

Functional Verification of Power Gated Designs by Compositional Reasoning

Cindy Eisner, Amir Nahir, and Karen Yorav

IBM Haifa Research Laboratory

Abstract. Power gating is a technique for low power design in which whole sections of the chip are powered off when they are not needed, and powered back on when they are. Functional correctness of power gating is usually checked as part of system-level verification, where the most widely used technique is simulation using pseudo-random stimuli. We propose instead to perform a sequential equivalence check between the power gated design and a version of itself in which power gating is disabled. We take a compositional approach that looks for partial equivalence of each unit under a suitable set of assumptions, guaranteed by the neighboring units. We make use of so-called circular reasoning rules to compose the partial equivalences proved on the individual units back into total equivalence on the whole chip.

1 Introduction

Power consumption is an important consideration in modern chip design. A portable device should use as little energy as possible, in order to extend battery life, and a non-portable device should use as little energy as possible in order to save electricity costs – the expense of powering a server farm can easily outstrip the cost of the servers themselves. Thus design teams go to great lengths to design systems that use as little power as possible.

Many power saving techniques are purely electrical, such as changing the type of transistor used. Other techniques, while electrical, can be understood at the logical level. Among them are multiple power domains, in which different areas of the chip get different voltages; dynamic frequency/voltage scaling, in which the frequency and/or voltage is changed while the chip is working; clock gating, in which the clock input of a memory element is prevented from “ticking” if a tick would be redundant; and power gating, in which whole sections of the chip are powered off when not needed, and powered back on when they are. In addition to the physical design challenges, i.e., getting the electronics right, each of these techniques adds a new dimension to the functional verification problem.

In this paper we focus on power gating [2]. Power gating is one of the most powerful techniques for saving power, as it reduces both dynamic power (the power used when the design transitions from state to state) as well as static, or leakage, power (the power used by the design when it is in a steady state). Functional correctness of power gating is usually checked as part of system-level

verification, where the most widely used technique is simulation with pseudo-random stimuli. However, because power gating is such a complicated technique, this approach can be a real barrier to its use – either because there is not enough confidence that the current methods can verify correctness, or because the effort required to do so is not economically feasible. Therefore a practical methodology to formally verify the correctness of power gating fulfills a real need.

We present a methodology that addresses functional verification of a design in the presence of power gating, in which we break the verification task into two parts. In the first, correct functionality of the design is checked when power gating is disabled, using the usual techniques (formal and/or dynamic, as the case may be). The second part is a sequential equivalence check between a version of the design with power gating enabled and one with it disabled.

We focus on the second part, which due to the size problem is not trivial. At the level of a single unit, we do not necessarily expect complete sequential equivalence. Rather, we expect that the cases in which the two versions of the unit differ are don't cares for the neighboring units. Thus we require only partial equivalence of each unit, and we make use of so-called circular reasoning rules to allow us to compose the units and the partial equivalences proved on them into total equivalence on the whole chip.

At a very high level, our method can be understood as follows: identify the conditions under which the interface between a power gated unit and its neighbors is “active”, and require that the power gated unit preserve functionality only then. Separately, prove that the neighbors are not affected by a difference in behavior when the interface is not active. We note that implicit in our method is the assumption that power gating does not change the external behavior of the chip, but this assumption does not hold in some cases – for example, pipelines that stall when the needed unit is not available. Overcoming this limitation is future work.

In general, compositional reasoning is difficult because of the manual effort involved in coming up with appropriate assumptions, and this has severely limited its use in industrial settings. In our case, the specification is not an arbitrary formula but rather sequential equivalence between outputs of the (composed) design. This simplifies the task of figuring out what the assumptions should be. We require only that the user supply us with an *observer*, a piece of code that observes the chip but does not influence its behavior. The observations that it needs to make are of a specific and limited nature, thus manual coding of an observer is not an unduly difficult task to expect of the user.

Related Work In this paper we do not invent a new compositional reasoning rule, but rather make use of the one presented in [4]. Our contribution to the compositional reasoning community is the identification of a class of real-life problems for which compositional reasoning is relatively easy, thus barriers to industrial adoption of it are minimal.

