Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Jan Láník

Thèse dirigée par **Oded Maler**
et codirigée par **Fahim Rahim**

préparée au sein **Verimag**
et de **EDMSTII**

Power Reduction in Digital Circuits

Thèse soutenue publiquement le **16. Juin 2016**,
devant le jury composé de :

Ahmed Bouajjani
Université Paris Diderot, Président
Sharad Malik
Princeton University, Rapporteur
Roderick Bloem
IAIK, TU Graz, Rapporteur
Julien Legriel
Synopsis, Inc., Grenoble, Examinateur
Dejan Nickovic
AIT Austrian Institute of Technology GmbH, Examinateur
Oded Maler
Verimag, Université Grenoble Alpes, Directeur de thèse
Fahim Rahim
Synopsys, Inc., Grenoble, Co-Directeur de thèse

Acknowledgement

The years I spent in Grenoble working on this thesis were enriching both professionally and personally. I would like to thank my supervisor Oded Maler for his guidance and support. He was always there to provide counsel and always found time for fruitful discussions with me even at times when he was occupied by other things from his wide portfolio of interests.

I thank my co-supervisor Fahim Rahim, who coordinated my work in Atrenta/Synopsys and introduced me to the world of industrial research.

I thank Ahmed Bouajjani, Dejan Nickovic and Julien Legriel for participating in my jury and special gratitude belongs to the reviewers Sharad Malik and Roderick Bloem.

Julien also worked with me on the development and implementation of Activity Triggers together with Emmanuel Viaud and I thank him both for the countless hours that we spent in meetings and for all the advice and help without which the second half of my thesis could never exist. Additional thanks go to the Atrenta's formal method experts Hans Peter and Nikos Andrikos who taught me much about the practical aspects of formal verification in the context of digital circuits. I thank also the rest of my colleagues at Atrenta who created a friendly cooperative atmosphere and who were always ready to answer my questions and provide assistance. Special thanks go to Audrey Patruno for much needed administrative support especially during my early days in Grenoble when I didn't speak almost a word in French.

I thank all the staff, researchers and students at Verimag for creating an unique environment full of interesting people and inspiring ideas. I thank especially my academical siblings Irini Eleftheria Mens, Abhinav Srivastav and Dogan Ulus for being not just colleagues but also very good friends.

I thank all my other friends who made my time in Grenoble better and also to my friends in the Czech Republic that didn't forget about me.

Finally, I want to express great gratitude to my parents and the rest of my family for their love and support. And to Lucia for her patience.

Résumé

Le sujet de cette thèse est la réduction de consommation dans les circuits digitaux, et plus particulièrement dans ce cadre les méthodes basées sur la réduction de la fréquence de commutation moyenne, au niveau transistor. Ces méthodes sont structurelles, au sens où elles ne sont pas liées à l'optimisation des caractéristiques physique du circuit mais sur la structure de l'implémentation logique, et de ce fait parfaitement indépendantes de la technologie considérée. Nous avons développé dans ce cadre deux méthodes nouvelles. La première est basée sur l'optimisation de la structure de la partie combinatoire d'un circuit pendant la synthèse logique. La seconde est centrée sur la partie séquentielle du circuit. Elle consiste en la recherche de conditions permettant de détecter qu'un sous-circuit devient inactif, de sorte à pouvoir désactiver ce sous-circuit en coupant la branche correspondante de l'arbre d'horloge, et utilise des méthodes formelles pour prouver que la fonctionnalité du circuit n'en serait pas affectée.

Abstract

The topic of this thesis are methods for power reduction in digital circuits by reducing average switching on the transistor level. These methods are structural in the sense that they are not related to tuning physical properties of the circuitry but to the internal structure of the implemented logic and therefore independent on the particular technology. We developed two novel methods. One is based on optimizing the structure of the combinatorial part of a circuit during synthesis. The second method is focused on sequential part of the circuit. It looks for clock gating conditions that can be used to disable idle parts of a circuit and uses formal methods to prove that the function of the circuit will not be altered.

Keywords

power reduction, clock gating, harware design, synthesis, RTL, switching

Contents

We see portability in electronics being a continuing requirement, higher functionality, better battery life, requiring lower power for the actual electronics.

David Milne

1 Introduction

Every computing and information-processing device consumes energy and produces heat [27]. During the last two decades, power consumption became one of the most important design concerns for electronic devices. The need for low power electronics stems from multiple factors. Besides the general demand for green computing in order to save non-renewable resources and protect the environment, power optimization is crucial for almost all types of devices, with each class of applications bringing its own particular motivation for low power. For instance, in modern data centers the power bill is on the same scale as the hardware cost [19] and significant resources must be put into an efficient cooling infrastructure. In portable devices there is a need for maximizing the operation/standby time per one battery recharge/replacement. The expected future expansion of the 'internet of things' will include extremely small wireless sensors and similar devices that will recharge by wireless techniques or by energy scavenging as direct battery charging or replacement would be impractical [58]. This implies extremely limited power budget for such devices so that energy-efficient design will directly determine their computational capabilities[59]. Finally, even in high performance chips, where the cost of energy may not be considered an issue, the operation temperature of the chips is limited and power efficient design is important in order to maintain the heat generated by the chip at a level that can be efficiently cooled down [55]. We can conclude that in the last decades power consumption has become a no less important performance measure than traditional area and speed [16, 6, 60, 8].

In this thesis we focus on reducing dynamic power consumption in integrated circuits, that is, the power dissipated while charging/discharging the transistors, which happens when logical gates in the circuit change their value. It is often the case that some of such switches are redundant, i.e. they are not necessary for the proper functioning of the circuit, but nonetheless they keep consuming power. Reducing such redundant switches is thus

one of the main concerns for low power design. We propose two novel methodologies to avoid some of such redundant switches and to transform circuits into more energy efficient forms while preserving functionality.

The first method (*power-aware synthesis*, published in [45]) is focused on reducing the average switching in combinatorial logic based on a statistical model of the input, leveraging the fact that the probability of switching is different for individual logic elements and there is typically a high level of correlation between the logical values of various signals in the design. This method is intended to be used during hardware synthesis in order to obtain a physical realization that will produce less internal switches when performing a typical computation thus reducing the long term power consumption.

The second method (*activity trigger detection*, published in [46]) is focused on reducing power primarily in sequential logic and its clock distributing network of by using the clock gating technique. Clock gating is an industry standard power reduction strategy that disables parts of a design when their function is not required. The essence of this work is to identify events that delimit the active periods of sub-circuits, thus allowing the application of clock gating to larger sub-circuits, using simple conditions that do not add a large overhead. We also provide a formal verification flow to prove the correctness of the potential clock gating opportunities that were found by the method. This method has been part of my work as an industrial PhD student in Atrenta (later acquired by Synopsys) and implemented within their commercial EDA tool.

The two power reduction methods presented in this thesis are independent and target different stages in the low power design flow. Hence, after discussing basic fundamentals of switching and power in Chapter 2, a separate chapter for each method follows. These main chapters will contain the fundamental theory, related work, description of the method, experimental results and a conclusion for each method separately. The combinatorial circuit synthesis method will be described in Chapter 3, the clock-gating method in Chapter 4. We will finish with a common conclusion in Chapter 5.

The whole VLSI approach is a triumph of engineering and industrial manufacture, and it's pity that ordinary people in the street don't appreciate how marvelous and beautiful it all is!

Richard Phillips Feynman

2 Circuits and Power

2.1 Power Dissipation in CMOS Logic Gates

In this section we describe the main *power dissipation* issues in complementary metal-oxide-semiconductor (CMOS) static logic [68]. At this point it is useful to clarify the difference between energy consumption and dissipation in electronic circuits. Energy consumption is the total energy that leaves the power supply and moves into the circuit. Energy dissipation is the amount of energy that leaves the circuit and is discharged into the ground and transformed to heat. Due to the law of energy conservation, these two values are equal and hence interchangeable.

An integrated circuit (IC) consists of logic gates that perform simple Boolean operations. Let us consider an inverter gate, a commonly used example to describe CMOS principles that are relevant as well to more complex gates. Figure 2.1 shows the schematics of an inverter in CMOS technology. The transistors, wires, and all the electronic circuitry driven by the gate output have an inner capacitance. Setting the gate output value to logical 1 is realized by charging this capacitance up to the supply voltage (*Vdd*). Conversely, discharging the capacity into the ground (*Gnd*) represents a switch of the gate to logical 0. For an ideal transistor, the charge from the source is never dumped directly to the ground, as one of the transistors is always closed. All energy consumption comes from charging the capacitance while switching from 0 to 1 and all energy dissipation (heat generation) comes from discharging into the ground while switching the gate back from 1 to 0. The power consumed by this phenomenon is called *switching power*.

However, real transistors are not ideal, which results in two additional major components of power consumption. One imperfection of transistors is that they do not switch infinitely fast. During the switching, the transistor stays for a very short time in an intermediate state where the resistance is quickly changing and the transistor is half open,

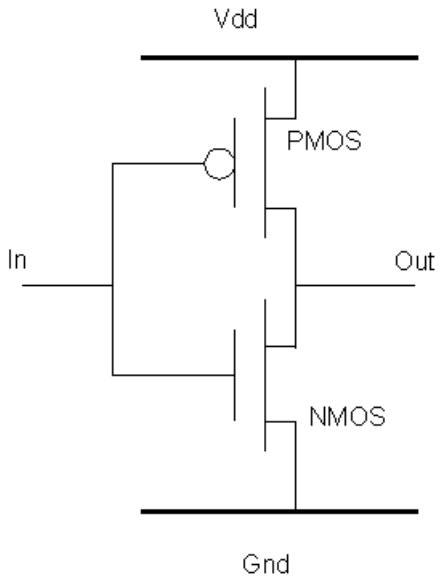


Figure 2.1: CMOS inverter circuit, image source:http://www.ics.uci.edu

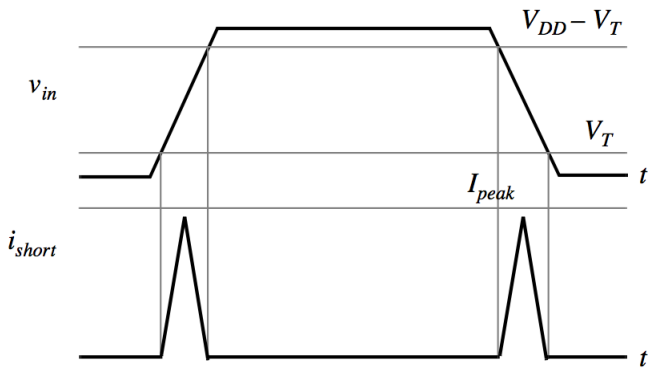


Figure 2.2: CMOS inverter short circuit, image source:[59]

which allows passage of some current. This can lead to a short circuit path being formed temporarily between voltage supply and ground. In the inverter case, if the input changes, one of the transistors is closed and the other is open at the same time and until the first is fully closed, some current can flow directly to the ground, consuming the so called *short circuit power* (Figure 2.2).

Finally, physical imperfections of various electronic components allow some direct leakage of the current from the power supply and the ground. The reasons encompass multiple physical phenomena like sub-threshold leakage, stacking effect and gate tunneling [59]. Leakage current is small compared to the currents responsible for switching, however, it is consumed continuously as long as the circuit is powered, even when there is no actual switching of the transistors. Until recently, leakage was accountable for a relatively small portion of the overall power dissipation, however the effect of leakage tends to be stronger as transistor sizes and supply voltages scale down and leakage power is now becoming an increasingly important factor in the overall power consumption [42, 59].

The complete energy consumed by a CMOS gate during a given time can be summarized by the following equation:

$$E \approx (C_L \cdot C_{SC}) \cdot Vdd^2 \cdot N + I_{leak} \cdot V_{dd} \cdot t. \tag{2.1}$$

The symbols in Equation 2.1 should be interpreted as follows:

E	...	total energy consumed per a unit of time
N	...	number of switches in the gate value per a unit of time
C_L	...	gate capacitance
C_{SC}	...	capacitance used to model the short-circuit power
V_{dd}	...	power source voltage relative to the ground
I_{leak}	...	leaking current
t	...	(time) duration

Power reduction strategies typically focus on minimizing some element in equation 2.1. Scaling transistor sizes and adjusting other physical properties can focus on minimizing C_L, C_{SC} and I_{leak}. Another approach is scaling down V_{dd} which has been reduced from $5V$ to less than $1V$ in the last two decades [59, 12]. While using these physical methods one should be careful not to break the balance between conflicting optimization goals. For instance, scaling down voltage reduces power, but at the same time induces a higher delay, therefore decreasing frequency. Furthermore, with small source voltage, we need to reduce also the threshold voltage values in the transistors, which leads to an exponential increase in leakage.

2.2 Dynamic power and switching activity

In this thesis we focus exclusively on reducing switching activity, a number of switches actually performed by gates. We see that this has linear impact on switching and short-circuit power, which are together called *dynamic power*. Abstracting away the physical characteristics, the energy consumed as dynamic power can be expressed as

$$E_{dyn} \approx energy\_per\_switch \cdot N \tag{2.2}$$

Note that it doesn't really matter that the energy cost of charging the gate to supply voltage is typically higher than discharging it, as for every charge there will be a discharge and therefore we can enumerate the average energy consumed by a switch as

$$energy\_per\_switch = \frac{energy\_per\_charge + energy\_per\_discharge}{2} \tag{2.3}$$

In this thesis we consider principally clocked or synchronous circuits, where the operations are performed periodically at a rate given by a clock signal of a fixed frequency f. In such circuits the maximal dynamic power consumption over time t is attained by a gate if it switches every clock cycle and hence $N = f \cdot t$. Dividing the energy by time t, we get the maximal dynamic power as

$$P_{max} = energy\_per\_switch \cdot f. \tag{2.4}$$

However, gates do not typically switch at every clock cycle, so the average dynamic power consumption P_{avg} of a gate performing some computation is smaller than the maximum P_{max}. The ratio

$$\alpha = \frac{P_{avg}}{P_{max}} \tag{2.5}$$

is called the *switching activity* and is always between 0 and 1. Switching activity can be interpreted alternatively as the probability for a given gate to switch in a given clock cycle. The average power consumption can be expressed as

$$P_{avg} = \alpha \cdot energy\_per\_switch \cdot f. \tag{2.6}$$

So far we assumed that the gate capacitance is charged only when the logical value on the output of the gate changes. In reality, mostly due to the fact that the changes in signals take some time to propagate through the circuit, it can happen that changes in different inputs of a gate are desynchronized and the gate is partially charged/discharged for a very short time followed by immediate discharge/recharge. Figure 2.3 shows an example of a circuit where the inputs to the AND gate are not synchronized due to an unequal length of the paths along which the signals propagated, which creates a glitch. This behavior is generally undesirable as power is spent without an actual useful computation being performed. Glitches are in general responsible for a significant fraction of dynamic power [26] and designers are putting a great effort to limit glitches using various techniques [21, 10], for example, reducing the imbalance between the lengths of different paths. We didn't considered glitching in our power aware synthesis technique, however the method is designed not to introduce large differences in path lengths. The savings achieved by activity trigger analysis apply also to glitching.

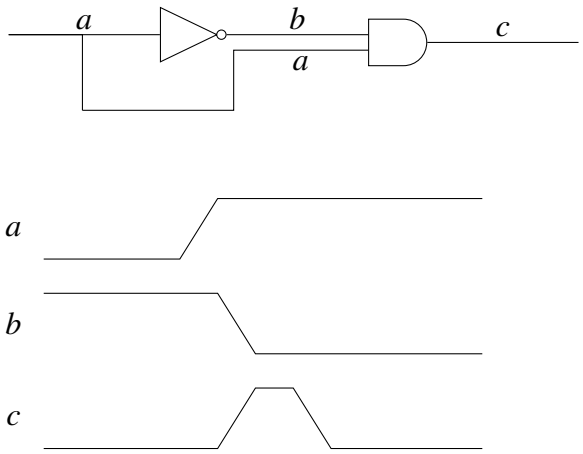


Figure 2.3: An example of a glitch. The transition in b happens slightly later after the transition in a due to the physical delay of the inverter. This, combined with the physical delay of the AND gate leads to a short period (glitch) when the signal c is high, even though logically it should always remain low (as $c = a \wedge b = \neg a \wedge \neg a = 0$).

2.3 Design flow

To put our power reducing techniques into context, we summarize in Figure 2.4 the general flow of application specific integrated circuit (ASIC) design. More information about this flow can be found in textbooks such as [62]. Here we focus only on the steps that are relevant to our methods, that is *logic design*, *circuit design*, and *physical design*.

We can see each of the design phases as a process that transforms a more abstract circuit representation into more concrete form. The part of the flow relevant to this thesis works with four main forms of circuit description:

1. **Behavioral level description** A circuit description in a language that contains high-level software-like constructs such as loops, conditionals or sub procedures. It describes the interface and function of the circuit rather than actual physical structure. Behavioral level languages includes Verilog [38], VHDL [39], System Verilog [37] and SystemC [36].

2. **Register Transfer Level (RTL)** Describes interface ports, registers, key nets and buses of the design and defines how register values should be updated. This is specified using simple Boolean expressions that may contain also simple arithmetical macros like addition or multiplication.

3. **Netlist** Set of gates, ports and registers and their interconnections. The gates and registers can be abstract (associated only with the logical function), however at some point they need to be mapped into standard cells.

4. **Physical Layout** Floor plan of a chip that contains actual physical positioning of the standard cells and interconnecting wires as well as supporting structures like clock distribution network.

During the ASIC design process, the designers move from one circuit representation to another (more concrete) using various techniques that can be grouped in the following stages:

1. **Logic Design** In this stage the behavioral description is translated into RTL. Logic minimization can be performed at this point to simplify the combinatorial logic expressions.

2. **Circuit Design** The abstract RTL registers and combinatorial logic is translated into real gates and registers that can be manufactured. These gates are provided in a technology library file by the manufacturer (foundry). Typically this process has two stages where the RTL is first translated into a netlist consisting of a limited number of abstract gates (e.g. only 2 input AND gates and inverters) and then physical gates from the technology library are structurally mapped onto this abstract netlist by a cost optimizing mapping algorithm.

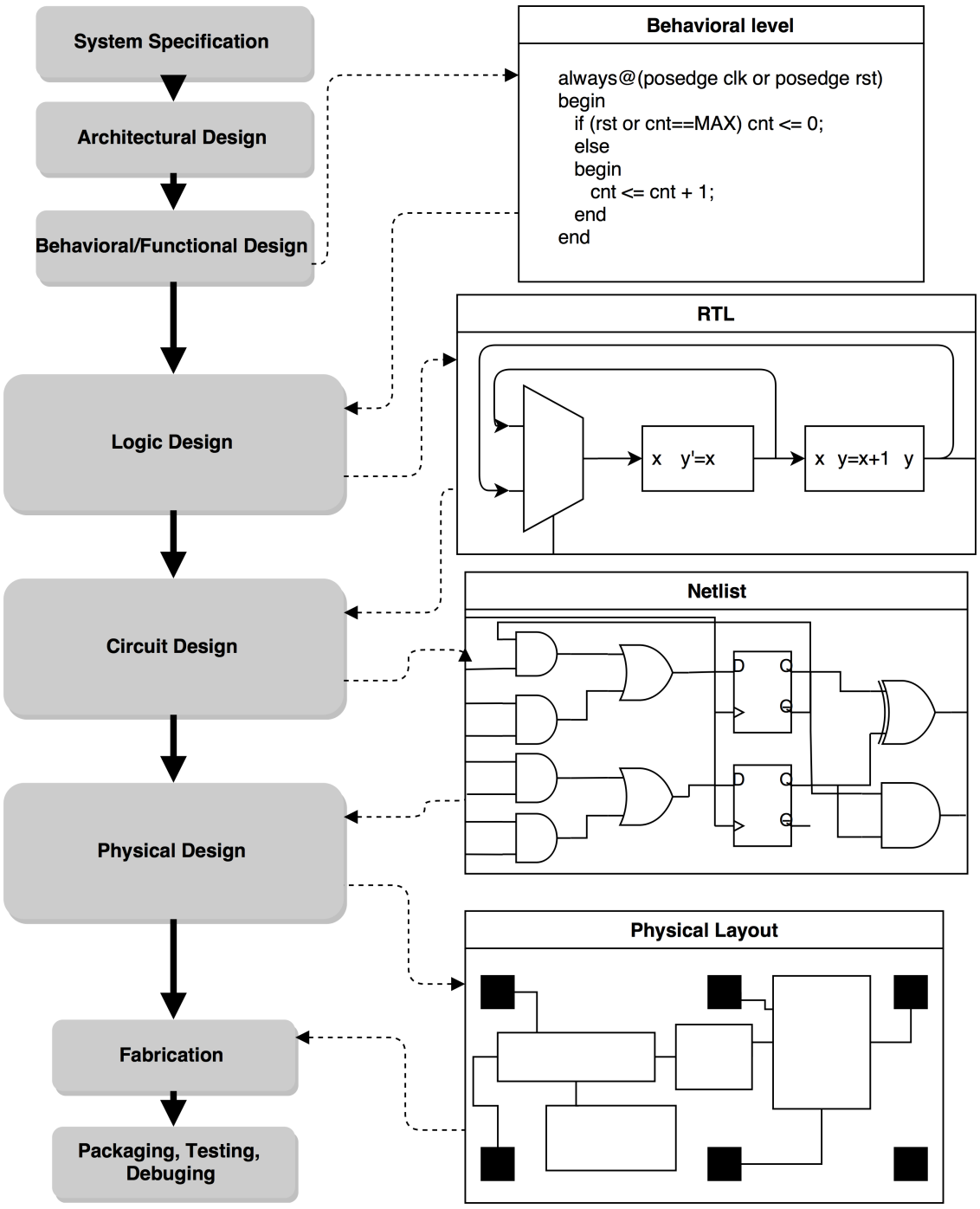


Figure 2.4: ASIC design flow and related circuit representations.

