to true.

```
# Initial Setup and Configuration
set_app_var fml_mode_on true
```

Specify the top level module and read the corresponding design.

```
# Read in the design
set top sub_module_1
catch {[read_file -top $top -sva -format sverilog -vcs … }
```

Define clock and reset.

```
# Reset
create_reset reset -high
# Clock definition
```

```
create_clock CLK -initial 0 -period 10
```

Simulate until signal activity settles.

```
# Run the simulation
sim_run -stable
```

Run proofs and generate a report of unproven assertions to be used for further processing.

```
# Run proof
check_fv -property [get_props "*non_functional_clk_gating_check*" -usage assert]
# Report passing and unproven assertions and create output file.
report_fv -list > unproven_assertions_submodule_1.txt
```

6. Closing the Unproven Assertion Gap with VC SEQ App

One area for further exploration is the use of *sequential equivalence checking* as a way to close the unproven assertion gap. *Sequential equivalence checking* compares two designs with the same set of primary inputs and outputs and the same sequential depth. One design is called the *specification* or reference, while the second is called *implementation.* With *sequential equivalence checking*, we ensure that *specification* and *implementation* designs produce identical outputs given identical inputs on a cycle-by-cycle basis. The *specification* or reference is assumed to be correct. When the *implementation* behavior deviates from the *specification*, counter-example traces are generated [1].

In the case of our clock gater verification methodology, *sequential equivalence checking* would use a *specification* design with clock gaters disabled, essentially providing free-running clocks to the corresponding flops. As shown in Figure 6, the *specification* is compared against the *implementation*, with fully functioning clock gaters. A formal analysis allows us to exhaustively compare both designs, making sure they are sequentially equivalent. A counter example trace gets generated when an input condition exposes a different behavior between *implementation* and *specification*, producing different outputs.

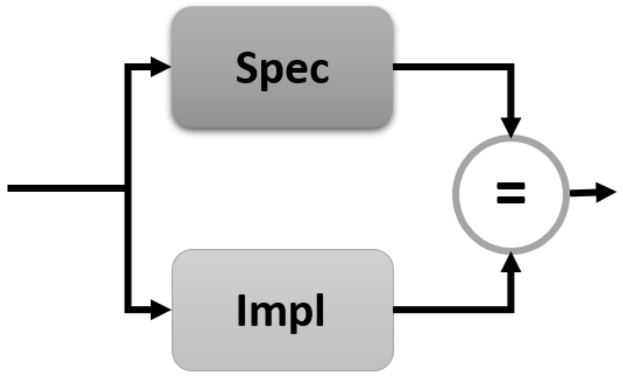


Figure 6. Sequential equivalence checking

Synopsys offers a *sequential equivalence checking* tool called VC SEQ App. **Error! Reference source not found.** shows a flow similar to the one on Figure 5. Threads are spawn, each executing a VC Formal property checker Tcl script. Then, unconstrained proofs are performed on each clock gater assertion, at different levels of the design hierarchy. The list of unproven assertions is fed to a VC SEQ App Tcl script with the intent to increase the number of exhaustively verified clock gaters. We believe that using the VC SEQ App as shown on **Error! Reference source not found.**, could potentially enhance the clock gater verification methodology described in this paper.