As defined in [5], sequential equivalence solves the problem of deciding whether two arbitrary designs are equivalent without knowledge of their initial states or

a reset sequence that will bring each of them to a single known (initial) state. Other works (e.g., [7]) use the term sequential equivalence to describe a more restricted problem, in which some knowledge about the initial state exists. We use *alignability equivalence* to describe the more general problem examined in [5], and *sequential equivalence* for the more restricted problem as described in [7].

The problem of *compositional* alignability equivalence is examined in [3]. In contrast, our concern is with sequential equivalence. Like us, [3] look for a partial equivalence, but their partial equivalence includes assumptions about the environment of the full chip, which they do not discharge. We also allow such assumptions, if necessary, but the main source of partiality in our work is knowledge about how a particular power gated design is intended to work. Furthermore, we discharge all assumptions that are derived from such knowledge.

Sequential equivalence checking is widely used to verify the correctness of local changes introduced by techniques such as retiming and clock gating [1]. It is used to verify that the change preserved the functionality of the original design. We propose something similar for power gating, however our equivalence check is complicated by the fact that the change causes non-local side effects.

To the best of our knowledge, there is no prior literature on functional verification in the presence of power gating. Technical articles available online discuss various limited aspects of the problem, for instance, that the power-up and power-down sequences work as expected, that the correct parts of the state are retained by the power retention logic, or that the power management unit itself functions correctly. However, no prior work has considered the problem of functional correctness of the chip as a whole in the presence of power gating.

2 Power Gating

Recall that in power gating, a unit or a large part of it is powered off when it is not needed, and powered back on when it is. For instance, the power to a floating point unit may be shut off when there are no floating point instructions to be processed, and turned back on when there is a need. The memory elements of a power gated unit lose their memory when powered off. Thus, power gating requires either that the power gated unit be *memoryless*, i.e., such that when turned back on it is able to function with no memory of the past, or else that there is some circuitry that takes care of *state retention* – remembering enough of the state to allow the unit to function correctly when turned back on.

When a logic gate is powered off, its output should not be read by a logic gate that is powered on, for physical design reasons that are beyond the scope of this paper. This results in two complications. First, the inputs of a powered on unit must be insulated from the outputs of a powered off unit. This is achieved by inserting an isolation device, which we will term a *fence*, between the outputs of a powered-off unit and the inputs of a powered-on unit. The physical implementation of a fence is beyond the scope of this paper. Logically, when the fence is “up” it drives a constant value and when it is down, it simply passes on the value of the output that it is fencing. A fence must be put up before a power

gated unit is powered off, and must be left up until the power gated unit has been powered on and sufficient time (some number of clock cycles – the exact number is design dependent) has passed to ensure that all outputs are reliable.

The second complication is that state retention is not simply a matter of powering off some memory elements while leaving others powered on, because the powered on memory elements must be fenced off from the powered off elements. Therefore, state retention is normally accomplished by copying part of the state to dedicated memory elements before powering off a unit, and then copying them back after the unit has been repowered (and before it is used).

The process of raising the fences and copying a partial state to the dedicated state retention before powering off is part of the *power down sequence*, while the process of powering up the design, copying the retained state back into the functional memory elements, and then lowering the fences is part of the *power up sequence*. Triggering the power down and power up sequences is the responsibility of the *power management unit*.

Conceptually, there are two parts to verifying the correct implementation of power gating. One part is syntactic, or structural: checking that fences exist at the right places, and checking that if the power is off, the fence is up and stays up for a sufficient number of clock cycles to ensure the (electrical) reliability of the gate. This is easily checked, and is orthogonal to the matter we examine in this paper. The other part is semantic, or functional: checking whether the implementation of power gating preserves the behavior of the chip. The latter is the problem that we are concerned with in this paper.

3 Methodology

We first show the setup of our methodology and the inputs required from the user, then show how we use these inputs to prove sequential equivalence between a power gated design and a version of itself in which power gating is disabled.