3. **Physical Design** In this phase physical structures such as transistors are placed on a chip and the wires connecting them are routed. Among other things, the clock signal has to be routed to each register and the clock gating cells are placed in this step.

2.3.1 Position of our techniques in the design flow

Power-aware synthesis technique is applied at the beginning of the circuit design stage (that is, translation of RTL into technology independent abstract netlist). The method minimizes switching in the resulting abstract netlist based on a stochastic model of input.

Many performance measures of a chip like delay or power consumption can be accurately estimated only at the level of physical layout, however the decisions done higher in the flow have serious impact on these costs. Furthermore, any change in the design is much less expensive (in time and financially) to implement if it occurs earlier in the flow. For instance fixing exceeded power budget leads to much shorter delay in production if it is discovered at RTL as opposed to discovering the issue after placement and routing or in extreme case after fabrication. For this reason, there is a demand for automatic tools that can estimate power consumption and detect power reduction opportunities already at the RTL even though they are less accurate than they would be at the physical level. Activity trigger detection works at the RTL suggesting clock gating opportunities. The integration into the Spyglass tool from Atrenta allows the user to estimate potential power reduction that will be achieved at the physical level using the RTL power estimation tool in Spyglass.

2.3.2 Formal model for circuits and switching

A digital circuit consists of various combinatorial logic gates and sequential memory elements. For the purpose of formal modeling we will consider the state of the circuit to be the current Boolean valuation of *all* the signals (inputs, gate outputs, memory elements). The dynamics of the circuit will be modeled by a transition relation that conveniently hides all the structural details. This definition differs from traditional circuit model in the sense that the states are not restricted to the states of the sequential elements. This assumes some discrete time domain and instantaneous signal propagation, a reasonable assumption for well-timed synchronous circuits. For simplicity, we assume that every design has one initial state. In reality, although the initial state of a circuit is typically undefined, a unique state is often reached via a reset sequence.

Definition 1. *A digital circuit* (design) is a tuple (X, Q, T, q_0), where

- X is a finite set of variables that correspond to signals in the circuit.
- Q is the state space of the circuit. Each state is of the form $x : X \to \mathbb{B}$ and has to satisfy combinatorial constrains imposed by the structure of the circuit.

- $T \subseteq (Q \times Q)$ is a transition relation describing the circuit dynamics.
- $q_0 \in Q$ the initial state of the design.

The semantics of the circuit, the execution traces it generates, is defined using paths in its automaton model.

Definition 2. An *execution trace* σ of a circuit $D = (X, Q, T, q_0)$ is a sequence $q[0], \ldots, q[\tau]$ of states, where

- $q[0] = q_0$,
- For all $1 \leq i \leq \tau$ it holds that $(q[i-1], q[i]) \in T$.

We use notation $x[t]$ to refer to the value of a signal variable $x \in X$ at the time t when the execution trace is clear from the context.

To model the stimuli applied to the circuit's vector, we introduce a notion of *input vector*.

Definition 3. An *input vector sequence* is an execution trace projected only to circuit inputs. The execution trace is determined by it's input vector and the initial values of signals. An *input vector* is one time frame of an input vector sequence.

Definition 4. Assuming a discrete time domain $\{0 \ldots \tau\}$, we say a signal a

- *rose* in time $t+1$ if $a[t] = 0$ and $a[t+1] = 1$,
- *fell* in time $t+1$ if $a[t] = 1$ and $a[t+1] = 0$,
- *switched* in time $t+1$ if $a[t] \neq a[t+1]$,
- *was stable* in time $t+1$ if $a[t] = a[t+1]$.

Definition 5. Let σ be an execution trace of a circuit D under a discrete time domain $\{0 \ldots \tau\}$. The activity of a signal a, $\alpha(a)$ is the number of $t \in \{1 \ldots \tau\}$ such that a switched in time t.

One execution trace corresponds to a deterministic behavior of the circuit under some fixed stimulus of it's inputs. As circuits are supposed to work under multiple input vectors that are not known a priori, we are interested in average *activity*. To define it, we introduce a stochastic model for circuit behavior based on a probabilistic distribution $\mathcal{P}$ over possible input vector sequences $\mathcal{V}$ to a design D. The probability of an input vector v being applied as a stimulus to D is then expressed as $\mathcal{P}(v)$.

Definition 6. Let D be a design and $\mathcal{P}$ the probability distribution over the set of possible input vector sequences $\mathcal{V}$. For a signal a, the activity of a is

$$\alpha(a) = \sum_{v \in \mathcal{V}} \mathcal{P}(v) \cdot \frac{|\{t \in \{1 \ldots \tau\} \mid a[t] \neq a[t+1] (\text{in the context of } v)\}|}{\tau - 1} \tag{2.7}$$

I keep hearing about battery innovation, but it never makes it to my phone.

Evan Spiegel

3

Power Aware Synthesis for Combinatorial Power Reduction

Synthesis of combinatorial (and sequential) logic [43, 15, 61, 30] from higher level descriptions to technology dependent standard cells is part of the circuit design stage of the ASIC design flow (Figure 2.4) and one of the core activities in Electronic Design Automation (EDA), well-studied in academic research and implemented in powerful commercial tools. This is the hardware analog of optimizing compilation, indispensable tool in producing efficient chips. Traditionally, the major optimization objectives in synthesis have been area and speed, the latter associated with the longest path from primary inputs of the combinatorial logic to its outputs, which is related to the minimal time span of a clock cycle in a sequential circuit. In this work we develop a new synthesis algorithm geared toward decreasing power consumption by reducing the expected number of switches in the circuit, an important factor in its dynamic power consumption.

Fig. 3.1 sketches a typical logic synthesis flow. Starting from a high level RTL specification of logic blocks, we extract the Boolean relationships between single bit signals. This is called multi-level logic specification. Then the circuit is brought into a form of an And-Inverter Graph (AIG), which is an abstract netlist consisting solely of AND and NOT gates. This representation is then mapped into a concrete technology of standard cells admitting physical properties such as size and electrical characteristics. Syntactically, AIGs are composed from two input AND gates (2ANDs) but by collapsing together all NOT-free 'cones', we obtain a semantically-equivalent function constructed from AND gates of unbounded fan-in (arity). Part of the technology-dependent mapping can be viewed as decomposing those ANDs into networks of 2ANDs and this is the problem we address in this chapter.

Dynamic power consumption of Boolean gates is associated essentially with their

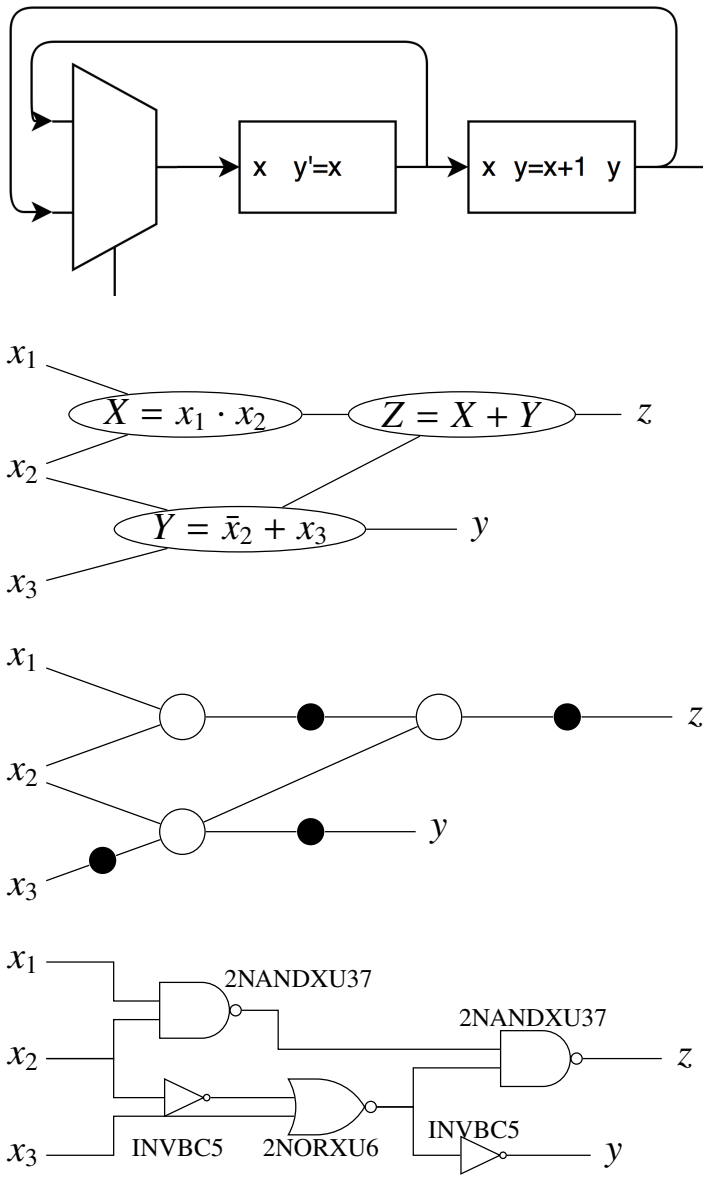


Figure 3.1: A circuit synthesis flow: from RTL to multilevel logic specification to technology independent netlist to standard cells.

switching between 0 and 1. In this work we consider synchronous combinatorial circuits that process sequences of input vectors. For each input vector, a circuit propagates values from input to output ports until it stabilizes and then reads the next input. The overall number of switches associated with a pair of inputs is the number of gates whose stable value for one input is different from their value for the next input. For one such pair it is possible to steer the synthesis process and obtain a circuit with significantly less switching compared to other arbitrary circuits that realize the same function. But of course, any circuit will process during its lifetime a long sequence consisting of diverse consecutive pairs of input vectors and optimizing synthesis with respect to all those is a challenging problem.

One natural approach is to define some probability function over sequences of input vectors, induced, for example, by a Markov chain which generates them. However, even the evaluation of the expected number of switches in a *given* circuit is an intractable problem for non-trivial probabilistic generators with many input variables. As an alternative we develop in this thesis a switching-aware synthesis procedure which optimizes the circuit relative to a *reference sequence* supposed to represent a typical input. In essence, the algorithm estimates the expected amount of switching associated with a conjunction of any pair of input variables and then solves an optimization problem to decide which variables to pair together as inputs to a 2AND gate. The procedure obtains quite a good switching reduction compared to arbitrary realizations of the same function by circuits of similar topology.

We then study the question of optimization with respect to inputs generated by Markov chains of small description size, that is, networks of sparsely-interacting 2-state probabilistic automata. We use such networks to generate the reference (training) sequences and then measure the performance gains on other sequences generated from the same model. We perform experiments on models of varying degree of variable dependencies and other assumptions on the inputs and obtain significant reduction in switching activity.

We introduce two small circuits - a reduced model of an instruction decoder and an open source serial peripheral interface circuit. We evaluate our procedure on these models under probabilistic assumptions concerning the input stream. Finally, we explore the effect of other optimization techniques and of technology mapping to the savings achievable by our method.

3.1 Circuit Synthesis with AIGs

The circuit synthesis from RTL to a technology dependent netlist is a multiple step process. An example of such a process, focused on combinatorial logic, is illustrated on Figure 3.1. First, multilevel logic specification is derived from RTL description. This is then, after optimization, translated to a technology independent netlist composed of abstract gates. These abstract gates come from a limited number of types, each type corresponding to a simple Boolean function. Different approaches, e.g., Boolean Expression Diagrams [1], And-Inverter Graphs (AIGs) [29] or Reduced Boolean Circuits [34], use different elementary functions. For our method we focus on the flow that uses AIGs as implemented, for example, in *ABC* [53, 63], a tool for circuit synthesis and verification developed at Berkeley Verification and Synthesis Research Center and used in academia as well as in commercial EDA solutions [52].

We apply our optimization technique at this stage – on the abstract netlist in the form of an AIG. We transform the AIG into a functionally equivalent form with a lower average switching during typical use of the circuit. Afterwards, the netlist is translated into a technology dependent form consisting of standard cells. The information about the logical and physical properties of the available standard cells is typically provided in a file known as *standard cell library*. Such a library may contain hundreds of cells. The logic functions provided by cells are translated into AIG form. These small AIG fragments are then structurally mapped onto the AIG representation of the circuit in a manner that optimizes traditional performance measures such as delay and area based on the physical properties of the associated standard cells that are provided in the library [41, 49]. This process is called *technology mapping*.

3.1.1 AIG data structure

An And-Inverter Graph (AIG) is a Directed Acyclic Graph (DAG). Nodes and edges in AIG have associated attributes that define the type of the node or edge. There are four types of nodes:

1. **Inputs:** Represent inputs to the circuit. Nodes labeled as inputs have no incoming edges.

2. **AND nodes:** Represent 2-input AND gates.

3. **Outputs:** Represent circuit outputs, no outgoing edges.

4. **Flip-flops** Represent sequential elements. They must have exactly one input edge and an initial value. A global clock is assumed to define the semantics - every clock cycle the input to the flip-flop is sampled and saved as its new value.

Furthermore, there are two types of edges: *direct* and *complemented*. The first supplies the value of the source as an argument to the destination node directly, the second first inverts the value.

AIGs do not provide a canonical representation of sequential circuits and may contain some redundant structures. However, modern tools apply efficient techniques that significantly reduce the redundancy by unifying many isomorphic nodes of the graph [54]. For instance, it is ensured that a node can never have two incoming edges from the same source and that no two AND nodes have the same input.

In this chapter we are interested in switching reduction in combinatorial logic. The combinatorial part of the circuit logic can be expressed in form of flip-flop free AIG. The transformation is straightforward - the inputs to the flip-flops become new combinatorial outputs and the outputs become new combinatorial inputs. In total two new nodes are added for each removed flip-flop node. This combinatorial AIG serves as the main circuit representation in our power-aware synthesis method.

3.2 Importance of Input Characterization for Switching

Whether a gate switches in a given cycle is determined by the behavior of its inputs. For instance, when a 32-bit adder adds very small numbers, the higher bits typically do not switch. If large number never occur we could use a smaller adder but if we need occasionally to use large numbers we will need all bits. Nevertheless, the information on the low probability of the higher bits operations can be useful for guiding the synthesis toward a solution which is power efficient on the average.

Because our interest is in AIGs, we will be mainly interested in the AND and inverter gate switching. For a signal a we denote the probability of $a = 1$ in a given clock cycle (time) t as $p_t(a)$. We say that signal a switched in time $t + 1$ if $a[t + 1] \neq a[t]$. The probability of such a switch is

$$P(a[t + 1] \neq a[t]) = p_t(a) \cdot (1 - p_{t+1}(a)) + (1 - p_t(a)) \cdot (p_{t+1}(a)). \tag{3.1}$$

If we assume that the probability doesn't depend on time ($p_t(a) = p_{t+1}(a) = p(a)$), the switching probability can be simplified to

$$2 \cdot p(a) \cdot (1 - p(a)). \tag{3.2}$$

and is equivalent to the activity of a.

Now, consider an AND gate with input signals a and b. The conjunction signal is high if and only if both a and b are high. Hence

$$p(a \wedge b) = p(a) \cdot p(b). \tag{3.3}$$

For the inverter, we have

$$p(\neg a) = 1 - p(a). \tag{3.4}$$

These equations are at the core of many switching estimation tools, however they contain an inherent deficiency. Consider the circuit in Figure 3.2 for which the inputs of the gate are always different and hence the AND gate value is equivalent to constant 0. There is no switching happening in such a gate, even though Equations 3.3 and 3.1 will give us positive numbers for $0 < p(a) < 1$. This is because Equation 3.3 assumes that there is no correlation between different circuit signals, which is not true for this circuit from as well as for the majority of real circuits. We call this type of correlation between different combinatorial signals *spatial dependency*.

Another commonly-used assumption is that of temporal independence, i.e., the value of the signal in time t doesn't depend on t or on previous values of the signal. This is also often not the case as circuits contain sequential elements that introduce temporal correlation as illustrated in Figure 3.3. The inputs to the AND gate are a signal and its previous value saved in a sequential element. The values change every cycle, hence the probabilities of being high are equal to 0.5 for both inputs and the probability that the AND gate is high should be equal to 0.25 according to Equation 3.3. However, the inverter in the sequential loop ensures that the signal value is always different than its previous value,

resulting in zero switching at the AND gate. This could be also seen just as spatial correlation between two inputs of the AND gate as in the Figure 3.3, however the correlation source in this case is clearly temporal and spatial dependency would not be sufficient in case of more sophisticated logic being used to update the register.

This shows that for accurate modeling of switching it is desirable to have a stochastic model of the input that includes both spatial and temporal dependencies.

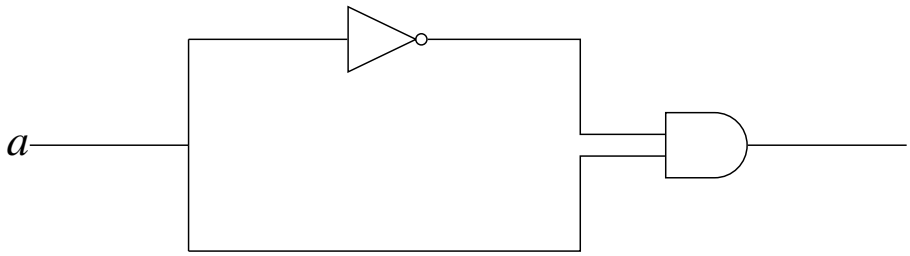


Figure 3.2: Effect of spatial correlation on switching.

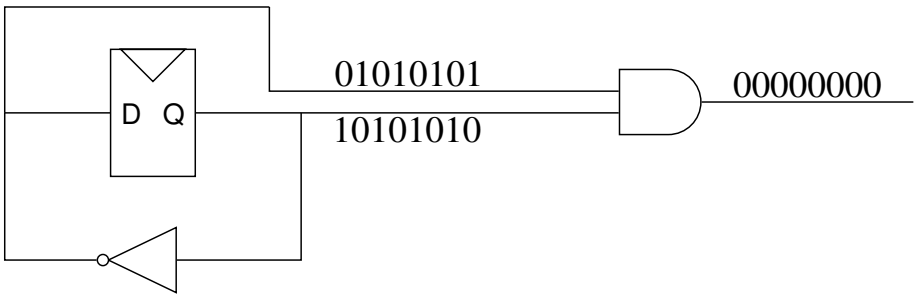


Figure 3.3: Effect of temporal correlation on switching.

3.3 AND Cone Decomposition

As combinatorial AIGs are composed from AND gates and inverters only, we can merge some of the neighboring AND gates to multiple-input-AND gates. The resulting multiple-input-AND gate graph will be semantically equivalent to the original AIG. This is illustrated in Figure 3.4 where the 2-input-AND (2AND) nodes of the original AIG are shown as white circles and complementing edges are marked with a black dot (that represents an inverter). The overlying triangles are the multiple output gates in the transformed graph that we call cones.

To transform an AIG into *cone* form, we need to identify the sub-structures of the AIG that can be safely merged. We do this by finding so called cut points that delimit the cones. A sub-structure in AIG can not be merged if it contains an inverted edge. Furthermore, if a node is referred from two different places, we can't hide it inside a multiple-input-AND gate, as its value would not be available in the circuit. Therefore, we denote all AIG nodes that have an outgoing complementing edge or have more than one outgoing edge as cut points. These cut points are at the top of the cones. The cone associated with a *cut point* contains all the nodes that are backward reachable from the cut point using a search algorithm that stops at non-complemented edges and another cut points. This procedure is shown in Figure 3.4. The correctness follows from the fact that each cone is by construction a binary tree composed exclusively of 2AND gates. Such a structure is equivalent to a multiple-input-AND gate by commutativity and associativity of logical conjunction.

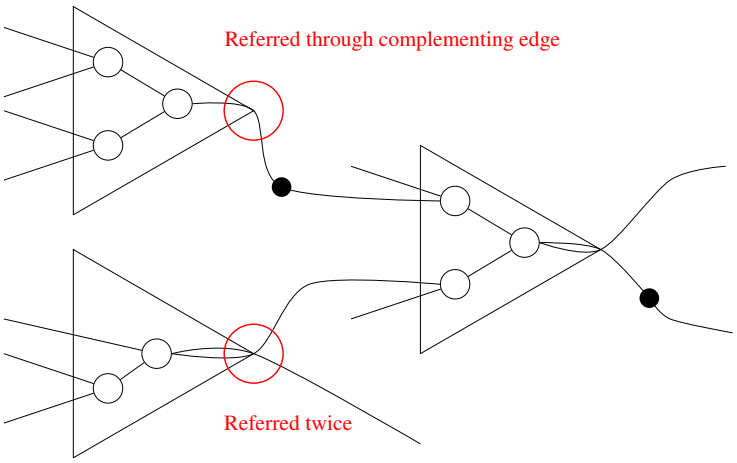


Figure 3.4: Partitioning AIG into AND cones.

Now consider an inverse procedure. We start with a directed graph of cones and we want to produce an equivalent AIG. This AIG is not unique, as every n input cone can be represented in AIG as an arbitrary binary tree of $n-1$ 2AND gates. All these trees are functionally equivalent but their average switching may be different. Our goal is to transform the original AIG first to the cone form and then decompose each cone into the tree with minimal or reasonably small average switching under some stochastic model of the incoming input vectors.