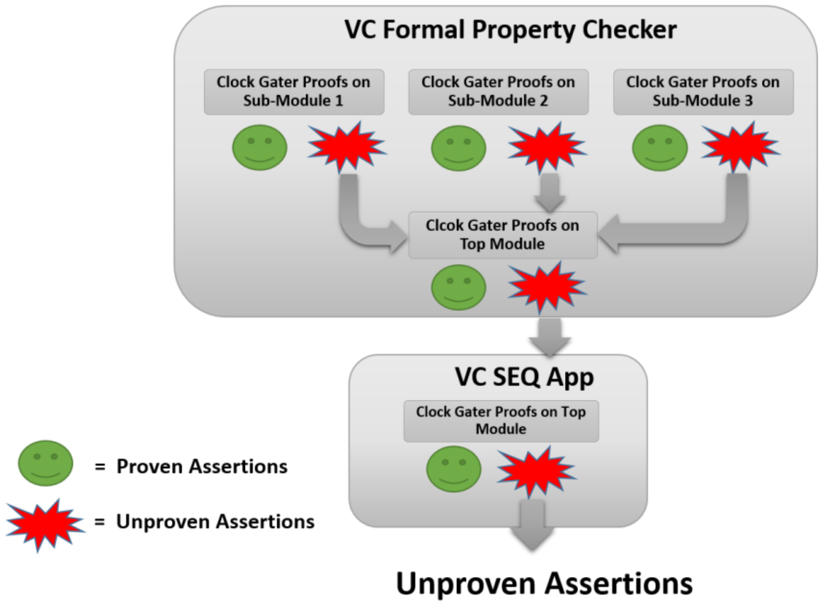

VC Formal Property Checker
Clock Gater Proofs on Sub-Module 1
Clock Gater Proofs on Sub-Module 2
Clock Gater Proofs on Sub-Module 3
Clcok Gater Proofs on Top Module
VC SEQ App
Clock Gater Proofs on Top Module
= Proven Assertions
= Unproven Assertions
Unproven Assertions

Figure 7. Clock gater verification flow including the VC SEQ App

7. Results

We used the clock gater verification flow described above, to successfully prove an average of 80% of power clock gaters within a microprocessor module. VC Formal formal property tool ran an unconstrained proof on each of the clock gater assertions at the sub-module level. Unproven assertions were re-run unconstrained at the top module level. The results are summarized in the following table.

Table 1. Results running VC Formal property checker

Module	Clock gaters	Property Checker Proofs	Proof time
Sub-module 1	1.2K	60%	2.7hr
Sub-module 2	1.3K	70%	2.5hr
Sub-module 3	1K	99%	3hr
Top module	0.8K	20%	4hr

The above results were achieved by running unconstrained proofs and essentially no debug effort other than getting the Ruby and Tcl scripts setup and run. Common script functionality was refactored for easier re-use in other areas of the design. The end result was a relatively large list of exhaustively verified clock gaters, leaving a smaller list that needs to be further refined in order to understand the nature of the failures.

As can be observed from Table 1, most of the proofs were achieved at the sub-module level. For the remaining proofs that were run at the top module, only 20% were proven, while the remaining were either falsified or inconclusive.

8. Conclusions and Future Work

In this paper, we propose a methodology to verify power clock gaters by leveraging the property checker available through the VC Formal tool. The main pre-requisite for this methodology is to have an assertion specifying the intent of each clock gater and a uniform naming convention for the corresponding flops and assertions. These pre-requisites will allow for easier querying by the VC Formal property checker Tcl scripts.

The flow is fully automated via Ruby and Tcl scripts, and applied to a microprocessor module. Once the flow was implemented, it was a matter of re-running the tool and waiting for the results. The results were presented as a list of flops and their clock gaters which needed further analysis.

This methodology resulted in an average of 80% of power clock gaters verified, without the risk of over-constraining and with a low engineering effort. The remaining unproven assertions can be further refined. For instance, we can implement a subset of input constraints that will reduce the illegal input state and corresponding false failures. Alternatively, we can increase the maximum proof time or experiment with other, potentially more optimum, tool parameters. Starting with unconstrained proofs and later implementing the necessary input constraints is, in general, a good practice that will reduce the risk of over-constraining a proof and missing potential bugs.

As discussed earlier, another area for future work is the use of *sequential equivalence checking* as a way to close the unproven assertions gap. Using the VC SEQ App, as shown in Figure 7, could potentially help reduce the number of assertions left unproven by VC Formal property checker.

9. References

[1] "Applied Formal Verification" by Douglas L. Perry and Harry D. Foster. April, 2005

[2] Synopsys, "VC Formal Verification User Guide", 2016.06-SP2