Preliminaries Given a set Σ of signals, a *path* is a function $\pi : \Sigma \times \mathbb{N} \rightarrow \{0, 1\}$ that assigns each signal a Boolean value at each time point. We can constrain the behavior of signals on a path by associating a predicate with each signal. Such a predicate can be either an *input predicate*, which imposes no constraint at all, a *gate predicate* such that the value of signal $\sigma \in \Sigma$ at time t is a function of the values of other signals at time t (including the constant functions 0 and 1), or a *latch predicate* such that the value of signal $\sigma \in \Sigma$ at time t is a function of the values of other signals at time $t - 1$. Formally,

Definition 1 (Input, gate and latch predicates). Let σ_i for $1 \leq i \leq n$ be signals, and denote the value of signal σ_i at time t by $\sigma_i(t)$. Let f be an n -ary function and let *init_values* be a subset of $\{0, 1\}$. Then:

- $p(t) \in \{0, 1\}$ *is an input predicate.*
- $p(t) \in f(\sigma_1(t), \dots, \sigma_n(t))$ *is a gate predicate*
- $p(t) \in \begin{cases} \textit{init_values} & : t = 0 \\ f(\sigma_1(t-1), \dots, \sigma_n(t-1)) & : t > 0 \end{cases}$ *is a latch predicate*

Let Σ be the set of signals in a non-power gated design D , and describe each signal $\sigma \in \Sigma$ as an input, gate, or latch predicate p_σ in the obvious way. The set D of predicates $\{p_\sigma | \sigma \in \Sigma\}$ describes the design D . Note that a multiply clocked design can be described as easily as a singly clocked design (as is usual when building a model of a design for model checking or equivalence checking).

In a power gated design, each gate and latch has an additional input pin that represents the state of the power supply – either on or off. Thus to describe a power gated design, we modify the “obvious way” of the preceding paragraph as follows. An input or gate predicate is not affected by whether the power is on or off, whereas a latch predicate has its “normal” value when the power is on, and a non-deterministic value otherwise. This gives us that when the power comes back on, the entire unit is in some arbitrary state.

Although this does not completely reflect reality (to do so, we would have to modify gate predicates as well), it is enough for our purposes assuming that each output of the power gated domain is fenced and that the fence is guaranteed to be up when the gate or latch driving the output is powered off. As stated in Section 2, this is orthogonal to the issue we examine in this paper and thus we can assume that it will be checked by other means.

Setup of our methodology We partition the design as shown in Figure 1. Here and in the rest of this paper we use the word “unit” to refer to a piece of the design of any size that has a well-defined interface, and may or may not be composed of other units or blocks of code. G consists of one or more power gated units G_i plus the power management unit PM , which controls the powering on and off of each G_i (when power is turned off to a G_i , not all other G_j ’s are necessarily powered off). We require the existence of one specific signal in G : a signal pg_enable that is read only by PM and whose role is to enable the power gating. For simplicity we assume that pg_enable is a constant within G .

U consists of a number of U_i ’s. These are every non-power gated unit in the design that directly interfaces with G . Any part of the design that interfaces with U but is not contained in G (call it the environment of U) exists intuitively to the right of U in the figure. Usually it will be completely abstract – that is, we will make no assumptions on it at all, although we may relax this slightly if necessary. Finally, the outputs of U to the left (in the direction of G) and those to the right (in the direction of U ’s environment) are not necessarily disjoint.

Note that G has no interface other than with U . That is, if G receives inputs directly from the chip interface or drives outputs directly to it, we assume for simplicity that they are buffered (with possibly zero delay) through U .

Now, if we can show that the design $G||U$ is equivalent to the design $G'||U'$, where the only difference between the primed and unprimed versions is that

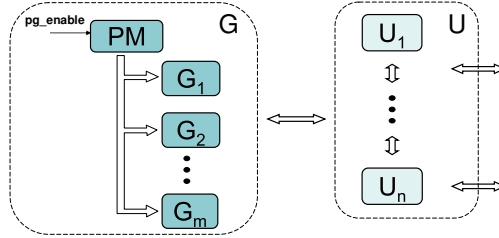


Fig. 1. Partitioning of the design for power gating verification

$pg_enable = 1$ in $G\|U$ whereas $pg_enable = 0$ in $G'\|U'$, then we will have shown that power gating does not affect the functionality of the design as a whole. Note that this strategy as stated is not complete – it is possible to create a correct design in which the effects of the power gating penetrate to the interface of U with its environment, in which case our method will not be able to prove that the design is correct. However, in such a case we can move part of U 's environment into U . In the worst case, $G\|U$ will include the entire chip (recall that we assume that power gating does not change the external behavior of the chip).