3.4 Problem Statement

Our starting point is a DAG having AND cones as vertices and (complementing or directing) edges. This is functionally equivalent to a Boolean circuit constructed from unbounded fan-in AND gates and NOT gates. Our goal is to replace the AND gates by 2AND gates (decomposing the cones), yielding an equivalent circuit C in the form of AIG. Once we have a good solution for the problem of decomposing a single AND cone, we can apply it to every cone separately and solve the problem for the whole circuit.

From now on we consider a Boolean function

$$f : (x_1, \ldots, x_n) \mapsto x_1 \wedge \cdots \wedge x_n$$

and a target circuit C which is a properly structured binary tree with edges directed towards the root. Non leaf nodes of this tree are 2ANDs of the form

$$g : (y_i, y_j) \mapsto y_i \wedge y_j.$$

Note that the number of 2ANDs in C is always $n - 1$. The n leaf nodes represent the inputs of the circuit, hence each leaf is uniquely labeled by a variable from $x_1, \ldots x_n$. We denote the input space $\mathbb{B}^n$ by X and the state-space of C, that is, the set of possible values in the output ports of all its gates, as $Y = \mathbb{B}^{n-1}$. The synthesized circuit C can be viewed as a memoryless transducer from an input vector sequence X^* to an internal state sequence Y^* such that for every t, $y[t]$ is the stable state of the circuit after processing $x[t]$. The amount of switching in C relative to input x and at time t is

$$S(C, x, t) = \Delta(y[t-1], y[t]) \tag{3.5}$$

where Δ is the Hamming distance between Boolean vectors. The total amount of switching while reading a sequence $x \in X^*$ is

$$S(C, x) = \sum_{t=1}^{|x|} S(C, x, t). \tag{3.6}$$

A circuit C is better than C' relative to x if $S(C, x) < S(C', x)$. We want to build circuits which are optimal or reasonable in this sense. A major issue is what to assume about the set of inputs used to evaluate $S(C, .)$. One can think of two approaches.

1. Assume some probability function P on X^*, or more precisely a family of probabilities $P_k : X^k \to [0, 1]$, defined for example via a Markov chain, and then attempt to optimize the expected number of switches per time step

$$S(C, P) = \lim_{k \to \infty} \sum_{x \in X^k} P_k(x) \cdot S(C, x)/k.$$

2. Use a long reference sequence $\underline{x}$ and evaluate C according to $S(C, \underline{x})$.

We will use a mixture of these two approaches. We optimize $S(C, \underline{x})$ for some training sequence $\underline{x}$ generated by a Markov chain and then evaluate the synthesized circuit according to the number of switches that occur while processing other sequences generated from the same chain.

3.4.1 Solution Space

The number of unlabeled binary trees with n leaves equals to the Catalan number C_{n-1}[44].

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1} \tag{3.7}$$

For instance, there are 5 different trees for $n = 4$ as illustrated on Figure 3.5. Four of them are chains (extremely unbalanced tree) and the other a balanced tree (a tree with the minimal possible depth $\lceil log_2(n) \rceil$). Using chain topology can be in principle very beneficial for switching reduction. On Figure 3.6 we show an example of an input pattern for which the chain structure leads to lower switching activity compared to a balanced tree. However, using such structures in practice can lead to higher delay and to increased glitching, therefore we are primarily interested in balanced trees. The number of balanced trees with n leaves is

$$\binom{2^{\lceil log_2(n) \rceil - 1}}{n - 2^{\lceil log_2(n) \rceil - 1}} \tag{3.8}$$

The leaf labeling is also important to determine switching in a circuit as illustrated in Figure 3.7, where reordering the inputs leads to 0 switching for a given input pattern. There are $n!$ possible labelings, so the actual number of circuits that implement f is

$$(n-1)! \cdot \binom{2(n-1)}{n-1} \tag{3.9}$$

and the number of balanced circuits is

$$n! \cdot \binom{2^{\lceil log_2(n) \rceil - 1}}{n - 2^{\lceil log_2(n) \rceil - 1}}. \tag{3.10}$$

The number of possible realizations grows extremely fast, limiting the scope of an enumerative approach very small cones. We develop two optimization algorithms. One is polynomial, heuristic based, but only approximative. The other is i an enumerative algorithm that uses symmetry reduction to avoid checking circuits which are equivalent with respect to switching. This allows us to deal efficiently with larger trees, however this approach is still exponential and therefore can be used only on cones of limited size.

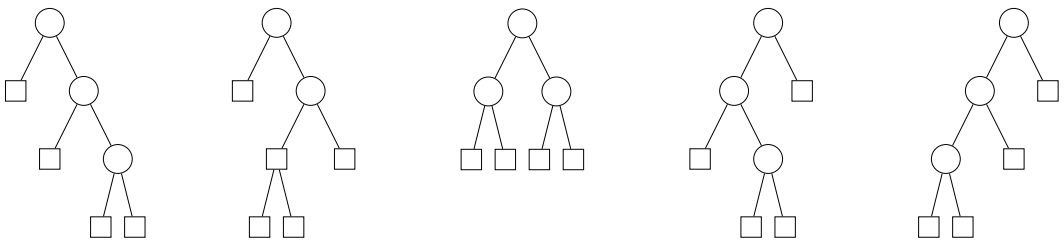


Figure 3.5: All possible binary trees with 4 leafs.

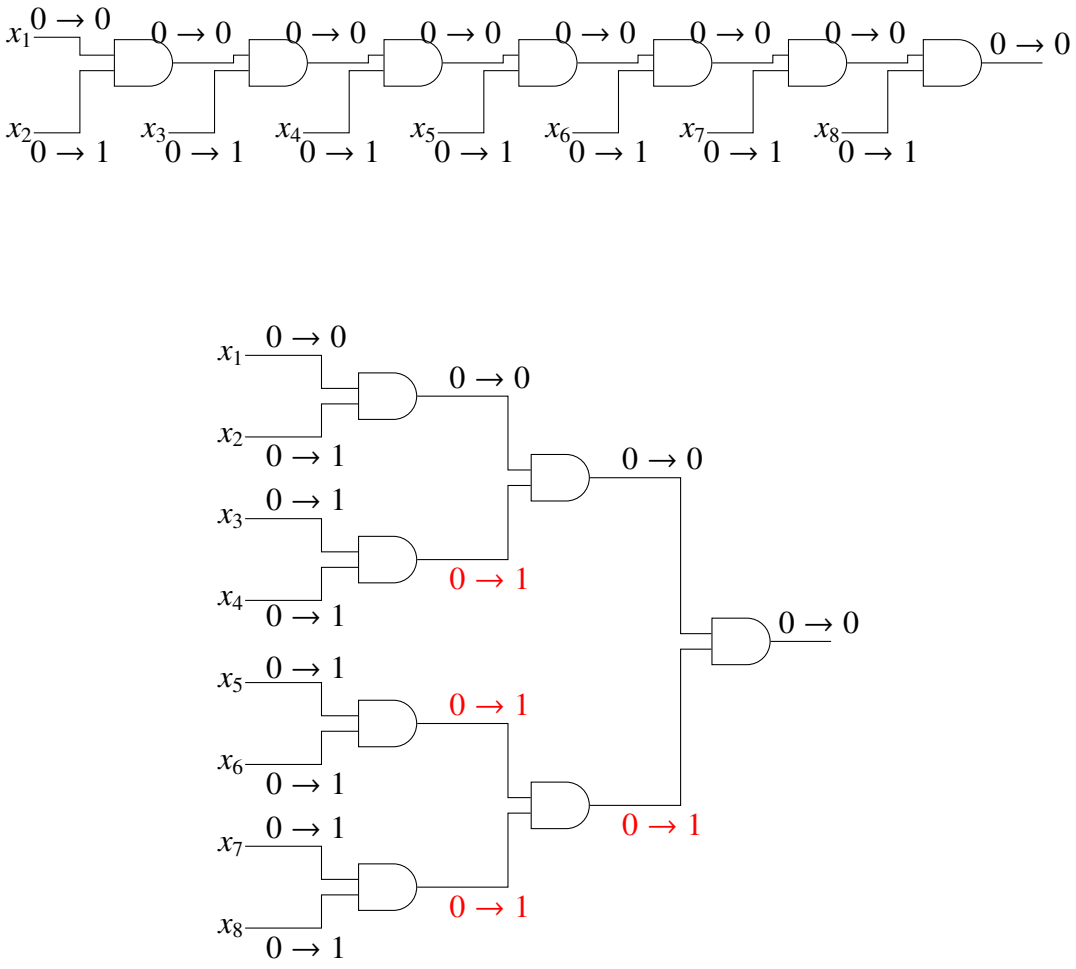
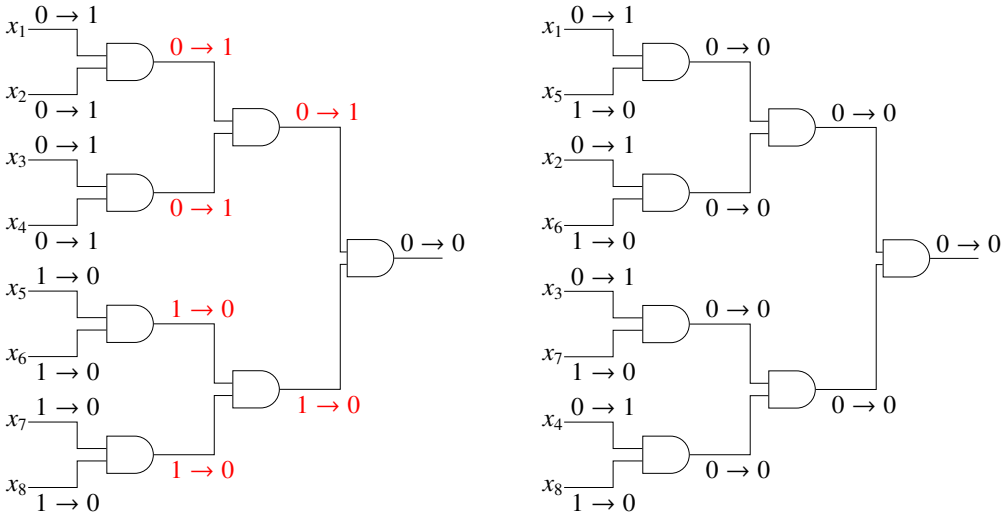


Figure 3.6: For an input transition $(0,0,0,0,0,0,0,0) \rightarrow (0,1,1,1,1,1,1,1)$ a chain realizations can abort all switchings but a tree cannot.



(a) (b)

Figure 3.7: Two pairings for input transition $(0,1,0,1,0,1,0,1) \rightarrow (1,0,1,0,1,0,1,0)$: (a) a bad pairing with 6 switchings; (b) a good pairing with no switchings.

3.5 A Level-Greedy Approach

To explore the effect of input ordering on balanced trees it is beneficial to restrict ourselves first to functions $f : (x_1, \ldots, x_n) \mapsto x_1 \wedge \cdots \wedge x_n$ where n is a power of two. In this case there is always only one shape of the balanced tree implementing the function and the solutions differ only in input mapping. This allows us to use a divide and conquer approach by partitioning the tree into *layers* and find the best input ordering for each layer separately. The first layer is composed of the gates directly driven by the inputs. The second layer contains gates driven by the outputs of the gates in the first layer and so on. The partitioning is illustrated on Figure 3.5.

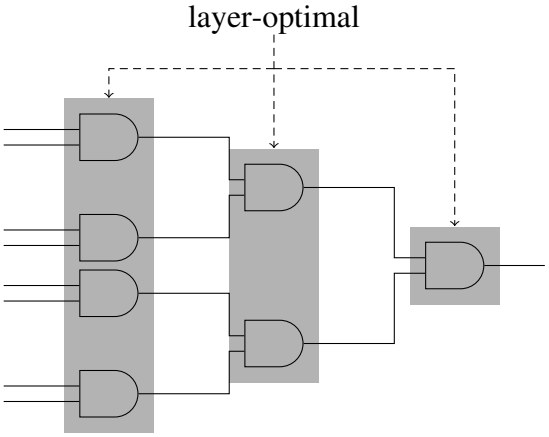


Figure 3.8: Solving the problem layer by layer

The problem is reduced to finding the best input mapping for one layer of 2AND gates. However, swapping two gates (along with its inputs) within the layer will not have any effect on the functions in the gates and hence switching. The only operation that can in principle lead to a change in overall switching activity in a layer is swapping the inputs of different gates, changing the functions realized by the affected gates. In fact this is essentially the problem of assigning two inputs to each gate or, viewed differently, the problem of pairing the inputs. Two paired inputs are then combined by a 2AND gate. We want to find such pairing that induces the minimal switching among all the possible pairings.

Now we will describe the level greedy algorithm in detail. We need to solve the problem of mapping input variables to the circuit input ports. The problem can be phrased recursively as follows. At level i of the tree, 2^{d-i} inputs should be partitioned into pairs to be mapped into 2^{d-i-1} 2AND gates. To understand which input signals should be paired together, let us look at Table 3.1-(A) which shows which transitions are taken by the output as a function of the transitions taken by the inputs. Table 3.1-(B) shows the number of output switches in each case while Table 3.1-(C) shows the net switching reduction effect, namely, the number of input switches minus the number of output switches. It is intuitively clear that for one consecutive pair of inputs, we should pair together variables taking respective transitions $1 \rightarrow 0$ and $0 \rightarrow 1$. Such transitions cancel each other and

	$0 \to 0$	$0 \to 1$	$1 \to 0$	$1 \to 1$
$0 \to 0$	$0 \to 0$	$0 \to 0$	$0 \to 0$	$0 \to 0$
$0 \to 1$	$0 \to 0$	$0 \to 1$	$0 \to 0$	$0 \to 1$
$1 \to 0$	$0 \to 0$	$0 \to 0$	$1 \to 0$	$1 \to 0$
$1 \to 1$	$0 \to 0$	$0 \to 1$	$1 \to 0$	$1 \to 1$

(A)

	$0 \to 0$	$0 \to 1$	$1 \to 0$	$1 \to 1$
$0 \to 0$	0	0	0	0
$0 \to 1$	0	1	0	1
$1 \to 0$	0	0	1	1
$1 \to 1$	0	1	1	0

(B)

	$0 \to 0$	$0 \to 1$	$1 \to 0$	$1 \to 1$
$0 \to 0$	0	1	1	0
$0 \to 1$	1	1	2	0
$1 \to 0$	1	2	1	0
$1 \to 1$	0	0	0	0

(C)

Table 3.1: (A) The output transitions of an AND-gate as a function of the input transitions; (B) The number of switches associated with every pair $(u \to u', v \to v')$ of input transitions; (C) The net switching reduction: number of input switches minus output switching.

send as inputs to the next level a variable doing $0 \to 0$ which will not trigger further switching with any other input it will be paired with. Fig. 3.7 shows two circuits and their performance differences with respect to a single consecutive pair of input vectors.

Let $R_{jk}(u, u', v, v')$ be the probability that a pair (x_j, x_k) of input variables takes the joint transition $(u \to u', v \to v')$. Given a reference input sequence $\underline{x}$, we can approximate $R_{jk}(u, u', v, v')$ by computing the number of occurrences of the given transition in the sequence. Denoting the number of switches of a hypothetical AND gate $x_j \wedge x_k$ implied by a pair of transition $x_j : u \to u'$ and $x_k : v \to v'$ (Table 3.1-(B)) by $s(u, u', v, v')$, which is always either 0 or 1, the expected number of switches in $x_j \wedge x_k$ is

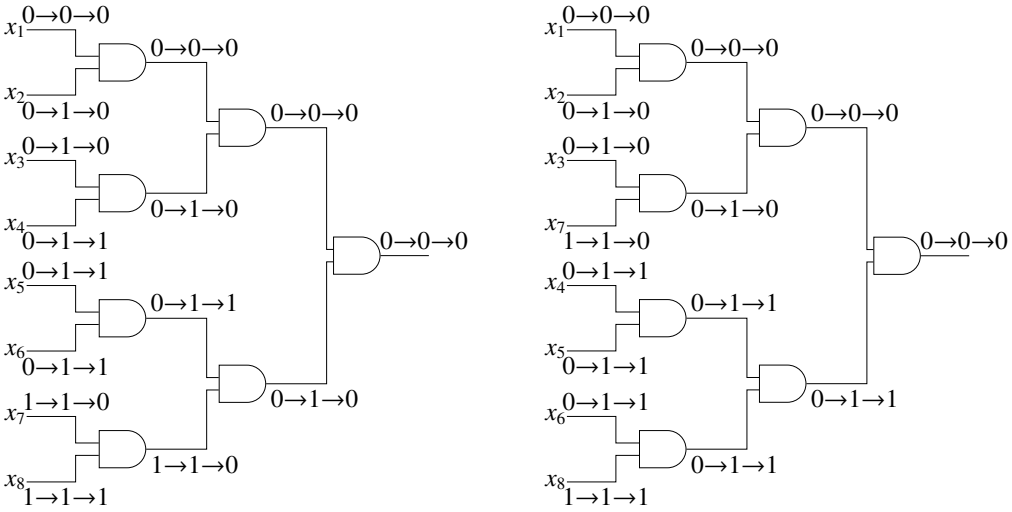
$$\mu_{jk} = \sum_{u,u',v,v'} R_{jk}(u, u', v, v') \cdot s(u, u', v, v'). \tag{3.11}$$

Let $G = (V, E, \mu)$ be a complete graph with n nodes where each edge (j, k) is labeled by μ_{jk}.

For the first level of the tree, the problem of finding input pairing which is optimal in terms of expected total number of switching is equivalent to the optimization problem known as *minimal-weight perfect matching* [57] for G. Once such an optimal pairing is found for level i, the outputs of the gates at this level serve as inputs for the pairing problem of the next level as summarized in Algorithm 1. The first polynomial algorithm for the optimal matching problem dates back to [22] using linear programming. The complexity of the algorithm has been improved in [48] from $O(n^4)$ to $O(n^3)$. Thus, together with the computation of μ from the training sequence the complexity of our procedure is $O(n^2 \cdot |\underline{x}| + n^3)$.

3.5.1 Examples of suboptimality

The result obtained by the level-greedy algorithm may deviate from the optimal expected number of switches in the AIG for two reasons. First, it is not based on real probability of switching but on its approximation from the training sequence. Secondly, the level-greedy algorithm, in principle, it is not guaranteed to produce the optimal among all circuits.



(a) (b)

Figure 3.9: A counter-example for the optimality of level-greedy algorithm: (a) A level-greedy pairing with 6 switches ; (b) An optimal pairing with 5 switches. Note that both pairings have the same (minimal) switching on the first level.

This is true even for circuits where the number of inputs n is a power of two as illustrated on the Figure 3.9. Additional issues are present when n is not a power of two. Applying the level-greedy approach when $n = 6$, we get three pairings at the first level, then we choose the best pair out of three outputs from the first level and the remaining two signals are paired to complete the circuit. The resulting topology of the circuit is shown in Figure 3.10, however, there is a different possible topology, illustrated in Figure

Algorithm 1: Level-greedy synthesis balanced-tree circuit for a conjunction of n variables.

procedure *Synthesize*($\underline{x}$)
Input: A sequence $\underline{x}$ of Boolean vectors of dimension $n = 2^d$
Output: A balanced-tree circuit C realizing $x_1 \wedge \cdots \wedge x_n$

$i := 0$
$F = \{(\{i\}, \emptyset)\}$ **while** $i < d - 1$ **do**
  $(F, \underline{x})$:=*Reduce*($\underline{x}, F, -i$)
  $i := i + 1$
end

function *Reduce*(x, i)
Input: A Boolean sequence x of dimension $m = 2^i$
Input: A forest of binary trees $F = T_1, \ldots, T_i$
Output: An optimal pairing of trees from T and a Boolean sequence y
of dimension 2^{i-1}

forall $j \neq k \in [1..i]$ compute μ_{jk}
let $G = (N, E, \mu)$ be the corresponding weighted graph
M :=*optimal_match*$(G) = \{(x_{r_1}, x_{r_2}), \ldots, (x_{r_{m-1}}, x_{r_m})\}$
$y := (x_{r_1} \wedge x_{r_2}, \ldots, x_{r_{m-1}} \wedge x_{r_m})$
$F' := join(T_{r_1}, T_{r_2}) \cdots join(T_{r_{m-1}, T_{r_m}})$
return(F', y)

function *join*(T_1, T_2)
Input: A binary tree $T_1 = (E_1, V_1)$ with root node r_1
Input: A binary tree $T_2 = (E_2, V_2)$ with root node r_2
Assume: $E_1 \cap E_2 = \emptyset$, $E_1 \cap E_2 = \emptyset$

Output: A new binary tree that has T_1 and T_2 as children of the root node.
$r := max(max(V_1), max(V_2)) + 1 \backslash\backslash$ r is a new root node
$V := V_1 \cup V_2 \cup \{r\}$
$E := E_1 \cup E_2 \cup \{(r_1, r), (r_2, r)\}$
return (V,E)

3.11. In this case we pair only four signals at the first level and two inputs are kept to be considered independently at the second level. Such an arrangement is not explored by the level-greedy algorithm. For a concrete example of an input sequence for which the circuit topology omitted by the greedy algorithm can lead to better switching, see Figure 3.5.1.

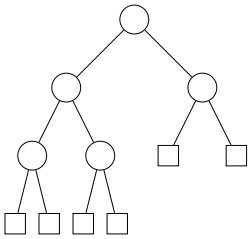


Figure 3.10: The topology resulting from a level-greedy algorithm.

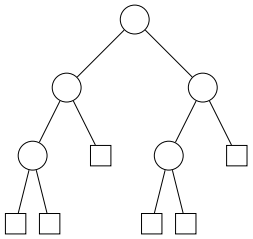


Figure 3.11: A topology not considered by a level-greedy algorithm.

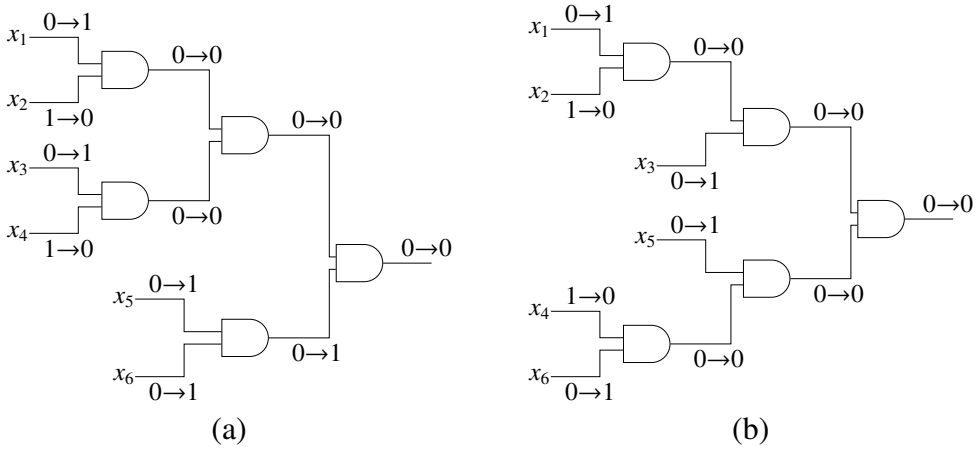


Figure 3.12: (a) A level-greedy pairing with 1 switching; (b) An optimal pairing with 0 switches using a topology not explored by a level-greedy algorithm.

3.6 An Enumerative Approach

The sub-optimality of the level-greedy approach brings us to the problem of constructing an algorithm that always finds the optimum. The straightforward brute-force approach of exploring every possible tree is very redundant as many trees are symmetrical and have exactly the same switching characteristics. In this section we explore how this symmetry can be used to reduce the overwhelming complexity of the enumerative algorithm and allow us to find optimal solutions for somewhat larger AND cones. We also show that the resulting algorithm is optimal. Finally we show how to adjust the algorithm to consider only balanced realizations.

Each AIG AND cone circuit implementing a function

$$f : (x_1, \ldots, x_n) \mapsto x_1 \wedge \cdots \wedge x_n$$

is essentially a binary tree where internal nodes represent 2AND gates and the leaves represent inputs. The output function of every node of the tree is of the form

$$g : (x_{r_1}, \ldots, x_{r_m}) \mapsto x_{r_1} \wedge \cdots \wedge x_{r_m},$$

where $x_{r_1}, \ldots x_{r_m}$ are the inputs labeling the leaves from which the given node can be reached. We can label every node with the set of variables that are in its fanin cone in the following way:

Consider a directed graph $G = (V, E)$ that represents a circuit implementing the function f. We assume that the variables $x_1, \ldots, x_n$ are the leaf nodes (without incoming edges). As this is a binary tree, we know that there is exactly $n - 1$ internal nodes that represent the AND gates:

$$V = \{y_1, \ldots, y_{n-1}, x_1, \ldots, x_n\}$$

Now, we can label each node with the set of indices of variables that influence its function:

$$vars(x_i) = \{i\} \tag{3.12}$$

$$vars(y_i) = vars(l(y_i)) \cup vars(r(y_i)) \tag{3.13}$$

where functions l, r return the children of internal node y_i, i.e., $(l(y_i), y_i), (r(y_i), y_i)$. We know that there are exactly two such nodes.

This allows us to precisely express the switching in the AND nodes of the circuit modeled as a graph $G = (V, E)$ under a given input vector sequence $x_1, \ldots, x_n \in \mathbb{B}^\tau$ as follows:

$$S(G) = \sum_{i=1}^{n-1} \alpha \left(\bigwedge_{i \in vars(y_i)} x_i \right) \tag{3.14}$$

We call $vars(z)$ a *signature* of a node z and the collection

$$\{vars(y_i) | i \in \{1 \ldots n-1\}\}$$

a *signature* of the circuit represented by (V, E).

The signature of a circuit provides full characterization of switching in the circuit and two circuits that have the same internal node functions must have the same switching. This means that in the enumerative algorithm it is enough to consider all possible signatures instead of all possible trees, which removes a lot of redundancy as one signature can be shared by multiple trees. For instance, if you rotate some subtree of a tree you get a different tree with the same signature. This means that the signature enumerating algorithm is optimal, a signature is a sufficient representation of the circuit structure with respect to combinatorial logic properties.

Besides optimality, we want also to show that our algorithm is not redundant, i.e., that it is not possible to omit any signatures and they have to be all checked. A signature would be redundant if there was another signature that would imply the same switching for all possible input sequences. The following theorem 3.1 shows that for every two signatures we are able to find some input vector sequence for which the switching in these two signatures will be different, hence no signature is redundant.

Theorem 3.1. *Let $A = \{A_1, \dots A_{n-1}\}$, $B = \{B_1, \dots B_{n-1}\}$ are two distinct signatures of 2*AND *tree circuits implementing $f : (x_1, \dots, x_n) \mapsto x_1 \wedge \cdots \wedge x_n$. Then we can binary sequences in $\mathbb{B}^\tau$ as values for $x_1, \dots, x_n$ such that the switching induced by this sequences in A differs from the switching induced in B:*

$$\sum_{i=1}^{n-1} \alpha\left(\bigwedge_{i\in A_i} x_i\right) \neq \sum_{i=1}^{n-1} \alpha\left(\bigwedge_{i\in B_i} x_i\right).$$

Proof. Let A' is the maximal (w.r.t) inclusion set in A such that $A' \notin B$. If such a set does not exist then $A = B$ hence it has to exist. There must be a set $A^c \in A$ such that $U = A' \cup A^c$ such that $U \in A \cap B$ and there must exist $B', B^c \in B$ such that $U = B' \cup B^c$. Seeing the signature as a tree, the U is the set of variables i the parent of A'. There must be always a parent as A' can not be root because root is the same in both A and B. A^c is a sibling of A'. The U is also present in B as otherwise A' would not be maximal. B' and B^c are the children of U in the context of B. Furthermore $A^c \notin B$ and $B', B^c \notin A$. This follows from the fact that both A', A^c and B', B^c are partitions of U. We can set

$$x_i = \begin{cases} 00 & \text{if } i \notin U \\ 01 & \text{if } i \in A' \\ 10 & \text{if } i \in A^c \end{cases}$$

Now

$$\sum_{i=1}^{n-1} \alpha\left(\bigwedge_{i\in A} x_i\right) = \sum_{\bar{A}\in A|\bar{A}\subsetneq U} \alpha\left(\bigwedge_{i\in\bar{A}} x_i\right) + \sum_{\bar{A}\in A|\bar{A}\subset A'} \alpha\left(\bigwedge_{i\in\bar{A}} x_i\right) + \sum_{\bar{A}\in A|\bar{A}\subset A^c} \alpha\left(\bigwedge_{i\in\bar{A}} x_i\right) + \alpha\left(\bigwedge_{i\in U} x_i\right)$$

and

$$\sum_{i=1}^{n-1} \alpha\left(\bigwedge_{i\in B} x_i\right) = \sum_{\bar{B}\in B|\bar{B}\subsetneq U} \alpha\left(\bigwedge_{i\in\bar{B}} x_i\right) + \sum_{\bar{B}\in B|\bar{B}\subset B'} \alpha\left(\bigwedge_{i\in\bar{B}} x_i\right) + \sum_{\bar{B}\in B|\bar{B}\subset B^c} \alpha\left(\bigwedge_{i\in\bar{B}} x_i\right) + \alpha\left(\bigwedge_{i\in U} x_i\right)$$

The input sequence was set in such a way that

$$\sum_{\bar{A}\in A|\bar{A}\subsetneq U} \alpha\left(\bigwedge_{i\in\bar{A}} x_i\right) + \alpha\left(\bigwedge_{i\in U} x_i\right) = 0 = \sum_{\bar{B}\in B|\bar{B}\subsetneq U} \alpha\left(\bigwedge_{i\in\bar{B}} x_i\right) + \alpha\left(\bigwedge_{i\in U} x_i\right)$$

and

$$\sum_{\bar{A}\in A|\bar{A}\subset A'} \alpha\left(\bigwedge_{i\in\bar{A}} x_i\right) = |\{\bar{A}\in A|\bar{A}\subset U\}| = |U| - 2.$$

Therefore also

$$\sum_{\bar{B}\in B|\bar{B}\subset B'} \alpha\left(\bigwedge_{i\in\bar{B}} x_i\right) = |\{\bar{B}\in B|\bar{B}\subset U\}| = |U| - 2.$$

The input is defined in such a way that A' contains always sequences of the form 01 and 10, therefore the value of their conjunctions is also 01 and 10, so there is a switch occurring in both A' and A^c. On the other hand the sequences of the form 01 are mixed with 10 in B' and B^c, therefore the value of the conjunction at these nodes is 00. Therefore Hence

$$S(A) = \sum_{i=1}^{n-1} \alpha\left(\bigwedge_{i\in A} x_i\right) = |U| - 2 + \alpha\left(\bigwedge_{i\in A'} x_i\right) + \alpha\left(\bigwedge_{i\in A^c} x_i\right) = |U|$$

$$S(B) = \sum_{i=1}^{n-1} \alpha\left(\bigwedge_{i\in B} x_i\right) = |U| - 2 + \alpha\left(\bigwedge_{i\in B'} x_i\right) + \alpha\left(\bigwedge_{i\in B^c} x_i\right) = |U| - 2$$

This means that switching in A is larger than in B. ⌟

3.6.1 Implementation

For a function

$$f : (x_1, \ldots, x_n) \mapsto x_1 \wedge \cdots \wedge x_n$$

we want to generate all canonical 2AND trees in the sense that one tree is generated per every possible signature. Furthermore, we want to represent canonical trees efficiently. A data structure representing a 2AND tree is a tuple with 4 data members:

- *size* – the number of inputs (leaves)
- *id* – a number serving as a unique identifier for the tree among the trees of the same size
- *lchild* – id of the left subtree, empty for leaves
- *rchild* – id of the right subtree, empty for leaves

- *lvars* – a set of indexes of variables that are driving the inputs in the left subtree assuming that the variables are indexed $1 \ldots size$. If the actual variables have different indexes, we use their relative order as index. For instance if x_2, x_5, x_7 are the variables driving this tree and x_2, x_7 are in the left sub-tree, we set $lvars = \{1, 3\}$. As the first variable goes always to the left sub-tree (explained later), we can further optimize this data structure by omitting 1 from *lvars*.

Now, we can create a list of all canonical trees of given size n. The canonicity is enforced by putting the smallest variable index always in the left subtree thus having only one tree for any given signature.

There is only one canonical tree for the trivial case of $f : (x_1) \mapsto x_1$. For $n > 1$, we build the list recursively (Algorithm 2). The actual enumerative power-aware synthesis algorithm (Algorithm 4) goes through all the canonical trees of size n, evaluates the switching in each tree with respect to some provided input vector sequence (Algorithm 3) and returns the tree with minimal overall switching.

Algorithm 2: Computes the list of canonical trees of size n and saves the result into a global variable *can_trees*.

```
procedure InitCanTrees(n)
Input Integer n > 0.
global can_trees\\ a list of lists.
if n == 1 do
  can_trees[n] := [(1, 0, Null, Null)]
done
else do
  for i ∈ 1...n − 1 do
    InitCanTrees(i)
  done
  can_trees := []
  for put_to_left ∈ {S ⊂ {1...n}|1 ∈ S} do
    lsize := put_to_left.size
    rsize := n − lsize
    for L ∈ can_trees[lsize] do
      for R ∈ can_trees[lsize] do
        tree := (n, can_trees.size(), L.id, R.id, put_to_left)
        can_trees.append(tree)
      done
    done
  done
end
```

Algorithm 3: Evaluating switching in a given tree under a given permutation and input pattern.

procedure *EvaluateSwitching*(tid, X)
Input: An input pattern $x_1 \dots x_n$ such that $x_i \in \mathbb{B}^\tau$ for each $1 \leq i \leq n$.
Represented as a list $X = [x_1 \dots x_n]$
Input: An id of a canonical tree t
Output: The amount of switching in the tree T driven by the input pattern $x_1 \dots x_n$

global *can_trees*\\ initialized by Algorithm 2.
if $n = 1$ **do**
  return 0
end
else do
  $T := can\_trees[n][tid]$
  for $i \in 0 \dots \tau$ **do**
    $top\_trace[t] := x_1[t] \wedge \dots \wedge x_n[t]$
  done
  $top\_switching := |\{t \mid top\_trace[t] \neq top\_trace[t+1]$ **for** $0 \leq i < \tau\}|$
  return $top\_switching +$
    $EvaluateSwitching(T.lchild, [X[i] \mid i \in T.lvars])+$
    $EvaluateSwitching(T.rchild, [X[i] \mid i \notin T.lvars, i \in \{1 \dots n\}])$
end

3.6.2 Balanced trees

As we are primarily interested in balanced circuit topologies, we need to be able to adjust Algorithm 4 in such a way that it considers only balanced trees. It is sufficient to modify the following line from Algorithm 2:

$$\textbf{for } put\_to\_left \in \{S \subset \{1 \dots n\} | 1 \in S\} \textbf{do} \tag{3.15}$$

This statement defines which variables will be placed in the left sub tree and which in the right. In the original algorithm we consider all possible partitions that puts the first relevant variable in the left trees. To keep the generated trees balanced, we need to add additional constraints that will ensure that there will not be too large disbalance between the number of variables put into the left and right tree. For this purpose we define an auxiliary function *powcap*, which takes a number n and returns the number of leaves in the largest possible balanced tree with the same depth as a balanced tree with n leaves. Note that balanced trees with the same number of leaves have the same depth.

$$powcap(n) = min(\{\, i \mid 2^i \geq n\})$$

For a balanced binary tree with n leaves and of depth d it follows $powcap(n) = d$ and

$$2^{powcap(n)-1} = 2^{powcap(n/2)} < n \leq 2^{powcap(n)}.$$

Algorithm 4: Synthesizing a balanced-tree circuit for a conjunction of n variables by brute force.

procedure *Synthesize*$((x_1 \dots x_n))$
Input: An input pattern $x_1 \dots x_n$ such that $x_i \in \mathbb{B}^\tau$ for each $1 \leq i \leq n$
Output: A graph representing a 2AND tree realizing $x_{p_1} \wedge \cdots \wedge x_{p_n}$ (AIG cone)

$opt\_tree := NULL$
$opt\_sw := \infty$
$InitCanTrees$
for $T \in can\_trees[n]$ **do**
  $sw := EvaluateSwitching(T, [x_1 \dots x_n])$
  if $sw < opt\_sw$ **do**
    $opt\_tree := T$
    $opt\_sw := sw$
  done
done

\\ Now we recreate a graph from T (Algorithm 5)
return $ToGraph(opt\_tree)$
end

Now consider a balanced binary tree T with n leaves ($n > 0$) of depth d. Let L be the number of leaves in the left subtree and R the number of leaves in the right subtree. We have $n = L + R$ and $powcap(n) = 2^d$. For a balanced tree, the left and right subtrees have to be also balanced, therefore

- $L \leq powcap(L) \leq powercap(n/2)$ as the depth of a subtree tree must be smaller than the depth of T.
- $L \geq powcap(n/4)$ as the depth of a subtree tree can not be smaller than $d - 2$.

Symmetrical relations hold also for R. Therefore as $L = n - R$, we have

- $L \leq n - powcap(n/4)$
- $L \geq n - powcap(n/2)$

Together, this gives the needed constraints on variable partitions and we can update line 3.15 as follows:

$$\begin{aligned} &\textbf{for } put\_to\_left \in \{S \subset \{1 \dots n\} \quad | \, 1 \in S, \\ &\qquad\qquad\qquad\qquad\qquad\quad | \, max(powcap(n/4), n - powcap(n/2)) \leq |S|, \\ &\qquad\qquad\qquad\qquad\qquad\quad | \, |S| \leq min(powcap(n/2), n - powcap(n/4))\} \\ &\textbf{do } \dots \end{aligned} \tag{3.16}$$

Algorithm 5: Reinterprets canonical tree representation into AIG cone.

procedure *ToGraph*(T)
Input: Canonical tree representation T
Output: An equivalent AIG cone represent as a graph (r, V, E),
where r is the root node, V the set of nodes and E the set of edges
procedure *ToGraph* $V := \{\}$\\ set of vertices
$E := \{\}$\\ set of edges
if $T.size == 1$ **do**:
  $V := \{1\}$
  $root := 1$
done
else do
  $(root_L, V_L, E_L) := ToGraph(can\_trees[T.lvars.size()][T.lchild])$
  $(root_R, V_R, E_R) := ToGraph(can\_trees[T.size - T.lvars.size()][T.rchild])$
  $lsz := |V_L|$
  \\ This is necessary, because V_L and V_R would otherwise not be disjoint
  $V := V_L \cup \{v + lsz | v \in V_R\}$
  $root := max(V) + 1$
  $V.insert(root)$
  $E := E_L \cup \{(s + lsz, d + lsz) | (s, d) \in E\} \cup \{(root_L, root), (root_r + lsz, root)\}$
done
return $(root, V, E)$
end

After this change Algorithm 2 generates all balanced canonical trees. We explained that the constraint will not eliminate balanced trees from consideration. We have to show that no tree that is not balanced will be placed into $can\_trees[n]$ by Algorithm 2. This can be proved by induction as follows.

Proof. For $n = 1$, $can\_trees[1]$ contains only one tree with only one node – it is hence balanced. For $n > 1$ we can assume by induction that for $i < n$, $can\_trees[i]$ contains only balanced trees. Therefore, if we put a tree T into $can\_trees[n]$, it follows that the left and right subtrees of T must be balanced. As T is not balanced, it means that there are two leaves in T such that the difference between the lengths of the paths form these leaves to root is at least two. As the immediate subtrees are balanced, this means that one of this two leaves must be located in the left subtree and the other in the right sub tree. Assume without loss of generality that the one located in the left subtree has shorter distance from the root, denoted d_L. Denote d_R the length of the path to the more distant leaf in the right subtree d_R we have $d_R \geq d_L + 2$. We can also assume without loss of generality that in the left subtree there is no other leaf closer to the root and in the right subtree there is no other leaf farther from the root, therefore the depth of the left subtree is not bigger than d as it is balanced and the depth of the right subtree is not smaller than $d + 1$. Therefore the

depth of the whole tree T must be at least $d + 2$, which gives us

$$powcap(n) \geq 2^{d+2}$$

Denoting L, R the number of leaves in the left subtree, we have

$$L \leq 2^d. \tag{3.17}$$

The constraint $L \geq powcap(n/4)$ implies

$$L \geq powcap(n/4) \geq 2^d$$

which together with Equation 3.17 gives us $L = 2^d$. This means that all the leaves in the left subtree have distance $d + 1$ from the root of T, which is a contradiction. Therefore T must be balanced. ∎

3.6.3 Complexity

The complexity of Algorithm 2 is given by the number of trees that have to be considered. The recursive initialization of *can_trees* for smaller arguments doesn't add any asymptotic complexity as every tree generated during this phase is accessed at least once during the main loop. Therefore the call $InitCanTrees(n)$ have asymptotic complexity $O(\mathcal{T}_n)$, where $\mathcal{T}_n$ is the number of distinct signatures for $n-$input AND function. $\mathcal{T}_n$ can be computed through the following recursive relation:

$$\mathcal{T}_n = \begin{cases} 1 & \text{for n=1} \\ \sum_{L=1}^{n-1} \binom{n-1}{L-1} \mathcal{T}_L \mathcal{T}_{n-L} & \text{otherwise.} \end{cases} \tag{3.18}$$

This definition follows directly from the structure of Algorithm 2. We need choose always $L-1$ variables out of $n-1$ to go to the left subtree (first variable goes there automatically). We consider all possible values of L from 1 to $n-1$ as the right subtree cannot be empty. For each such set of variables we include in the result all combinations of trees from $can\_trees[L]$ and $can\_trees[n-L]$.

If we restrict ourselves to balanced trees, we need to include the additional constraints as follows

$$\mathcal{T}_n^B = \begin{cases} 1 & \text{for n=1} \\ \sum_{L=A_L}^{\Omega_L} \binom{n-1}{L-1} \mathcal{T}_L \mathcal{T}_{n-L} & \text{otherwise.} \end{cases} \tag{3.19}$$

where

$$A_L = max(powcap(n/4), n - powcap(n/2)),$$
$$\Omega_L = min(powcap(n/2), n - powcap(n/4)).$$

To show how fast these functions grow, we compare in Table 3.2 the number of all trees, which would have to be considered when implementing the enumerative algorithm

in	all trees	canonical	balanced	canonical & balanced
1	1	1	1	1
2	2	1	2	1
3	12	3	12	3
4	120	15	24	3
5	1680	105	480	30
6	3.0240e+04	945	4320	135
7	6.6528e+05	1.0395e+04	2.0160e+04	315
8	1.7297e+07	1.3514e+05	4.0320e+04	315
9	5.1892e+08	2.0270e+06	2.9030e+06	1.1340e+04
10	1.7643e+10	3.4459e+07	1.0161e+08	1.9845e+05
11	6.7044e+11	6.5473e+08	2.2353e+09	2.1830e+06
12	2.8159e+13	1.3749e+10	3.3530e+10	1.6372e+07
13	1.2953e+15	3.1623e+11	3.4871e+11	8.5135e+07
14	6.4765e+16	7.9059e+12	2.4410e+12	2.9797e+08
15	3.4973e+18	2.1346e+14	1.0461e+13	6.3851e+08
16	2.0284e+20	6.1903e+15	2.0923e+13	6.3851e+08

Table 3.2: How many combinations brute-force algorithm has to check

naively without any symmetry reduction, the number of canonical trees (signatures) that are checked by our approach and the number of the balanced trees. We can see that the numbers grow very fast, however the symmetry reduction clearly allows to process larger AND cones. If we assume, for instance, that the maximal number of trees that we can afford to check is 500, the symmetry reduction allows us to find optimal 2AND balanced decompositions for AND cones with up to 8 inputs, compared to only 5-input cones that could be handled without the symmetry reduction. In fact, in realistic designs the majority of AND cones are relatively small, so the enumerative algorithm can be efficiently used for most of them, while the few bigger cones can be processed by the (sub-optimal) greedy algorithm.