Recall that our goal is to show that $G\|U$ is equivalent to $G'\|U'$, and due to size problems we would like to do it compositionally – that is, compare each G_i with G'_i and each U_i with U'_i . For simplicity of the explication, we first show how to break the problem into comparing G with G' and U with U' , and only afterwards how to break the problem down further.

Obviously we do not have complete equivalence – when G is powered off, its outputs are not necessarily equivalent to those of G' . And although U and U' will surely behave the same if they receive the same inputs (because there is no difference between them), in our scheme U will get its inputs from G and U' from G' , thus showing equivalence between them is not trivial. Furthermore, when comparing G with G' we have to be careful: if the inputs of the power management unit PM “misbehave”, it might shut off some G_i at an inappropriate time – for instance, when it is in the middle of processing a transaction. Thus we may need some minimal assumption over the inputs that influence PM , and we need to be able to guarantee this assumption.

We therefore ask the user to supply a simple *observer*, a piece of code that monitors the interface between G and U and outputs flags that indicate properties of the interface. Each flag is used as an assumption by one of $G\|G'$ or $U\|U'$ and is guaranteed by the other, and the apparent circularity is broken by induction over time. Thus the setup of our methodology is as shown in Figure 2, where the flags are signals partitioned into sets $GoodU$, $GoodG$ and V , as follows:

- **GoodU** Each flag in this set has the value 1 as long as some assumption about the behavior of U is preserved. These assumptions do not specify the exact correct behavior of U on this interface, only the minimal needed

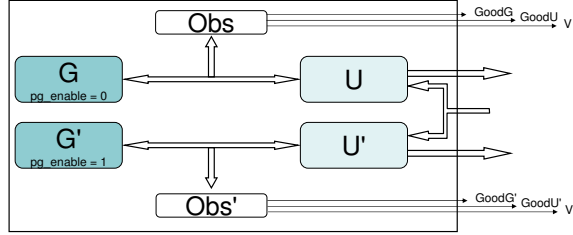


Fig. 2. Setup for equivalence checking

restrictions. As soon as a violation of these assumptions is detected the flag goes to 0 and stays so forever.

- **GoodG** This set is similar to $goodU$, but over G .
- **V** Conceptually, this set contains a single flag v , which is a “valid” signal that indicates whether the interface between G and U is active. When $v = 1$ we expect the outputs of G and G' to be equivalent, and when $v = 0$ we do not. For example, v could be $ready \wedge transmitting$, where $ready$ is an output of U signifying that U is ready to receive data and $transmitting$ is an output of G signifying that G has data ready on the bus. In fact, V is not a single flag but a set of flags, because each U_i may have its own interface with each G_j , and even across a single interface not all signals necessarily follow the same protocol.

Note that the observer that generates the flags is coded manually. At first glance this may seem to be a very complicated task, one that perhaps should be automated. However, an observer is a direct result of the interface protocol between G and U , thus should be very easy for a human to code, and indeed all the designers we have spoken with understand intuitively what the observer for their design should look like. On the other hand, because it requires an understanding of the design intent, coding an observer would be extremely difficult to automate. We note that if the user makes incorrect assumptions that are either too strong or too weak, then some proof obligation (that will be presented below) will fail with a counterexample. Thus, wrong assumptions in the coding of $GoodU$, $GoodG$ or V will never lead to an incorrect claim of equivalence.

Regarding the sets $GoodU$ and $GoodG$, we expect the typical user to start off with empty sets and gradually add constraints refining them as needed. Examples of such sets are presented in Section 4. In the general case of assume-guarantee reasoning for functional correctness, this refinement process is difficult because it requires some kind of semantic understanding of how the design is intended to work. In our simplified setting, these conditions will typically be simple translations from the English specification of the interface, such as “there are no requests during reset”. Moreover, we need much weaker assumptions than those necessary to check functional correctness, because we don’t care if the designs misbehave as long as the two copies (mis)behave in exactly the same way.

Note that it is possible to code a correct design in which the interface between G and U is always active (despite the fact that G can be powered down), and that this does not break our methodology. In such a case the fences and the state retention logic of G will be such that the valid signal has the constant value 1, and the equivalence between U and U' will be trivial.