So far we analyzed complexity of *InitCanTrees*. The synthesis (Algorithm 4) goes through every tree generated by *InitCanTrees* and calls *EvaluateSwitching* (Algorithm 3) for each of them. *EvaluateSwitching* is linear with respect to the size of the tree and linear with respect to the length of the input vector sequence. Therefore the overall complexity of the enumerative algorithm for power-aware synthesis is $O(\mathcal{T}_n \cdot n \cdot \tau)$ for the unrestricted and $O(\mathcal{T}_n^B \cdot n \cdot \tau)$ for the balanced case when τ stands for the length of the input vector sequence. In practice, τ can be very long, so sometimes it may be useful to use just a random segment rather then the full input vector sequence to trade execution speed for lower accuracy.

3.7 AIG level Evaluation

To evaluate our algorithms on the AIG level, we follow two distinct approaches. In the first we use a model of 16-input AND cone and various synthetic input generators. We show that the savings in such a large AND cone can be significant and that the savings are higher when the probabilistic input model has a higher level of 'orderliness'. In the second approach we use two small designs to evaluate our algorithms in a more realistic setting. One design is our own implementation of a decoder within a very simple calculator and the second design is an open source circuit implementing the SPI protocol. For the decoder we used assumptions about the typical operation of the calculator to create a Markov chain model that generated the input sequence. In case of the SPI design we used a test bench that was contained in the design to generate a representative use case.

3.7.1 Synthetic Input Generators

We evaluate our method against different classes of probabilistic 16-dimensional input generators. In this evaluation we use the level-greedy algorithm. Note that the brute-force approach cannot cope with cones of size 16. To define probabilities over X^* using arbitrary Markov chains we need to handle transition matrices of size at least $2^n \times 2^n$. For large n even writing down such a matrix is infeasible, not to mention computing its steady state probability. As is common in domains such as probabilistic verification and performance analysis, we use a compositional model consisting of a network of sparsely-interacting probabilistic automata. A probabilistic automaton $\mathcal{A} = (Q, \Sigma, \delta)$ is an input-dependent Markov chain where every input letter $\sigma \in \Sigma$ induces a different transition matrix over state-space Q. The probabilistic transition function is thus of the form $\delta : Q \times \Sigma \times Q \to [0, 1]$ satisfying

$$\sum_{q' \in Q} \delta(q, \sigma, q') = 1$$

for every q and σ. A Markov chain can be viewed as a degenerate probabilistic automaton without an alphabet and a transition function of the form $\delta : Q \times Q \to [0, 1]$.

Let $N = \{1, \ldots, n\}$. A network of n interacting probabilistic automata is given as $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, h)$ where $\mathcal{A}_i = (Q_i, \Sigma_i, \delta_i)$ and $h : N \to 2^N$ is an *influence function* such that $h(i)$ is the set of the other automata (besides itself) whose states are observed by $\mathcal{A}_i$ and influence its transitions. In our network each automaton has a state-space encoded by one bit, $Q_i = \mathbb{B}$, and an input alphabet $\Sigma_i = \mathbb{B}^{|h(i)|}$ which is the state-space of the influencing automata. The composition of the automata yields a global Markov chain (Q, δ) with $Q = Q_1 \times \ldots Q_n = \mathbb{B}^n$. The local input letter read by automaton $\mathcal{A}_i$ in a global state q is the projection of q on the variables in $h(i)$ that we denote by $\pi_i(q)$. The transition function of the global Markov chain is defined as

$$\begin{gathered}\delta((q_1, \ldots, q_n), (q'_1, \ldots, q'_n)) \\ = \\ \delta_1(q_1, \pi_1(q), q'_1) \cdot \delta_2(q_2, \pi_2(q), q'_2) \cdots \delta_n(q_n, \pi_n(q), q'_n).\end{gathered}$$

The structure of $h(i)$ can be used to classify models according to variable interaction. When the maximum of $|h(i)|$ is small, the system admits a small description from which random sequences for training and evaluation can be generated.

For each class of models we draw model instances randomly and measure the reduction obtained by our algorithm with respect to inputs generated by the model. All model classes share a tuning parameter $\alpha \in [0, 1]$ intended to quantify the degree of regularity in the input sequences which can be exploited to come up with good input pairing. Whenever we need to fix a probability while defining a model instance, we draw it from I_α defined as

$$I_\alpha = \begin{cases} [0, \alpha] \cup [1 - \alpha, 1] & \text{when } \alpha \leq \frac{1}{2} \\ [\alpha - \frac{1}{2}, 1 - (\alpha - \frac{1}{2})] & \text{when } \alpha \geq \frac{1}{2} \end{cases}$$

The regularity in the inputs (and the potential effectiveness of our procedure) is monotone decreasing with α. When $\alpha = 0$ the probabilities are taken from $\{0, 1\}$ and the resulting model is deterministic. When $\alpha = 1/2$ the probabilities are drawn from the whole interval $[0, 1]$ and when $\alpha = 1$ all probabilities in the model instances are equal to $1/2$. In this case there is no regularity in the input, all sequences of states and transitions are uniformly distributed and no switching reduction is expected because any input pairing would be as good as another.

The whole experimental protocol is summarized in Algorithm 6. For each model class and value of α, we draw randomly a set $\{M_1, \ldots, M_{50}\}$ of model instances. For each instance M_i we generate a training sequence $\underline{x}_i$ of length 10000, apply our algorithm and synthesize an optimized circuit C_i. We generate an evaluation sequence x_i of length 10000 and let $\underline{S}_i$ be the number of switches it induces in C_i. Then we draw a set $\{C_{i1}, \ldots C_{i20}\}$ of arbitrary circuits, let S_i be the average number of switches induced by x_i in these circuits and let R_i be the relative improvement in $\underline{S}_i$ relative to S_i. Finally R is the average reduction over all model instances of the same class.

Independent Inputs

We start by evaluating the switching reduction for two simple cases where the input variables are independent of each other. The first is the case where the value of each x_i is drawn according to a stateless Bernoulli process with parameter a_i while in the second model each bit is generated by an independent Markov chain with parameters a_i and b_i. The respective transition matrices are:

$$\begin{pmatrix} a_i & 1 - a_i \\ a_i & 1 - a_i \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} a_i & 1 - a_i \\ 1 - b_i & b_i \end{pmatrix}$$

For these models μ_{jk} is computed analytically (see Table 3.3) without a training sequence. Fig. 3.13-(a) shows for these two model classes the average reduction obtained by our algorithm as a function of α. In both cases the reduction is around 70% when the system is close to deterministic and 30% when probabilities are taken from $[0, 1]$.

	$0 \to 0$	$0 \to 1$	$1 \to 0$	$1 \to 1$
$0 \to 0$	$(1-a_j)^2(1-a_k)^2$	$(1-a_j)^2a_k(1-a_k)$	$(1-a_j)^2$	$(1-a_j)^2a_k^2$
$0 \to 1$	$a_j(1-a_j)(1-a_k)^2$	$a_j(1-a_j)a_k(1-a_k)$	$a_j(1-a_j)a_k(1-a_k)$	$a_j(1-a_j)a_k^2$
$1 \to 0$	$a_j(1-a_j)(1-a_k)^2$	$a_j(1-a_j)a_k(1-a_k)$	$a_j(1-a_j)a_k(1-a_k)$	$a_j(1-a_j)a_k^2$
$1 \to 1$	$a_j^2(1-a_k)^2$	$a_j^2a_k(1-a_k)$	$a_j^2a_k(1-a_k)$	$a_j^2a_k^2$

(a)

	$0 \to 0$	$0 \to 1$
$0 \to 0$	$\frac{a_ja_k(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$\frac{a_j(1-a_k)(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$
$0 \to 1$	$\frac{(1-a_j)a_k(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$\frac{(1-a_j)(1-a_k)(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$
$1 \to 0$	$-\frac{(a_j-1)a_k(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$-\frac{(a_j-1)(1-a_k)(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$
$1 \to 1$	$-\frac{(a_j-1)a_kb_j(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$-\frac{(a_j-1)(1-a_k)b_j(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$
	$1 \to 0$	$1 \to 1$
$0 \to 0$	$-\frac{a_j(a_k-1)(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$-\frac{a_j(a_k-1)(1-b_j)b_k}{(a_j+b_j-2)(a_k+b_k-2)}$
$0 \to 1$	$\frac{(a_j-1)(a_k-1)(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$\frac{(a_j-1)(a_k-1)(1-b_j)b_k}{(a_j+b_j-2)(a_k+b_k-2)}$
$1 \to 0$	$\frac{(a_j-1)(a_k-1)b_j(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$\frac{(a_j-1)(a_k-1)b_jb_k}{(a_j+b_j-2)(a_k+b_k-2)}$
$1 \to 1$	$-\frac{(1-a_j)(a_k-1)(1-b_j)(1-b_k)}{(a_j+b_j-2)(a_k+b_k-2)}$	$-\frac{(1-a_j)(a_k-1)(1-b_j)b_k}{(a_j+b_j-2)(a_k+b_k-2)}$

(b)

Table 3.3: (a) The probabilities of transition pairs for two sequences generated by: (a) Bernoulli processes with parameters a_j and a_k; (b) independent Markov chains with parameters a_j, b_j and a_k, b_k.

Algorithm 6: Average switching reduction evaluation for a class of probabilistic input generators.

Input: A class of probabilistic input generators
Output: An estimation R of the average switching reduction obtained by our algorithm
for i := 1 to 50
  draw a model M_i
  generate a training sequence $\underline{x}_i$ of length 10000
  C_i :=*Synthesize*($\underline{x}_i$)
  generate an evaluation sequence x_i of length 10000
  $\underline{S}_i := S(C_i, x)$
  for j = 1 to 20
    draw a circuit C_{ij}
    $S_{ij} := S(C_{ij}, x)$
  end
  S_i :=*average*$_j$ S_{ij}
  $R_i := (S_i - \underline{S}_i)/S_i$
end
R :=*average*$_i$ R_i

Cascades

Next we explore the class of cascade structures where the automata are ordered and each automaton observes the state of some of its predecessors. A network is a cascade of depth k if $h(i) = \{i-k, \ldots, i-1\}$ and the number transition matrices for each automaton is 2^k. The results for cascades of depth 1 and 2 are plotted in Fig. 3.13-(b). For depth 1 the reduction ranges from 70% for close to deterministic inputs to 15% for $\alpha = 1/2$ while for depth 2 the range is from 50% to 10%.

Partitioned Variables

Next we applied our procedure to a network where the variables are partitioned into clusters of size 2 and 4 and each automaton observes only the states of the automata in its cluster. The results are plotted in Fig. 3.13-(c). For 2-clusters the range or reduction is between 65% for almost deterministic inputs and 15% for $\alpha = 0.5$, while for 4-clusters the corresponding reductions are less than 50% and 10%.

Arbitrary Sparse Network

In the last class of examples we consider arbitrary networks where each automaton observes the states of k randomly chosen other automata. Fig. 3.13-(d) shows the results obtained for $k = 2$ and 4. In the former case we obtain 45% for $\alpha = 0.05$ and around 5% for $\alpha = 0.5$, while for the latter we obtain the worst results: less than 10% for quasi-

α	Bern	iMar	casc1	casc2	part2	part4	spar2	spar4
0.05	0.115	0.110	0.117	0.102	0.118	0.060	0.093	0.020
0.10	0.106	0.104	0.105	0.076	0.091	0.041	0.075	0.017
0.15	0.095	0.097	0.089	0.060	0.081	0.037	0.057	0.015
0.20	0.093	0.091	0.079	0.050	0.074	0.029	0.047	0.013
0.25	0.084	0.088	0.066	0.041	0.061	0.023	0.040	0.011
0.30	0.084	0.081	0.063	0.032	0.055	0.019	0.032	0.009
0.35	0.071	0.071	0.048	0.029	0.049	0.016	0.027	0.008
0.40	0.065	0.067	0.040	0.022	0.043	0.013	0.023	0.007
0.45	0.063	0.061	0.037	0.021	0.036	0.012	0.021	0.006
0.50	0.054	0.057	0.036	0.019	0.031	0.011	0.018	0.005
0.55	0.040	0.044	0.026	0.013	0.024	0.008	0.014	0.004
0.60	0.031	0.031	0.018	0.010	0.017	0.006	0.010	0.002
0.65	0.023	0.024	0.013	0.007	0.013	0.004	0.006	0.002
0.70	0.016	0.017	0.009	0.005	0.009	0.002	0.004	0.001
0.75	0.010	0.011	0.006	0.003	0.005	0.001	0.003	0.001
0.80	0.007	0.007	0.003	0.002	0.003	0.000	0.001	0.000
0.85	0.003	0.003	0.001	0.000	0.001	0.000	0.000	0.000
0.90	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000
0.95	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table 3.4: The *absolute* reduction in number of switching per gate per time step for all the models.

deterministic inputs and less than 5% when probabilities are drawn anywhere in $[0, 1]$.

Table 3.4 shows the average number of *absolute* switching elimination per gate in one time step. Upon closer inspection we observe that the results become consistently worse as the number of variables observed by an automaton becomes larger, quite independently of the interaction pattern. This may be an artifact of the way we generate model instances. The reason is that when an automaton has several transition matrices, the values of an entry (u, v) in different matrices may be taken from opposite sides of I_α, cancel each other and render the behavior of the variables more random and less regular.

3.7.2 Evaluation on small circuits

For evaluation we use the following circuits where we apply our procedure to full AIG

Mini Instruction Decoder We consider a very simple hand-held calculator whose instructions are listed in Table 3.5. The instruction are encoded using 4 bits although 3 bits would suffice, to reflect the fact that in a real application often not all the possible input combinations are used.

We assume that the typical use of the calculator will be just to perform an operation (add, subtract, multiply, divide) on two numbers typed into a numeric keypad. More

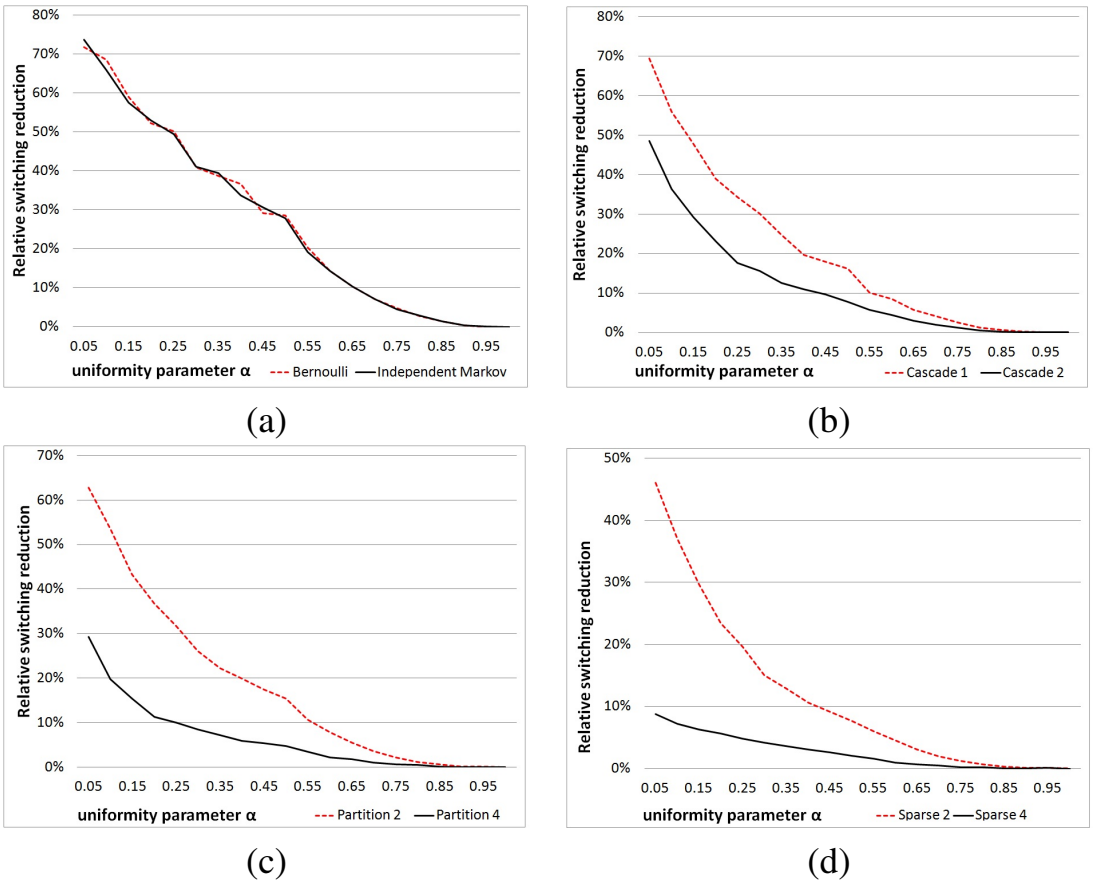


Figure 3.13: The average switching reduction as a function of the uniformity parameter α for different input models: (a): Independent inputs – Bernoulli (dashed red) and Markov processes. (b): Variables are arranged in a cascade structure of depth 1 (dashed red) and 2 (c): Variables are partitioned into mutually-dependent clusters of size 2 (dashed red) and 4 (d): Each variable depends on 2 (dashed red) and 4 other arbitrary variables.

instruction	code	meaning
LOAD	1001	loading from numerical keys
LOADM	1010	loading from memory
SET_ADD	1100	pressing '+'
SET_SUB	1101	pressing '−'
SET_MUL	1110	pressing '×'
SET_DIV	1111	pressing '÷'
EVAL	0000	pressing '='
STORE	0101	saving result to memory

Table 3.5: The instruction set of the calculator.

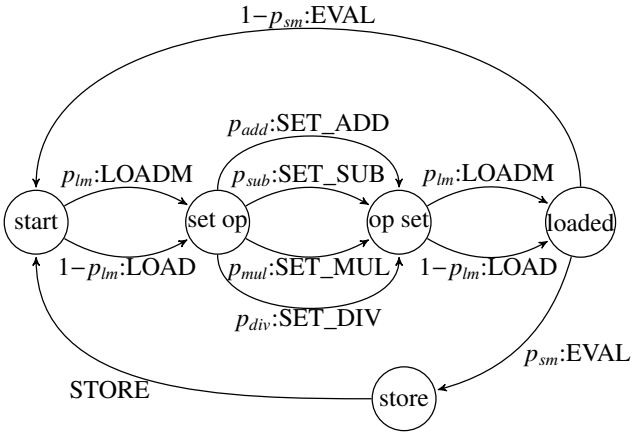


Figure 3.14: The probabilistic model of the instruction generator.

sophisticated users might perform more complex operations, say add three numbers at once, but with a lower probability. The Markov model for instruction sequences is depicted in Fig. 3.14 and explained below:

1. With probability p_{lm} load an argument previously stored in memory, otherwise just type in some number as the first argument.
2. Press one of $\{+, -, \times, \div\}$ with respective probabilities $\{p_{add}, p_{sub}, p_{mul}, p_{div}\}$.
3. Load the second argument either from memory (probability p_{lm}) or by typing the number.
4. Evaluate by pressing '=' and then with probability P_{sm} store the result in memory.

For the experiment we set the parameters of the model as follows:

$$p_{lm} = 0.1 \quad p_{add} = 0.4 \quad p_{sub} = 0.3$$
$$p_{mul} = 0.2 \quad p_{div} = 0.1 \quad p_{sm} = 0.1$$

Core SPI As a second example we use an open source implementation of a Serial Peripheral Interface, which is a synchronous communication interface specification available on Motorola's MC68HC11 family of CPUs [70, 33]. The project is available online at opencores.org.

We generated input patterns by simulating the test benches of the designs. The data was used to optimize the circuits on the AIG level. Table 3.6 compares the number of switching during the test bench execution in each design. The column *level-greedy* contains the switching when the circuits were optimized using the level-greedy algorithm. The *minimum* contains the minimal switching obtained by the enumerative algorithm. In the column *maximum* we used the enumerative approach to find the architecture with the

	maximum	minimum	level-greedy
Mini Instruction Decoder	250338	128118	158726
Core SPI (opencores)	20577	19681	19681

Table 3.6: Performance without preprocessing on small realistic circuits

	maximum	minimum	level-greedy
Mini Instruction Decoder	73976	72288	72288
Core SPI (opencores)	19555	19233	19233

Table 3.7: Performance with preprocessing on small realistic circuits

worst switching. The difference between minimum and maximum gives us some information about the size of potential optimization that we can achieve. In case of the Mini Instruction Decoder we achieve 49% switching reduction with the enumerative algorithm as compared to the worst case. The level-greedy approach gives slightly worse results with 37% of reduction. In the SPI case the results are considerably worse, achieving only 4% reduction for both algorithms, which is probably due to a relatively smaller number of large AND cones in the design.

3.7.3 Effect of the preprocessing

The ABC tool provides various algorithms that can be used to optimize logic networks by reducing the number of AIG nodes and logic levels, balancing the cones, merging functionally equivalent nodes and so forth. Such processing would certainly interfere with our reorganization of the AIG, wiping out all the savings if applied after our method. Still we can consider the case when the design is preprocessed before our method is applied in order to see if the standard optimization leaves some space for our algorithm to optimize. The preprocessing was done by subsequently applying the ABC optimization commands *refactor*, *rewrite*, *fraig* and *balance*. Table 3.7 compares the number of switching during the test bench execution in each design achieved by our algorithms when applied to designs that were preprocessed. We can see that the potential savings are significantly smaller (about 2% in both cases) as most of the switching was already removed by the preprocessing. The preprocessing also wipes out the difference between the enumerative and the level-greedy method.

	maximum		minimum		level-greedy	
	#	switching	#	switching	#	switching
All gates		179142		175242		192944
AND2X1	6	44460	6	28716	9	36496
INVX1	12	47180	14	58048	14	51608
NAND2X1	2	15744	4	18032	6	32052
NAND3X1	14	47266	15	46226	15	43472
NOR2X1	2	14900	3	15744	3	20840
NOR3X1	2	9592	2	8476	2	8476

Table 3.8: Performance without preprocessing on mapped Mini Instruction Decoder

	maximum		minimum		level-greedy	
	#	switching	#	switching	#	switching
All gates		106936		104132		104132
AND2X1	11	63812	11	66508	11	66508
INVX1	8	32960	8	32116	8	32116
NAND2X1	3	5236	0	0	0	0
NAND3X1	1	1116	4	5508	4	5508
NOR2X1	0	0	0	0	0	0
NOR3X1	1	3812	0	0	0	0

Table 3.9: Performance with preprocessing on mapped Mini Instruction Decoder

3.8 Technology level Evaluation

So far we considered only savings on the AIG abstraction level. However, what counts in the end are the savings on the physical level. To get a better estimate of this we used ABC to map the optimized AIG models to a 0.35 micron TSMC library. We tested the switching reduction with and without preprocessing. Tables 3.8, 3.9, 3.10 and 3.11 sum up the results with and without preprocessing on the Decoder and SPI designs. For each design we applied the enumerative algorithm to find the best and worst switching and we also used the level-greedy algorithm. Than we mapped the resulting AIGs to the gates provided by the library. For each type of gates present in the designs after mapping we show the number of the gates of this type and the switching in the gates of this type. However, we were not able to isolate some effects inherent to the mapping and AIG representation, therefore the results are not directly comparable. For instance in Table 3.10 the variant with minimal switching on AIG level has more mapped gates than the variant with maximum AIG level switching, therefore it is sub-optimal in space which makes the switching results incomparable. Additionally, different gate types have various physical properties and the power consumed by a gate during switching depends also on the gate's fanout. Therefore the results are inconclusive and the question of the effect of technology mapping on the AIG optimized by our method remains open.

	maximum		minimum		level-greedy	
	#	switching	#	switching	#	switching
All gates	177	21819	162	17530	162	17530
AND2X1	2	451	3	163	3	163
AOI21X1	4	580	4	580	4	580
INVX1	53	9699	54	9795	54	9795
MUX2X1	32	1648	48	1648	48	1648
NAND2X1	29	3118	40	3677	40	3677
NAND3X1	8	963	7	834	7	834
NOR2X1	32	4128	4	513	4	513
OAI21X1	16	1040	1	128	1	128
XNOR2X1	1	192	1	192	1	192

Table 3.10: Performance without preprocessing on mapped SPI

	maximum		minimum		level-greedy	
	#	switching	#	switching	#	switching
All gates	174	21438	175	20732	175	20732
AND2X1	2	451	1	2	1	2
AOI21X1	4	580	2	386	2	386
INVX1	53	9699	55	9637	55	9637
MUX2X1	32	1648	32	1648	32	1648
NAND2X1	25	2671	22	2157	22	2157
NAND3X1	8	963	12	1349	12	1349
NOR2X1	32	4128	32	4128	32	4128
OAI21X1	17	1106	18	1233	18	1233
XNOR2X1	1	192	1	192	1	192

Table 3.11: Performance with preprocessing on mapped SPI

3.9 Discussion

The interest in switching reduction and in the evaluation of circuit behavior against probabilistic models in general [31] is not new. Concerning switching reduction we can distinguish between an abstract approach like ours which focuses only on the number of transitions as an approximate indicator of power consumption and more physical approaches that map abstract circuits onto a concrete technology where power consumption can be measured more accurately. The work of [64] which belongs to the second category, mentions the abstract problem that we solve here as a suggestion for future work that could be plugged upstream to their own work on power-aware mapping using a real technology library. The work of [71] is also of this type, mapping abstract AIGs to real gates. The input is specified as a set of input vectors (patterns) and simulation with these patterns is used to estimate power consumption for different mappings alternatives onto real gates from a library.

The work of [67, 66] applies a similar reasoning concerning input pairing for 2and gates and uses a variant of Huffman's algorithm for constructing a binary tree with minimal average weighted path length [47]. However, this work is restricted to the case were variables are assumed to be generated by independent Bernoulli processes while our approach is applicable to any small-description Markov process or any user-provided training sequence. Moreover, they use a greedy pairing algorithm such that at each step of the algorithm one pair of variables, the one which induces the least expected number of switching is selected as an input to an and gate. Experiments show that our scheme which treats at once a complete level of the tree via optimal matching is significantly more efficient.

The work of [56] also uses Huffman's algorithm but in a different way that seems to yield a random balanced tree. They do not give any explicit probabilistic model but introduce some delay assumptions and claim their algorithm to be optimal in terms of reducing only the switching activity which is due to glitches. This is the place to mention that as we do not model gate delays, we cannot detect glitches but one may argue that their importance in balanced trees structures is less pronounced. The work of [66] is extended significantly in [72] who give an optimal algorithm for unbounded depth 2and synthesis, restricted to a Bernoulli input model. Their algorithm tends to produce deep circuits with long delays.

To summarize, we devised a novel procedure for an early step in the synthesis flow for digital circuits/functions. The major novelty of the algorithm is its ability to approximate in a tractable manner, polynomial in the number of inputs to an and gate, the minimal average-case number of switches, based on a training input sequence. The approach can be applied, in principle to any probabilistic model of the input but, of course, formal guarantees of approximation quality can be given only in restricted cases.

For synthetic empirical evaluation we developed an original framework based on sparsely interacting networks of probabilistic automata and ran extensive experiments under various probabilistic models of the input. The reduction obtained on these synthetic examples were quite impressive, reaching, in some cases, dozens of percents. Then we

explored the question of applicability of our method to real applications in terms of circuit structure and input model. The results on the AIG level seems encouraging, however the savings are minimal if other AIG optimization methods were used as preprocessing steps. Finally we did experiments in the direction of preliminary estimation of the impact of the technology mapping step, however the results have been so far inconclusive.

When you are stuck in a traffic jam with a Porsche, all you do is burn more gas in idle.

Steve Swartz

4

Sequential Power Reduction with Activity Triggers

Although power consumption can be measured and estimated precisely only after physical implementation it turned out to be important to have reliable approximative power analysis early in the design flow, at the register transfer level (RTL). Architectural decisions made at the RTL level can severely impact power consumption and detection of bad architectural choices at later stages might require an iteration back to RTL. In the current highly competitive and fast paced market any delay in production causes significant financial consequences for the manufacturer not only due to the price of additional development but often mainly due to the cost of losing the opportunity of marketing a device early before the competitors start offering similar or more advanced solutions. Therefore, in order to reduce the development time it is essential for the designers to be able to assess and optimize power related properties of the design as early in the design flow as possible, even if these are only approximations of the physical level behavior.

A prominent source of power dissipation is dynamic power, which is consumed when the transistors in the circuit change their states [17]. A significant part of this power is dissipated in the clock tree [9]. *Clock gating* is one of the most efficient techniques for reducing power dissipation. It is based on disabling the clock of design blocks when they do not perform any useful computation. This leads to direct power savings in the clock tree and, in some cases, in the block itself [40]. The implementation of this technique depends on *clock gating conditions*, which specify when an associated block can be safely deactivated.

Clock gating conditions can be identified at two abstraction levels. At the architectural (or conceptual) level, large functional units may be clock-gated using high-level control signals. For instance, the floating point unit of a processor can be clock-gated when the

currently processed instructions do not require such a computation. At the local level, registers can be clock-gated based on a local analysis of the circuit logic. This typically involves looking at the conditions under which the register data is being read (e.g., using select conditions), or propagating enables of upstream/downstream registers.

Local clock-gating has been shown to significantly reduce dynamic power consumption and is supported by several EDA tools that identify and implement clock-gating based on ODC/STC conditions, a technique which has shown some success [7]. However, a typical issue in this methodology is the complexity of the enable conditions at the local level. Sometimes these conditions are too complex and will be avoided by designers because it is unclear if the change will be safe at later design stages such as timing closure and routing. Another issue is the trade-off between the power savings achieved and the number of required changes in the RTL. Local conditions typically apply to a single bus or few flip-flops and complex (and hard to verify) conditions should be used to obtain significant savings.

Being able to provide simple and easily understandable power saving opportunities is thus an important requirement for power reduction tools. We claim that this can be achieved by detecting clock gating conditions at an *intermediate* level of abstraction, coarser than typical local clock gating methods but small enough to be difficult to spot by manual analysis. Typical targets for this type of conditions are medium-size functional units such as HDL modules with significant dynamic power. There are several advantages in using such clock gating conditions.

1. We can target simple and architecture-related conditions made of a few control signals which are cheap to implement in terms of added circuitry. They are more comprehensible to designers who can judge their correctness by themselves and make more confident decisions whether or not to use them. Ideal conditions are such that the designer himself could discover by a detailed manual analysis.

2. A single clock gating implementation can save much larger amount of power if it is used higher in the design hierarchy which furthermore enhances the complexity/saving trade-off compared to classical local methods.

Intermediate level clock gating closes a gap between conceptual and local clock gating. Clock gating at this level is very attractive, because of its potential to provide significant power savings with minimal changes to the circuit. Moreover, to the best of our knowledge, conditions for intermediate level clock gating are currently beyond the scope of what EDA tools can identify. It is an empirical question whether they are abundant, at least in some application domains, and how difficult it is to find them. The preliminary findings from our experiments are encouraging.

Main contributions

- We introduce a class of clock gating conditions called *activity triggers* that target intermediate size design blocks and which are typically related to architectural intent.

- We develop an algorithm which heuristically detects potential activity triggers

based on a statistical analysis of activity files (VCD or FSDB) generated by RTL simulation of the design. These potential triggers should be verified by a designer or using formal methods.

- We formalize the concept of activity triggers and their associated clock gating conditions. We define the temporal property corresponding to the fact that the trigger is correct and the clock gating based on the trigger is safe. We use model checking to formally verify the property.

- We propose and discuss a complete methodology where these techniques, that we have implemented within a commercial EDA tool, are used in an iterative semi-automatic fashion for finding activity triggers and efficient clock gating conditions. We demonstrate the methodology on an industrial video processing design, achieving a significant reduction of power.

Organization of the chapter

The rest of this chapter is organized as follows. Section 4.1 serves to summarize the concept of clock gating and to present some common methods of implementation. Section 4.2 presents related work on power reduction using clock gating techniques. Section 4.3 introduces the idea of *activity triggers* and illustrates them on a simple (but yet realistic) design. The concept of *activity triggers* is then more formally stated in Section 2.3.2. The core of our method – the statistical detection and formal verification of activity triggers is described in Sections 2.3.2 and 4.5. The complete proposed methodology is detailed in Section 4.6. In Section 4.7 we present the experimental results on the video processing case study. Finally we conclude in Section 4.8 and discuss potential extensions of our work.

4.1 Clock Gating Fundamentals

In this section we present the idea of clock gating and some well known techniques to determine the conditions under which clock gating can be used safely. Consider a simple sequential memory element (flip-flop) depicted in Figure 4.1. The flip-flop is synchronized by the clock signal *clk*. This means that the value of the input signal is propagated to the output always at the rising edge of the clock (when the value of the *clk* changes from logical zero to one).

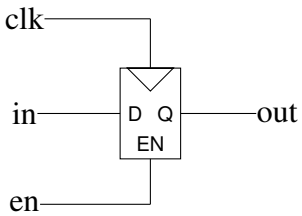


Figure 4.1: A simple sequential element

In case that the output value of the flip-flop is not needed at some point during the operation of the circuit, it is advantageous to disable the flip-flop so that it ignores the rising clock and is idle until re-enabled. The idea of disabling flops is based on the fact that the change of the flop state costs energy, therefore if we can switch off the flop when it is not needed, we can reduce the number of switches and save energy.

Disabling can be realized by a special enable input pin shown in Figure 4.1. Another way is to *gate the clock*. An idealized version of this approach is portrayed in Figure 4.1. When the enabling signal is 0, the clock going to the flip-flop is gated – set to 0, therefore as it is stable there will be no activity in the flip-flop. This view is idealized as in reality, using a simple AND gate for clock gating can lead to glitches in the flip-flop. This can be fixed by using a special clock gating cell with a slightly more complicated structure, however full explanation of these issues is unnecessary for understanding the principle of clock gating.

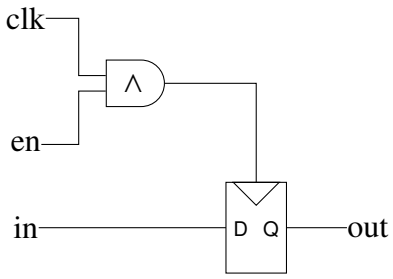


Figure 4.2: Implementation of clock gating (idealized)

In typical circuits there is a very large number of memory elements. The clock signal must be routed to all of them. This is realized by a *clock tree*, which is rooted at some global clock signal and whose branches lead to flip-flops. This allows us to apply clock gating higher in the clock tree, disabling a group of sequential elements at once by adding

only one clock gating cell to the design. Furthermore, the switching activity is reduced also in the clock tree structure itself which is beneficial as clock trees typically account for a significant part of dynamic power dissipation

A signal that can be used as enabler is called a *clock gating condition*. The obvious problem is how to compute clock gating conditions in correct and efficient way. In the rest of this Section we describe the principles of two commonly-used methods.

ODC (Output don't care)

ODCs [65, 4, 28, 20, 7] were originally introduced for application in logic synthesis. The clock can be gated if the outputs of a module are not observable.

For illustration consider the circuit in Figure 4.3. If $select = 0$, we know that in the next cycle the output of a sub-circuit X will be chosen by the multiplexer and therefore it is not important what value will be saved in the flop which feeds its value to the other input of the multiplexer. Hence we can enable this flop in a particular cycle if $select = 1$. The enabling condition is thus

$$en = select.$$

A naive implementation of this condition is shown in Figure 4.4

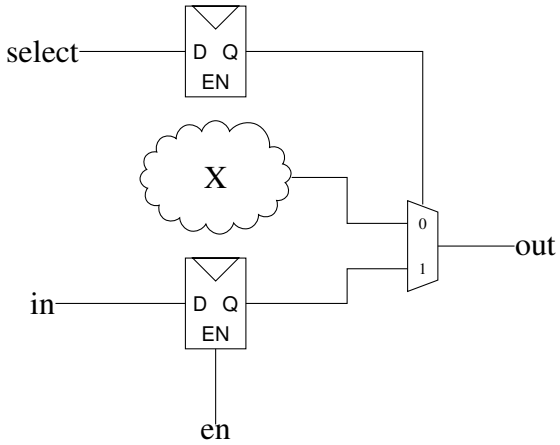


Figure 4.3: ODC example

STC (Stability Condition)

STCs[28, 7, 35] utilize the fact that if the next value of a memory element is equal to the current one, the register can be gated and the clock switching power saved. This is especially useful if an STC is valid for a larger block of registers, so that gating can be performed higher in the clock tree.

An example of a circuit from which we can derive STC is shown in Figure 4.5. We know that if the inputs to the combinatorial block X do not change, also the output of the block will stay stable. Therefore we can safely disable the flip-flop at the output. We know that a register output doesn't change if the register is disabled or if the input to the

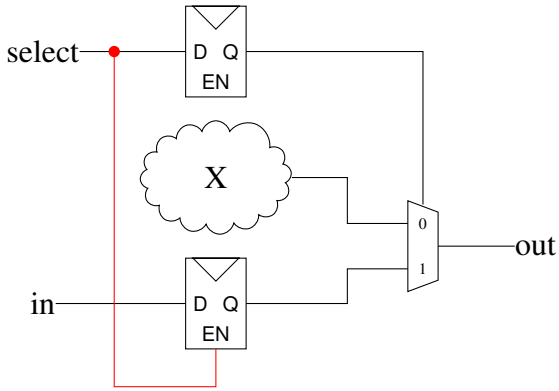


Figure 4.4: ODC example – with implemented enabling condition

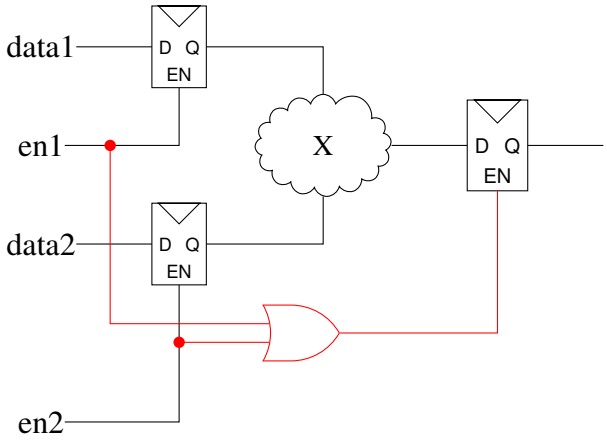


Figure 4.5: STC example – with implemented enabling condition

register is the same as the current output. Therefore the full enabling condition in the case of the last flop in the circuit of Figure 4.5 is

$$((D_1 = Q_1) \vee \neg EN_1) \wedge (D_2 = Q_2) \vee \neg EN_2)).$$

To implement such a complex condition in its entirety requires a relatively large amount of additional circuitry, so it would not be advantageous to use it in the circuit unless power reduction is very significant, for instance if we were disabling a thousand of registers instead of just one. In case that the cost of the enabling function is too high to be justified by the obtained power reduction, it is common to use a simpler condition which implies the optimal condition. In the case of Figure 4.5 we might use $EN_1 \vee EN_2$. The implementation is depicted in the figure.

4.2 Related work

There are multiple algorithms to compute clock gating conditions. In general, the approaches can be classified as *Observability don't care (ODC)* or *Stability condition (STC)* based (Section 4.1). *Activity triggers* are essentially STCs that are valid for relatively large design blocks.

Ideal clock gating conditions are often very complex and the cost of the additional clock gating circuitry would be higher than the savings. Therefore, approximate methods that compute weaker but simpler conditions were developed [3, 2, 4]. Our methodology is designed to compute simple conditions, hence approximation methods are not needed.

When designers are analyzing a circuit in order to find local clock gating conditions, some gating can be already present. This can be due to gating on the architectural level, or due to reusing parts of the design that are already gated. Some authors [28, 4] focused at strengthening such conditions that are already implemented in the circuit, strengthening conditions that were computed by other methods and combining them into more powerful conditions. Conditions discovered by our approach can be used as an input for these strengthening methods. Conversely, some partial clock gating in the analyzed design block could be used for simplifying the formal verification part of our flow, however this is not implemented in our tool. For detection, we ignore signals that are already used for clock gating, as our main goal is to find conditions that were not considered previously by a designer.

Hurst [35] introduces a guess-and-prove approach, which selects clock gating condition candidates from the existing nets in the design using heuristic based on timing and structural considerations. Candidates are then pruned using simulation and a formal proof is attempted for the remaining nets. Such conditions have the benefit of being already physically available in the design, therefore the added clock gating circuitry is very simple. A similar guess-and-prove approach was described in [24] where the candidates are not single nets but pairs of them, such that a simple invariant holds over them (a particular value of one signal implies a particular value of other signals). The invariant can be used for ODC based clock gating (the fanin of the implied signal can be disabled). The functional correctness has to be formally verified.

Another guess-and-prove method is described in [5] and [69]. This approach uses machine learning on simulation traces to infer clock gating conditions that are less complex and cheaper to implement than those coming from the traditional structural detection methods (e.g., [7]). In contrast with [35], simulation traces in [5, 69] are not used only to prune incorrect candidates, but also to collect the candidates from the positive examples in the simulation. The conditions are however very local (the analysis is done on the level of single registers). Outside of power reduction context, the idea of mining simulation traces for useful design intent related invariants was introduced in [25] for software and later in [32] for hardware.

Our approach is based on a guess-and-prove concept, the candidates are generated based on statistical analysis of a simulation trace, which yields a relatively small set of candidates for clock gating. Furthermore, we include the user in the loop, so that he or

she can interactively add constraints, which may be necessary for the formal check or he can verify candidates manually if the module is too large for a formal check, but the condition is simple and easy to understood. The main advantage compared to previous work, however, is that the conditions that we collect are usually coarse grained, simple and closely related to the design intent.

4.3 Activity triggers

Our approach is focused on clock gating conditions that are not necessarily optimal for a particular register but are simple to implement and shared by many registers; typically by an entire HDL module. These conditions are related to different modes in which a digital design operates. If a circuit operates in multiple different modes, there may be parts of the circuit that are used only for one of these modes. Thus, the goal is to find conditions that correspond to moving in and out of such modes.