Proving sequential equivalence We base our approach on the compositional reasoning rule presented by McMillan in [4], and borrow our notation from there. Following [4], we abuse notation by using Q to denote the conjunction of all predicates in the set Q .

Let P be a set of predicates describing the design and let S be a set of predicates defining the specification. For each predicate $s \in S$ let $\mathcal{E}_s \subseteq P \cup S$ be the environment of s . Intuitively, this is the set of predicates needed in order to show that s holds. We assume a well-founded order \prec on S that defines for each predicate s which other predicates will be assumed up to time i when proving s at time i (this is Z_s), and which will be assumed only up to time $i - 1$ (this is \bar{Z}_s , the complement of Z_s). Then by [4] we can use Theorem 1 below.

Theorem 1 (McMillan [4]). *Let P and S be sets of predicates, for each $s \in S$ let $\mathcal{E}_s \subseteq P \cup S$, and let \prec be a well-founded order on S . Let $Z_s = P \cup \{s' \in S : s' \prec s\}$, and for a predicate p let $p \uparrow^\tau$ stand for $\bigwedge_{t \leq \tau} p(t)$. Then, if for all $s \in S$,*

$$(\mathcal{E}_s \cap Z_s) \uparrow^\tau \wedge (\mathcal{E}_s \cap \bar{Z}_s) \uparrow^{\tau-1} \Rightarrow s(\tau)$$

is valid, then $(\forall t. P(t)) \Rightarrow \forall t. S(t)$ is valid.

Our goal is to use Theorem 1 to prove sequential equivalence between $G \parallel U$ and $G' \parallel U'$. Since we have assumed that all outputs of $G \parallel U$ are outputs of U it is enough to show that the predicate

- $EqU(t) \stackrel{\text{def}}{=} \{o(t) \leftrightarrow o'(t) : o \text{ is an output of } U\}$

holds at all times t . We will need the following auxiliary sets of predicates:

- $P_{GoodU}(t) \stackrel{\text{def}}{=} \{s(t) = 1 \mid s \in GoodU\}$
- $P_{GoodG}(t) \stackrel{\text{def}}{=} \{s(t) = 1 \mid s \in GoodG\}$
- $P_V(t) \stackrel{\text{def}}{=} \{v(t) \leftrightarrow v'(t) \mid v \in V\}$
- $EqG(t) \stackrel{\text{def}}{=} \{v_o(t) \rightarrow (o(t) \leftrightarrow o'(t)) : o \text{ is an output of } G \text{ and } v_o \in V \text{ is its associated valid bit}\}$

Let G, G', U, U', Ob and Ob' be the sets of predicates describing the respective designs of Figure 2. Let $\hat{G} = G \cup G' \cup Ob \cup Ob'$ and $\hat{U} = U \cup U' \cup Ob \cup Ob'$. Let $P = \hat{G} \cup \hat{U}$, and $S = P_V \cup P_{GoodU} \cup P_{GoodG} \cup EqU \cup EqG$.

To begin with, let's assume that the relation \prec is empty, thus for every element s of S , we have $Z_s = P$ and $\bar{Z}_s = S$. Therefore proving the following

$$\hat{G} \uparrow^\tau \wedge (EqU \cup P_{GoodU}) \uparrow^{\tau-1} \Rightarrow (EqG \cup P_{GoodG} \cup P_V)(\tau) \quad (1)$$

$$\hat{U} \uparrow^\tau \wedge (EqG \cup P_{GoodG} \cup P_V) \uparrow^{\tau-1} \Rightarrow (EqU \cup P_{GoodU})(\tau) \quad (2)$$

will allow us to conclude that $(\forall t.P(t)) \Rightarrow \forall t.S(t)$, and in particular that $(\forall t.P(t)) \Rightarrow \forall t.EqU(t)$, which is our goal.

In practice there will usually be some combinational paths from inputs to outputs in one or more of G , U and Ob , in which case we will need stronger assumptions for some of the proof obligations. That is, we will need $s \uparrow^\tau$ as opposed to $s \uparrow^{\tau-1}$ for some element $s \in S$ used on the left-hand side of Obligation (1) or (2). Thus we will need to set an order, easily determined from the topology of the design, between the elements of S . As noted by McMillan in [4], such an order is guaranteed to exist when there are no combinatorial loops in the design. Since a combinatorial loop is a basic structural design error, we are guaranteed the existence of a well-founded order. Using the well-founded order, each of the Obligations (1) and (2) will be split into a number of proof obligations, one for each predicate in the conjunction on the right hand side. For example, let one such predicate be $s(t) = (v_o \rightarrow (o(t) \leftrightarrow o'(t))) \in EqG$, and let $A = \{s'(t) | s' \prec s \text{ and } s' \in EqU \cup P_{GoodU}\}$ and $B = (EqU \cup P_{GoodU}) \setminus A$. The corresponding proof obligation for s is then:

$$(\hat{G} \cup A) \uparrow^\tau \wedge B \uparrow^{\tau-1} \Rightarrow (v_o \rightarrow (o(\tau) \leftrightarrow o'(\tau))) \quad (3)$$

Conceptually, it has been convenient up till now to view G and U as monolithic units. However, in reality each will typically consist of a number of smaller units, as shown in Figure 1. Thus we would like to decompose the verification problem further by considering each G_i and U_i separately. For an output o of some U_i , we would like to use only U_i rather than all of U on the left hand side of its proof obligation. To do so, we must add the following predicates to S :

- $EqIntU(t) \stackrel{\text{def}}{=} \{s(t) \leftrightarrow s'(t) : s \text{ is an interface signal between } U_i \text{ and } U_j \text{ for some } i \neq j\}$

The situation for a single G_i is slightly more complicated: we must include the power management unit PM together with each G_i , and the predicates that we add for the outputs of G_i will be conditional, thus we might need to add some new valid signals. Denote the new valid signals by V_{new} . Then we add the following additional predicates to S :

- $P_{V_{new}}(t) \stackrel{\text{def}}{=} \{v(t) \leftrightarrow v'(t) | v \in V_{new}\}$
- $EqIntG(t) \stackrel{\text{def}}{=} \{v_s(t) \rightarrow (s(t) \leftrightarrow s'(t)) : \begin{array}{l} s \text{ is an interface signal between } G_i \text{ and } G_j \text{ for some } i \neq j \text{ and} \\ v_s \in \{V \cup V_{new}\} \text{ is its associated valid bit} \end{array}\}$

The order \prec is easily extended to the new predicates by a topological analysis of the design. For each output o of some G_i or U_i , we verify its proof obligation using \hat{G}_i or \hat{U}_i in place of \hat{G} or \hat{U} , where $\hat{G}_i = PM || G_i || G_i'$ and $\hat{U}_i = U_i || U_i'$.

Note that the theory supports multiply clocked designs as well as singly clocked ones. In the case of a singly clocked design, each time t is simply a tick of the clock. In the case of a multiply clocked design, each time t is a tick of the smallest granularity of time as seen by the verification tool (this is exactly the same as in model checking or equivalence checking of multiply clocked designs).

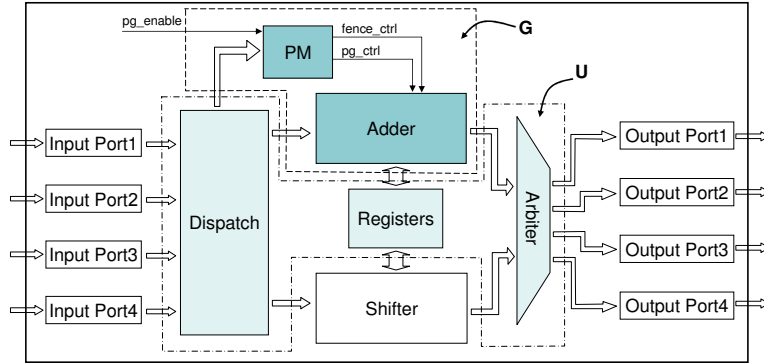


Fig. 3. The architecture of Calc3 enhanced with power gating

4 Case study on an execution unit

We applied our methodology to Calc3 [8], an RT-level implementation of a simple execution unit. Calc3 has 4 user ports through which commands are entered. It can process up to 4 commands from each port in parallel, supporting out-of-order execution of commands. The unit has 16 internal 32-bit registers and supports 4 types of commands: *load-store*, *branch*, *shift*, and *add-subtract* commands. The branch commands contain a condition on a register; if this condition is met then the next command on that port is skipped.