To illustrate the concept we use a simple UART (Universal Asynchronous Receiver/-Transmitter) circuit. UART is a common digital design that can be used to interface fast electronic circuitry (from now on 'a computer') with slow peripheral devices. A device is connected to the UART by a serial line that allows to communicate data in a sequential fashion (one bit at a time). The communication follows a specific protocol. When there is no data transfer, the value on the serial line is set to logical 1. When the sender wants to transfer data, it first sets the serial line to 0 (start bit) for one time frame and then follows by transferring one byte of data, 1 bit per frame (8 frames). Subsequently, the protocol requires the sender to insert at least two frames with logical value 1 (stop bits). Then the sender can continue sending another byte (starting with start bit) or stays idle until another transmission is needed. Fig. 4.6 depicts a transmission of one byte via serial line.

The implementation of UART that we use is based on an open source design available at `http://opencores.org/project,osdvu`. It is a simple implementation that supports only the core functionality. The design (Fig. 4.7) is divided into two main modules.

The `RECEIVER` facilitates the communications sent from a peripheral device to a computer. It listens at the serial input line `rx`. It is idle until the peripheral sets the line to 0 announcing a start of a transmission. During the next 8 frames the UART samples value from the serial line in the middle of the time frame multiple times in order to minimize possible errors. The collected bits are placed in the output register `rx_byte`. After a whole byte is received, the availability of new data is signaled to the computer by raising flag `received` (which will typically generate an interruption in the computer). If the communication protocol is not followed (e.g. missing stop bits), the UART goes to an error state (signaled by raising flag `recv_error`). The UART recovers from the error by waiting for a long time (sufficient to perform multiple transmissions) and then returning to the initial state.

The `TRANSMITTER` can be used by a computer to send data to a peripheral device. The computer fills the input bus `tx_byte` with a byte of data that it wants to transmit. Then it sets `transmit` to 1. The UART transmits the byte to the serial line `tx`. The UART utilizes the flag `is_transmitting` to announce its current state: 0 signalizes to the computer that the UART is ready to transmit another byte.

Both the `TRANSMITTER` and the `RECEIVER` are divided into three main register subgroups, which can be implemented as HDL modules but do not have to, if the designer does not aim for a maximal modularization.

1. `CONTROL` is a unit, which controls the operation. It contains a register bus that represents the control `FSM` plus a few additional control registers.

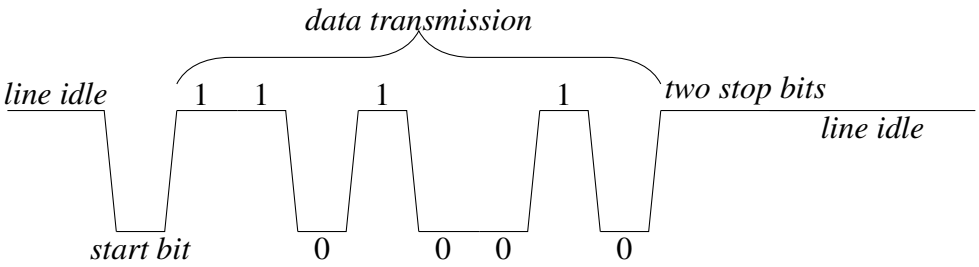


Figure 4.6: Serial line transmission of the character 'K' in the ACSCII encoding

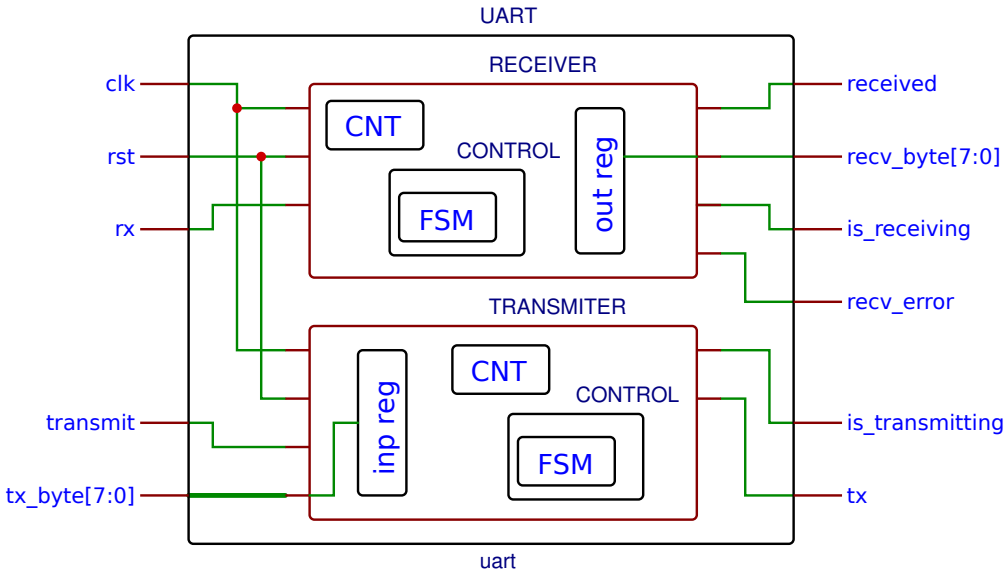


Figure 4.7: Schema of a simple UART design

2. `CNT` is a counter that is used to measure the time elapsed in various `FSM` states.

3. `inp_reg/out_reg` are register buses, which are used to store the received byte in the `RECEIVER` and the byte to be transmitted by the `TRANSMITTER`.

Consider the control automaton of the `TRANSMITTER` module in Figure 4.3. Every state corresponds to a mode of operation of the `TRANSMITTER` and not all submodules of the `TRANSMITTER` are used in every state:

IDLE: Initial state before the beginning of a transmission. The state is left when the `transmit` flag is raised. The input `tx_byte` is propagated into the input register together with changing the FSM state. All registers in the `TRANSMITTER` are stable in this state.
SENDING: The transmission is performed in this state. Some registers in `CONTROL` and `CNT` submodules are used extensively.
DELAY RESTART: The `TRANSMITTER` waits for a predefined time after the transmission is finished. `CNT` is used.

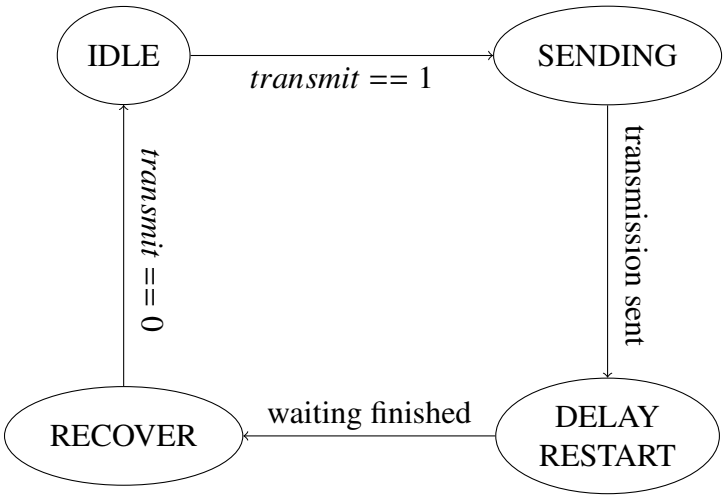


Figure 4.8: UART transmitter control FSM

RECOVER: The design waits in case the `transmit` flag was not set to 0. No registers are used beside those in the `FSM` upon leaving the state.

The registers that are not changing value in a particular mode can be clock gated. For instance, we can clock gate the entire transmitter when it is in the mode associated with the IDLE state of the `FSM`. However the state cannot be used as a clock gating condition directly, because clock gating would prevent the state of the `FSM` to be changed and the design would remain in IDLE state indefinitely. Instead, we identify the conditions associated with entering and leaving the state. Particularly, the condition associated with entering the IDLE state and starting the clock gating would be '`FSM==RECOVER && transmit==0`'. The condition for disabling the clock gating and leaving the `FSM` state would be simply '`transmit==1`'.

We can find similar conditions for every state in the `FSM` and use them for clock gating of registers that are not used in that state. Furthermore, some registers are used only in rare situations. For instance the value in the input registers can be changed only when taking the transition from IDLE to SENDING. Hence the clock enabling condition for this register group would be '`FSM==IDLE && transmit==0`' and the clock disabling condition would be '`FSM==SENDING`'. This way we can activate the input registers clock only when it is needed (one out of multiple thousands clock cycles).

The conditions that we identified are defined only on a few signals, so the additional clock gating logic will be simple and they are valid for relatively big design blocks (more than just one bus or a few registers), so they are more coarse than typical local clock gating conditions.

Ideally, such intermediate level coarser conditions should be identified by a designer. Then, if their validity is not obvious, our framework can be used to formally verify it. However, because of the extreme complexity of current designs and the reuse of old modules and purchased IPs, it is usually not possible for one person to have a detailed knowledge of all low level aspects of a design. For instance, if the UART was used in practice,

it would be probably just a small part of a much bigger system and the designer might reuse the UART from some old project. In such a case would probably regard it as a black box component and he would not have time to perform a detailed manual analysis of the internal implementation.

Hence, intermediate level clock gating conditions are often not recognized by designers and it is very useful to have an automatic tool to identify them. Furthermore, we observed that activity of some modules is not always triggered by a static condition like a state of a bus (representing an FSM state for instance) or a signal but often the activity triggering event corresponds rather to a transition. For instance, setting an instruction register to a particular value triggers the activity, which can continue for some time even after the value of the register is changed. This often occurs in cases in which some handshake mechanism is implemented.

We introduce the concept of *idle modes* and *activity triggers*. An idle mode of a sub-circuit is a mode in which it is safe to clock gate the sub-circuit. It will be associated with two events: a *stop* event, which forces the design to enter a idle mode, and a *start* event, which happens always before the exit from the idle mode. The combination of *stop* and *start* can be used for clock gating and is called an *activity trigger*.

It is possible that it takes some time for the module to enter the idle mode after an occurrence of a stop event. In this case, the module becomes stable only after a given number of cycles. This parameter is called *offset* and is also a part of the activity trigger.

4.4 Formal modeling and verification

We will use *linear time temporal logic* (LTL) with past operators (PLTL) [50, 51] to specify execution traces. This allows us to define the validity of activity triggers as properties that must hold for every execution of the circuit. Using past operators is more natural in our context than standard LTL as the clock gating conditions need to refer to the past behavior of a circuit.

Definition 7. A PLTL formula over a set of variables X is defined inductively as follows.

1. For $x \in X$, x is a PLTL formula.
2. Let Ψ and Φ be PLTL formulae. The following are also PLTL formulae:
   - $\neg\Phi$
   - $\Phi \wedge \Psi$
   - $\ominus\Phi$ (previously Φ)
   - $\Phi \mathcal{S} \Psi$ (Φ since Ψ)

The rest of the Boolean and temporal operators can be derived from $\neg, \wedge, \ominus, \mathcal{S}$. We also introduce a shortcut for a multiple application of the $\ominus$ operator, letting $\ominus^0\Phi = \Phi$ and $\ominus^i\Phi = \ominus\ominus^{i-1}\Phi$ when $i{>}0$.

Definition 8 (PLTL Semantics)**.** Satisfaction of a PLTL formula Φ by a trace σ at a time t, denoted by $(\sigma, t) \models \Phi$, is defined as follows

$$\begin{array}{lcll} (\sigma,t) \models x & \Leftrightarrow & x[t] & \text{(in the context of } \sigma) \\ (\sigma,t) \models \neg\Phi & \Leftrightarrow & (\sigma,t) \not\models \Phi & \\ (\sigma,t) \models \ominus\Phi & \Leftrightarrow & t{\neq}0 \text{ and } (\sigma,t{-}1) \models \Phi & \\ (\sigma,t) \models \Phi\mathcal{S}\Psi & \Leftrightarrow & \exists j, 0 \le j \le t \text{ such that } (\sigma,j) \models \Psi & \\ & & \forall i,\ j < i \le t \text{ such that } (\sigma,i) \models \Phi. & \end{array} \tag{4.1}$$

Satisfaction of a formula by an entire trace is defined as satisfaction (backwards) from its last state.

$$\sigma \models \Phi \Leftrightarrow (\sigma, |\sigma|) \models \Phi \tag{4.2}$$

We will use PLTL to define stability and other properties related to activity triggers.

Definition 9. Let $D = (X, Q, T, q_0)$ be a design and let $\sigma = q[0], \ldots, q[\tau]$ be an execution trace. The stability of a signal $x \in X$ is defined as follows

$$\mathit{stable}(x) = (x \wedge \ominus(x)) \vee (\neg x \wedge \ominus(\neg x)). \tag{4.3}$$

Furthermore, we can naturally extend the notion to define stability of an arbitrary set of signals $M \subseteq X$.

$$\mathit{stable}(M) = \bigwedge_{x \in M} \mathit{stable}(x). \tag{4.4}$$

This notion is important for clock gating because if M represents a set of sequential elements of the original circuit, these elements can be gated in clock cycles in which $stable(M)$ is satisfied.

We can now define activity triggering events that control the transition of sub-circuits into and out of idle modes where all their registers are stable. Such events can be, in principle, sequences specified by any PLTL formulae, but for statistical detection we restrict them to be transitions on a set of signals.

Definition 10 (Signal transitions)**.** Let σ be an execution trace and let $(x_1, \cdots, x_n)$ be an ordered set of signals. Let $b_1, \cdots, b_n, b'_1, \cdots, b'_n \in \mathbb{B}$. We say that $(x_1, \cdots, x_n)$ makes a transition $(b_1, \cdots, b_n) \rightarrow (b'_1, \cdots, b'_n)$ at time t if

$$(\sigma, t) \models \bigwedge_{i=0}^{n} ((x_i \leftrightarrow b'_i) \wedge \ominus(x_i \leftrightarrow b_i)).$$

An *activity trigger* for a module M consists of two events α and β such that α initiates the activity of M and β stops it. Typically, a module needs a few cycles to stabilize after the occurrence of β. The length of the stabilization period (*offset*) is denoted by d. When M is active, the occurrence of β should enforce M to become idle within d time steps unless it has been aborted by an occurrence of α. This condition, whose satisfaction initiates clock gating, is expressed in PLTL as:

$$\ominus^d \beta \wedge \bigwedge_{i=0}^{d} \ominus^i \neg\alpha.$$

Clock gating can be continued as long as the start event α has not been observed.

Definition 11 (Valid activity trigger)**.** Let α, β be signal transitions (or more generally any PLTL formulae) and d a positive integer. A triple (α, β, d) is a valid activity trigger for a module M if all traces of the circuit satisfy

$$\neg\alpha \mathcal{S} \left(\ominus^d \beta \wedge \bigwedge_{i=0}^{d} \ominus^i \neg\alpha \right) \Rightarrow stable(M), \tag{4.5}$$

Formal verification

A key feature of our methodology is that it can be formally verified whether a given candidate is a valid activity trigger for a module or a set of registers. To prove that an activity trigger (α, β, d) is indeed valid, we need to show that (4.5) holds for all possible behaviors of the system.

To implement this check using a circuit-oriented model checker we encode the condition $stable(M)$, which monitors the stability of a set of registers in the sense of (4.3) and (4.4), as an additional signal which is low if the value of some registers changed in the last clock cycle. Change detection for a simple memory element is realized by applying 'exclusive nor' (XNOR) to its input and output. For a more complex register, we

store the previous value in an auxiliary register and apply the XNOR to the two registers. Combining the results for all registers we obtain the required signal.

Using this new signal we construct an observer automaton (Fig. 4.9) which corresponds to the temporal formula (4.5). When this automaton is composed with the circuit automaton, it will enter the property violation state and only if (4.5) is violated. Thus validity of candidate activity triggers amounts to non-reachability of the error state, which can be proved using any formal verification tool capable of proving safety properties. For the tools used in our implementation see Section 4.6.

If the activity triggers are valid, the monitor automaton can be used, in principle, to control clock gating, enabled exactly when it is MODULE IDLE state (Fig. 4.9). This application of the monitor is, however, possible only if the inputs α and β of the automaton are not affected by the clock gating itself.

Constraints

Often it is not possible to prove the validity of an activity trigger under every possible input, but it is still valid under all input scenarios that can occur in our system. For instance we assume by default that the reset signal is used only at the beginning of any possible execution to initialize the design. A non-default assumption can, for instance, fix the value of some configuration registers or force the input signal values to follow some protocol. The UART's communication protocol can be an example of a complex input assumption. Another assumption for UART is that the clock driving the input is much slower than the clock driving the UART. Most of modern electronic designs are configurable and are used as a part of a bigger system that implies constraints on the input, therefore it is important to have a mechanism that allows to specify such constraints so that only constraint-satisfying behavior is considered by the reachability analysis. In our tool, we employ user-defined constraints during the formal check.

It is typical that industrial designs do not contain all the constraints necessary for a successful proof of an otherwise valid trigger. For such cases we propose an *iterative constraint refinement* procedure – when the trigger is disproved, the designer can manually examine the provided counter-example. In case the counter-example does represent a behavior that cannot occur in the circuit, the designer can add a constraint and run the formal check again. This can be repeated until we find a realistic counter-example or until the trigger is proven.

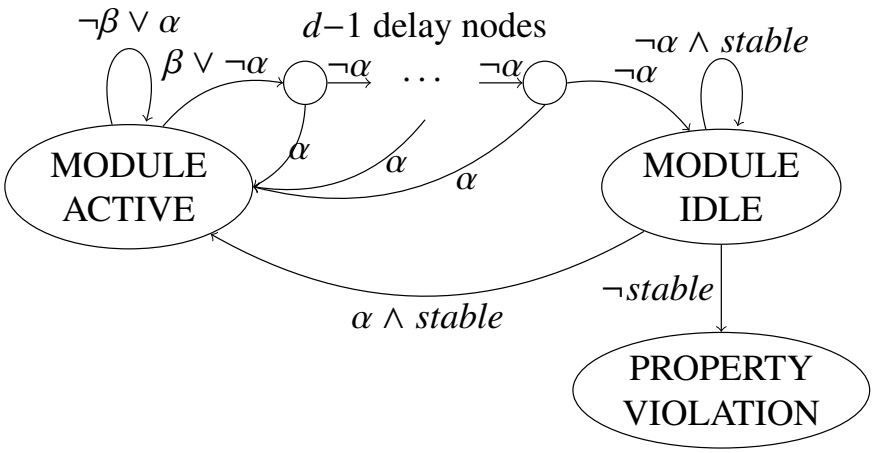


Figure 4.9: An automaton for checking validity of activity triggers.

4.5 Statistical trigger detection

Power optimization at the architectural level is sometimes performed by design teams as they analyze the simulation to find potential optimizations (e.g., idle periods where the clock is still active) or power bugs in the RTL (e.g., cases where the idle state of a block is badly implemented and the block still has activity). As the design scale increases this methodology may get difficult, prone to error, and time consuming. Also the use of third party IPs for which little knowledge on the architectural properties is known may complicate the analysis. The methodology we propose starts with automatically analyzing the simulation file in a first step, in order to identify the design blocks of interest (i.e. with significant potential power savings) as well as to point out the interesting activity events and signals that trigger them. In a second step, we provide a formal flow that can verify clock gating conditions based on activity triggers. This may be done either on the triggers found during our automatic simulation analysis or on trigger conditions directly provided by the user. When the verification is successful, the clock gating condition is proved to be safe in the sense that a whole block may be clock-gated without disrupting the functional behavior of the design.

To detect activity triggers we use a heuristics-based statistical approach. The idea is based on a hypothesis that the activity triggering events can be found in a short time window before and after the idle periods in the simulation traces. The user performs an RTL simulation of the design, based on a set of test vectors that should represent standard behavior of the circuit. Such vectors are commonly used by designers, e.g., for functional verification or power estimation. The changes of signal values during the simulation are stored in a file (VCD or FSDB). Then our detection tool performs the following steps in order to find a set of events that have a good chance of being activity triggers.

Figure 4.10: Idle periods in a simulation

Design decomposition

First, we need to define the set of sub-circuits for which we want to analyze the activity. In general, a sub-circuit can be any set of registers chosen by some meaningful strategy. Typically, these are RTL modules or user-defined register groups. For simplicity, we refer to the chosen sub-circuits as *modules*.

Idle periods detection

For every module we analyze the activity. We identify all the idle periods – i.e., intervals in which no registers were switching. For such intervals, there will be a minimal length in order to exclude very short periods that may be noise.

Finding potential triggers

For each idle period, we look for signals having a transition during a short window before/after the period. The size of this window is a parameter to the procedure. For instance, if a signal goes $0 \rightarrow 1$ before every idle period, it is a good stop event candidate. We also analyze transitions of small buses. For instance if a bus always goes $0001 \rightarrow 0010$ after an idle period, it is a start signal candidate. This is especially useful for buses that hold an FSM state.

To distinguish between potential candidates, we introduce two statistical indicators for each transition, *coverage* and *noise*. The *coverage* is the percentage of the idle periods that may be affected by a transition (i.e., the transition is observed before the period). The noise is the percentage of transition occurrences outside of the window before/after the idle period (Fig. 4.11). Candidates with high coverage and low noise are considered to be strong candidates, even though not all of the true triggers need to have 100 % coverage (not every idle period has to be controlled by that trigger) and 0% noise (e.g., a start event happening within an active period is possible but has no effect). However, experiments show high/low coverage/noise to be a good indicator.

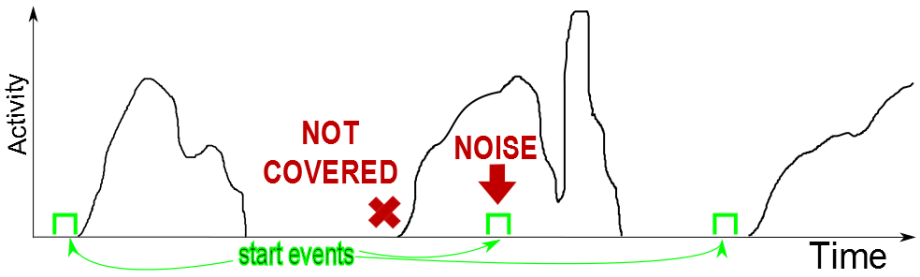


Figure 4.11: Coverage and noise

Filtering, ranking and reporting

We employ a number of heuristics to remove weak candidates or candidates that are strongly depending on each other.