Although Calc3 is relatively large for formal verification, it is relatively small compared to a typical execution unit. Still, the implementation is far from trivial. For example, some parts of the design work on the rising edge of the clock and others on the falling edge. There is also some amount of speculative calculation in which operations are performed before it is known whether they will be needed, and then results are ignored if they are not. These optimizations add complexity to the design, and of course to the verification. We chose it because it is simple enough to serve as a first application of a new methodology but still contains enough complications and design optimizations to make it interesting.

Figure 3 gives an overview of the internal structure of Calc3. Commands are injected into the unit through the four input ports, and are held in the dispatch queue until they are sent to either the adder or the shifter, depending on their type. The results pass to an arbitrator, which distributes them to the four output ports. The adder is responsible for all add/subtract and branch commands, while the shifter executes shift and load/store commands.

For our case study we added power gating to the adder. The new power management unit *PM* keeps track of the types of commands that are in the dispatch queue by monitoring the inputs and outputs of the dispatch. Whenever there are no pending commands for the adder, *PM* initiates the power down sequence for it, which involves fencing the outputs and one clock cycle later powering down the adder. Whenever *PM* detects an incoming command destined

for the adder, it initiates the power up sequence for it, which consists of restoring power and one cycle later lowering the fences. When powered down, most of the adder unit is inactive, except for one block which must remain powered at all times (this is the part that remembers that a branch command was taken and outputs the appropriate indication towards the arbiter). The power gated version of Calc3 has 2200 state variables, 200 input bits and 144 output bits.

We divided Calc3 into the G side, containing the adder and PM , and the U side, containing the dispatch unit, the registers, and the arbiter, as shown in Figure 3. Note that the shifter does not directly interface with G and therefore is not part of U , but belongs to U 's environment, as described in Section 3. We also built an observer as described in Section 3, that outputs the required flags.

GoodU This set contains several simple properties, as follows: interface signals must change value only on the appropriate clock edge and no commands enter during reset. In addition, there is a flag indicating when the outputs of the dispatch that go to PM and those that go to the adder are inconsistent (i.e. the adder is given a command to execute and the PM was not informed that this command ever entered the dispatch).

GoodG The equivalence on the U side is much easier, and does not require any assumptions on the behavior of G . This set contains only a flag indicating that there is no activity during reset.

V We have two valid signals. The first is a Boolean combination of the outputs of the adder; it is 1 iff the adder is outputting a command towards the arbiter. The second is associated with outputs that signal whether or not the next command should be skipped, and it is a constant '1'.

\hat{G} , \hat{U} and S are built as per the methodology described in the previous section. In order to make the sequential equivalence work we needed a few assumptions on the external environment: input signals must change on the correct clock edge, no activity during reset, and no commands are injected into the unit when there are 16 active commands inside. The last requirement was due to the fact that illegally injecting too many commands caused internal counters in the power management machine to overflow. We note that these properties were enough to prove equivalence, but are not sufficient to prove even the simplest functional properties of the design. Furthermore we note that such assumptions are common in the world of practical equivalence checking. The equalities generated 697 proof goals. *GoodU* was broken down to 4 proof goals, and *GoodG* had one proof goal.

The well-founded order \prec was defined solely by examining the topology of the design. The full list of signals and dependencies is too large to be described here. However, we will note that there were combinational paths from inputs to outputs in each of the units G , U and Ob . This, of course, did not break the well-foundedness of the order since there are no combinational loops in the design. Overall there were 37 signal dependencies.

The required proof obligations are easily translated into model checking of temporal logic specifications. However, as we have described, many of our predicates actually describe a form of conditional sequential equivalence, thus for reasons of capacity we would prefer to use a dedicated equivalence checking

algorithm. We therefore translated those proof obligations into sequential equivalence problems. We explain the translation for the \hat{G} side; the method for the \hat{U} side is similar.

Our translation is a modification of a standard equivalence check between G and G' . The first adjustment is that we leave the inputs of G and G' assumed to be equivalent up to time $\tau - 1$ free, rather than tying them together as if we wanted to check standard equivalence. Those assumed equivalent up to τ are tied together as usual. Notice that a practical implication of this is that it is useful to add additional dependencies to the order \prec (while maintaining well-foundedness) in addition to those resulting from the topology of the design, because this results in more inputs tied together, which makes the equivalence checking problem easier. We added dependencies on almost all of G 's inputs, and many of U 's inputs; without this, the \hat{G} side would be intractable.

The second adjustment is to factor in the assumptions (including that untied inputs should be equivalent up to time $\tau - 1$). We define a signal *mask_outputs* that is asserted when an assumption is violated, and stays that way forever. We then modify each output of G and G' to have (the same) constant value when *mask_outputs* is asserted. Proving the equivalence of an output o between the two modified versions implies that there does not exist a computation that violates the proof obligation for o (because it implies there can not be a reachable state such that all assumptions hold and $o \neq o'$).

We verified Calc3 using IBM's internal tools. Equivalences (including conditional equivalences) were proven using SixthSense [1], IBM's sequential equivalence checking tool. The proof of each pair of outputs was done separately, since this lets SixthSense utilize its optimizations to their full power. The other properties (*GoodU* and *GoodG*) were proven using RuleBasePE [6], IBM's model checking tool.

The proof obligations on the U side went through relatively easily. On this side there were 493 proof obligations, and each took between 60 to 8500 seconds to prove. Overall the proof of the U side ran 124 hours on a 2.4GHz Opteron dual core machine with 8GB, running Red Hat Enterprise Linux 4.¹

On the G side there were only 209 proof obligations, but it was more difficult to prove. The overall runtime was 91 hours, with a single goal taking up to 21000 seconds (1500 seconds on average). There were several measures we took in order to reduce the problem size. For certain internal bits we were able to prove that the fence being down implies equality between the two versions. We then wired the two versions to use the same copy of the signal when the fence is down, and each their own copy when the fence is up. This resulted in a significant performance boost. Also, the tuning of SixthSense was extremely important. The first runs ran out of memory even for the simplest goals. It took some fiddling with parameters to be able to prove all goals.

During the verification of \hat{G} we discovered three bugs in the implementation of *PM*. Each resulted in traces in which the power gated version of the adder

¹ This is the cumulative time of the whole verification effort on a non-trivial industrial example, and is negligible compared to years of CPU time used for simulation.

failed to output a command when the non-power gated version did. In each trace it was obvious that the reason was because *PM* failed to turn on the adder on time, and from there it was easy to diagnose the error in *PM*.

5 Conclusions and future work

We have presented a methodology for the verification of power gating, based on comparing a power gated design to a version of itself in which power gating is disabled. In order to be able to deal with real world designs, we take a compositional approach in which we check each unit for partial equivalence across a suitable set of assumptions. In contrast to general assume-guarantee reasoning, which can be difficult to employ because of the manual effort involved in coming up with appropriate assumptions, the goal of our verification effort is one specific and simple formula – sequential equivalence – and this greatly simplifies the task of figuring out what the assumptions should be. We have shown the feasibility of our method by applying it to a non trivial execution unit.

Future work is to extend our methodology to designs in which power gating changes the external behavior of the chip (for instance, pipelines that stall when the needed unit is not available), to apply it to real-life designs under development at IBM, and to develop a tool that automates much of the work of coming up with the assumptions (for example, those that deal with the clock edges and activity during reset can be derived automatically) and deciding on the well-founded order.

Acknowledgments Thank you to Jason Baumgartner and Hari Mony of the SixthSense team for their help in getting the equivalence check on *G* to run.

References

1. Jason Baumgartner, Hari Mony, Viresh Paruthi, Robert Kanzelman, and Geert Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *ICCD'06*, October 2006.
2. M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual*. Springer US, 2007.
3. Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *ICCAD'04*, pages 58–65, 2004.
4. Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV'98*, pages 110–121, 1998.
5. Carl Pixley. A theory and implementation of sequential hardware equivalence. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(12):1469–1478, 1992.
6. RuleBasePE. http://www.haifa.ibm.com/projects/verification/RB_Homepage/.
7. C. A. J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(7):814–819, 2000.
8. B. Wile, J. C. Goss, and W. Roesner. *Comprehensive Functional Verification - The Complete Industry Cycle*. Elsevier, 2005.