- We filter out all transitions that do not have a good coverage and noise rankings.
- We perform a structural check to remove signals that are not in the fanin/fanout of the module. A start event related to a signal that cannot reach the module in its fanout cone is eliminated (as it cannot influence the module). Respectively, a stop

event related to a signal that does not have the module either in the fanin or fanout cone is eliminated (it cannot influence or be influenced by the module).

- We remove highly active signals (like clocks), if they were not already removed due to a high noise ranking.
- We compute shortest path from the signal to the module. This can be used to filter out candidates that cannot have a causal relation to the activity event.
- We eliminate the candidates, which are already used for clock gating.

To illustrate the detection we will use the UART `RECEIVER` module. On Fig. 4.12 we can see the combined activity of all nets in the receiver module during a test execution. It contains blocks of activity that represent receptions of one transmission block each. After the statistical analysis we find that the transition 000→001 of the `FSM` is reported as a possible start event and the transition 0→1 of the signal `received` is reported as a possible stop event. Both events have coverage 100% as the receiver's `FSM` is moving from 000 (wait for a new transmission) to 001 (start processing a new transmission) before every block of activity (transmission processing) in the activity graph (Fig. 4.12) and we can see the `received` flag being raised at the end of each activity period. Both events have noise 0% as neither of them occurs in any other place in this simulation trace.

The pair (`FSM` : 000→001, `received` : 0→1) forms a valid activity trigger (with an offset 1), which is then formally proved by the verification tool. However, the fact that the start signal is defined on the `FSM`, which is contained in the module, means that we cannot use it directly for clock gating of the entire module (as the `FSM` would be clock gated and the module would never be activated again). We can use it to correctly clock gate all the registers other than `FSM` though, which represent most of the module's power consumption.

Still, there exists an event which is a more desirable candidate for clock gating. The UART's communication protocol requires that the input serial line `rx` is set to 0 at the beginning of an incoming transmission. This event can be used to clock gate the whole receiver module, including the `FSM` and it is defined on one wire, hence the added clock gating circuitry will be smaller. However, `rx` is used not only to start the transition, but also to communicate the content of the transmission, hence it switches not only before the active periods but many times also within the active periods. Therefore the noise ranking of `rx` is very high and the event is filtered by the detection engine. An experienced designer can easily recognize that the reported `FSM` : 000→001 event is triggered by `rx` : 1→0 and use the latter for clock gating, however this phenomenon represents a room for improvement of the current detection heuristic that will be addressed in future research.

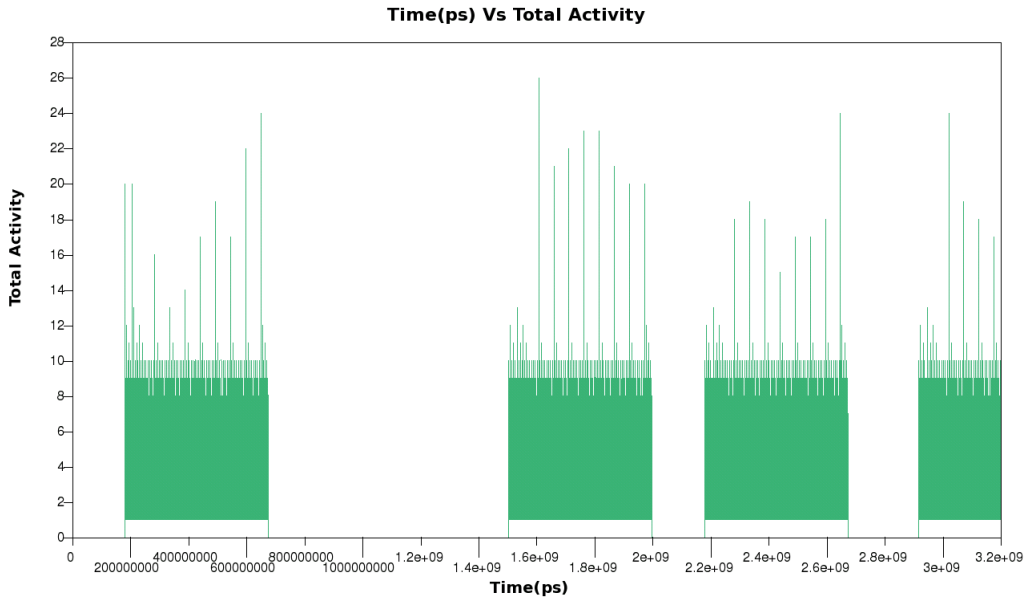


Figure 4.12: Activity of UART receiver module

4.6 Application flow

We provide a methodology and a tool for an extensive analysis of activity triggers in digital designs. In our flow, we expect the user to provide HDL files describing the design, the constraint definitions (clocks and resets definitions, input constraints, etc.), and simulation data (VCD, FSDB). Our tool provides two core functionalities:

Trigger detection

The input is the design HDL description and simulation data. The data is analyzed by our statistical engine to infer a set of events that seem to trigger activity. These triggers need to be verified by a designer or by the formal verification tool.

Formal verification tool

The input consists of an HDL description of a design, an activity trigger specification and a time budget for verification. The trigger specifications can be supplied manually or taken from the trigger detection engine. The tool uses ABC's [63, 14] PDR engine (property directed reachability [23, 13]) in attempt to prove the validity of the trigger, while running BMC (bounded model checking [11]) and Rarity Simulation [18] engines in parallel, which allows to disprove some incorrect triggers faster than only with PDR. The result of the formal check can be either "VALID" if the trigger was proven, "INVALID" if the verification engine found a counter-example, which is saved, or "TIMEOUT" in case that the time budget was exceeded.

We propose a semi-automatic flow for the designers to be able to exploit the clock gating opportunities maximally.

1. Run statistical trigger detection on trace files generated during simulation. The tool will create a list of possible activity triggers for every module where some promising candidate is identified according to the heuristics described in Section 4.5.

2. Run the formal check with a reasonable time budget for every detected candidate. The candidates that are proven at this stage can be used directly for clock gating.

3. For the candidates that are disproved or which were not proven within the allocated time, these that have a high potential power reduction (number of affected registers) should be selected for a manual examination. Sometimes it can be obvious that a trigger candidate is correct and the designer may use it directly based on his expert assessment.

4. Otherwise, the designer should examine the counter-example trace of the failing triggers. If he realizes that the trace corresponds to a behavior that would not be possible in a real execution of the circuit, he can specify input constraints as System Verilog Assertions (SVA [37]) and add them to the design files. This process of

adding constraints can be repeated multiple times. Constraints that fix some configuration inputs or registers are usually necessary for the formal check to be successful if the design is configurable.

5. In case the designer suspects that the activity is limited to just a part of a module, it is possible to run the verification for a specified group of registers only.

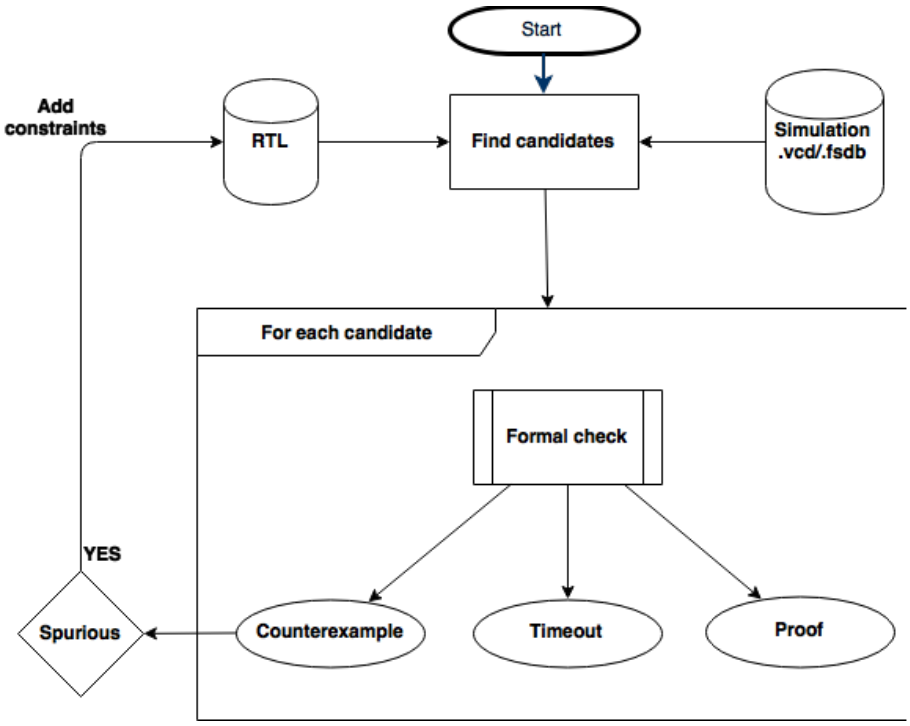


Figure 4.13: ATD semi-automatic flow

For reasons explained before, the detection method performs best when the design is highly modularized. For instance, in the original version of the UART example, the `receiver` and the `transmitter` functionalities are implemented in the same Verilog module. This leads to a strong interference between different scenarios, e.g., the active periods resulting from reception in UART are marked as 'not covered' when evaluated with respect to a trigger that is valid for the transmitter. Furthermore, it is not possible to prove validity of any trigger, without manually specifying the group of registers for which the trigger is valid (some triggers are valid for the `receiver`, some for the `transmitter`, but not for both). Highly modularized design, where `receiver`, `transmitter` and ideally also register groups as `FSM` and `CNT` allows the tool to detect and verify opportunities that would be otherwise missed due to activity interference.

4.7 Experimental results

In order to demonstrate the efficiency of our methodology we applied it to a real case study: a cluster for video processing called SENDS (Smooth ENgine for Data Stream, Figure 4.14).

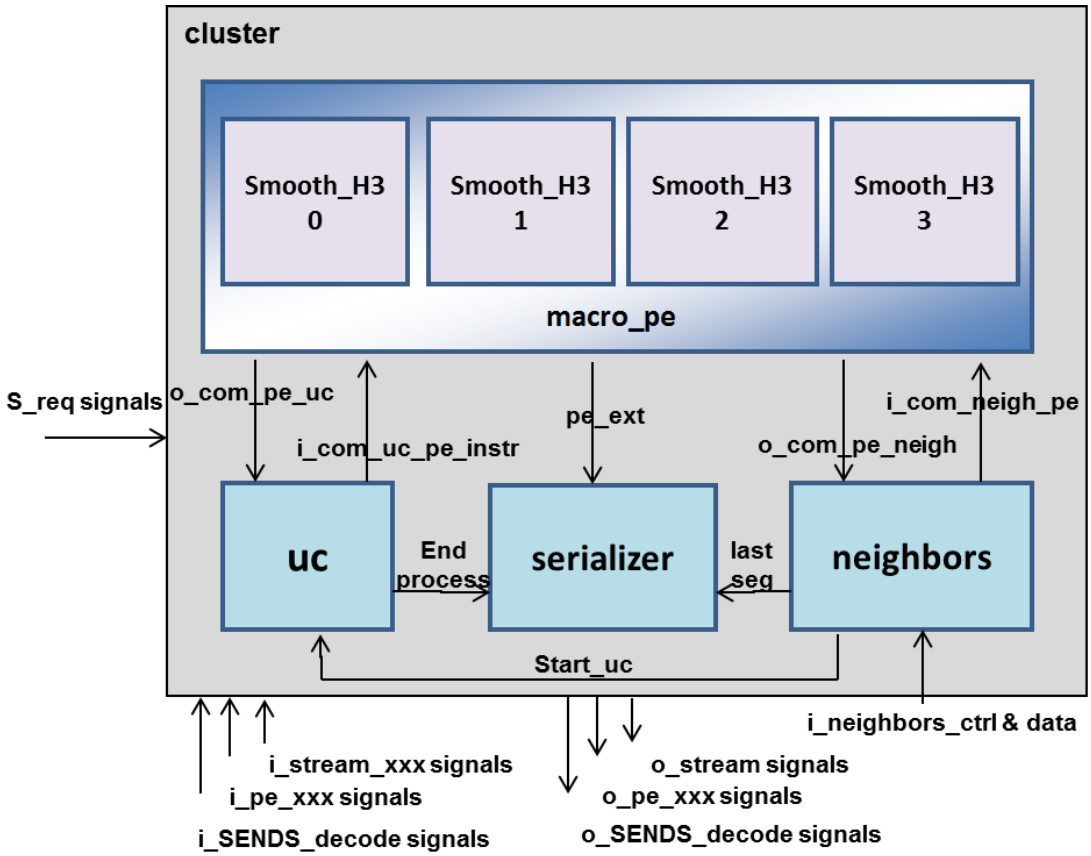


Figure 4.14: A video processing architecture

SENDS is part of a larger, clusterized and configurable architecture for video processing. This architecture is coupled with a CMOS image sensor integrated on the same chip, from which it takes a pixel stream as input. The circuit can be configured to perform (concurrently or sequentially) several low-level processes such as various filtering, contrast enhancement, de-noising or YUV to RGB conversion. These computations constitute a pre-processing step for more involved image processing such as complex features extraction. This kind of architecture is used, for instance, in smart cameras.

The SENDS design is organized around an FSM controlling the data path scheduling and reception/emission of the pixel stream. The data path can be configured. In this particular case it features smoothing engines, which operate concurrently. The IP is connected to its interface through a bus, and it has several inputs for configuration, control and synchronization, as well as data/control inputs related to the pixel stream. The data consists of columns of pixels each coming from the sensor. A pixel has 3 components:

Red, Green, and Blue, 8 bits each. The processing is performed on a pixel window, where the pixel in the center is computed based on neighboring pixels. The number of concurrently processed windows corresponds to the number of processing units available in the architecture.

The pixel stream is first collected in the neighbors module. When the neighbors register file has been filled with sufficient data, it notifies the unit control (uc) by raising the signal `start_uc`. Then the unit control sends appropriate instructions (in particular pixel addresses to be processed) to the `macro_pe` module. The `macro_pe` triggers the data retrieval in the neighbors register file and launches the smoothing process. When the computation is finished it notifies the uc which in turns notifies the serializer using signal `end_process`. The filtered pixels are then sent by the `macro_pe` to the serializer which outputs the result.

There are several activity triggers which may be identified for different modules of the SENDS architecture. In particular, the smoothing engines are all controlled by the pair of signals (`start_uc`, `end_process`), and they could be clock gated whenever they are non active, waiting for sufficient pixels to come into the neighbors so that a new pixel window may be processed. Given that the smoothing engines consume the largest part of dynamic power, this activity trigger should be a valuable power reduction opportunity.

We applied our methodology to the SENDS design, using different test benches corresponding to different image processing algorithms. Our statistical detection was able to correctly identify the activity trigger (`start_uc`, `end_process`) for the smoothing engines in all the test benches, as well as other activity triggers for the unit control and the serializer. We also applied the detection algorithm to the top level design composed of 4 SENDS clusters working in parallel. We were able to detect the corresponding pair for each individual cluster in the design, which shows that the statistical detection scales up to a higher design level. For each of those experiments, the detection took less than 20min of computation.

For the case of a single SENDS cluster, we clock gated the `macro_pe` module using the detected activity trigger. This was straightforward given that a single signal transition controls the wake-up (`start_uc`: $0 \rightarrow 1$) and the shutdown (`end_process`: $0 \rightarrow 1$). We did this experiment for two different image processes using 3x3 and 5x5 pixel windows. We then estimated the power savings by running an RTL power estimation tool before and after the modification (technology is CMOS TSMC 45nm worst-case). For the test benches that we used, the clock gating brought power savings of 33% for the 3x3 case and 40% for the 5x5 case.

Of course, the activity trigger for the smoothing engines can also be identified manually with a detailed understanding of the SENDS architecture and how it is used and configured at the top level. However, such power optimization is typically finer grained that what is performed manually at the architectural level, and it could easily be missed. This optimization is somehow at an intermediate level between what is done at the architecture/system level (such as shutting down a whole cluster when it is not used) and what is done at the local register level such as ODC/STC. Furthermore we ran a commercial tool for power reduction using ODC/STC based methods on the SENDS design, and it

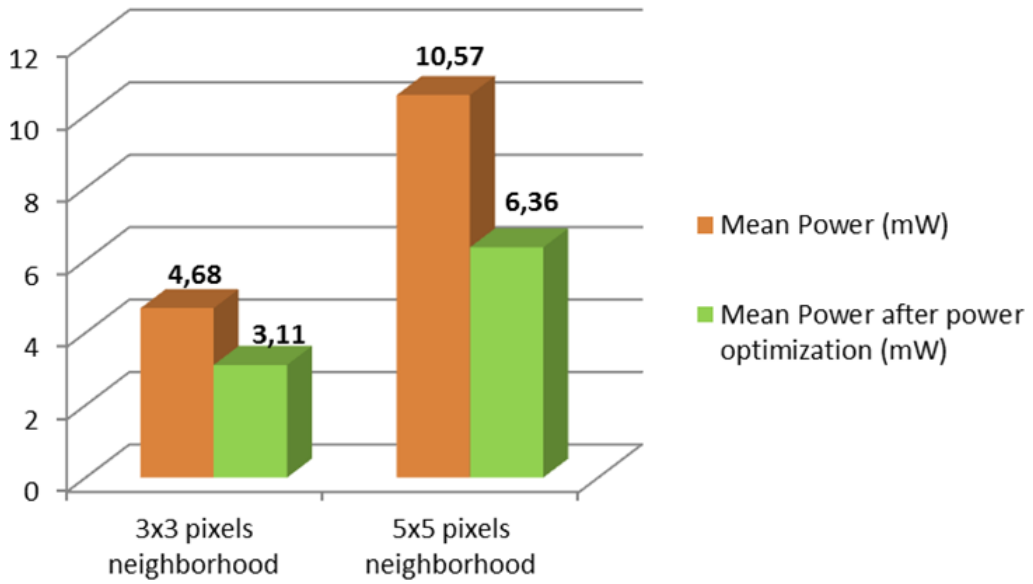


Figure 4.15: Mean power consumption depending on the pixel neighborhood width before and after power optimization with the new elaborated optimization rule

was unable to find a simple and effective condition like the one found using our methodology. Some clock-gating opportunities were identified at the register level, but the enable conditions were really complex, not comprehensible to designers that know the architecture. In total ODC/STC based power reduction brought only 1 % power savings for the same test benches.

Concerning the formal part, the initial result for the activity trigger (`start_uc`, `end_process`) was unproved as we obtained a counter-example. Looking at the violating trace we could quickly observe that the behavior was spurious. To eliminate the behavior we added the following two SVA constraints:

1. A constraint for the architecture configuration (size of pixel window, number of parallel smooth engines to use etc.). The constraint basically consisted in setting the value of a configuration register appropriately.

2. A constraint to limit the pixel stream throughput. This was necessary to avoid the behavior where the system gets overloaded with more data than it can handle (in practice, the pixel stream is coming from the camera which has a limited frame rate). To set this constraint we specified a minimum amount of time between changes at the neighbors module input.

With these constraints our tool was able to formally prove the activity trigger property in about 5 hours of running the formal engine. Therefore, in addition to detecting the optimization automatically, we were able to provide a strong guarantee that the clock gating based on the activity trigger does not alter the functional behavior of the IP.

Furthermore, to explore the applicability of our method to a wider set of designs, we tested our tool on a set of 49 industrial designs in a fully automated mode (i.e., we run the formal check on all the candidates generated by the detection tool without specifying any SVA constraints). We were able to identify a good number of triggers using the statistical detection for most of the designs. On the other hand the results of the formal check were not conclusive as we got only few proofs but also some failures and many timeouts. Given our experience on the SENDS design and other test cases, this can be explained by the fact that SVA constraints are usually needed before we can prove the activity trigger property, at least when it applies to a medium-size block. Additionally, in this fully automatic experiment we had to set quite a small timeout for the formal check, 15 minutes per property. We did a more in-depth analysis for few designs where the activity triggers looked promising. In particular we could see some triggers related to FIFOs (e.g., when a FIFO is full or empty this entails that some other modules get blocked), and DMA (e.g., a module is idle while waiting for a DMA request to be processed).

design	power	power covered	# registers	reg-covered
1	4.169 mW	75.84%	26680	37.17%
2	7.163 mW	54.93%	9128	56.38%
3	0.479 mW	49.62%	1352	82.84%
4	7.145 mW	49.47%	9128	13.56%
5	5.314 μW	31.04%	326	33.74%
6	0.606 mW	16.30%	2070	6.96%
7	8.891 mW	15.58%	690	28.70%
8	92.491 mW	6.77%	30520	4.65%
9	55.851 mW	4.54%	107848	8.14%
10	92.444 mW	2.87%	114546	1.56%
11	1.430 μW	1.61%	162	14.81%
12	4.079 mW	0.70%	5292	0.15%
13	149.955 mW	0.61%	111012	1.11%
14	400.937 mW	0.31%	160530	1.58%
15	8.013 mW	0.05%	43504	0.01%
16	2.799 mW	0.02%	38158	0.04%

Table 4.1: Automated run – formal verification results

We present in Table 4.1 the formal verification results for the designs whose triggers could be proved automatically. The ‘power’ column indicates the dynamic power consumption of the circuit as computed by a power estimation tool and the ‘power covered’ column show what fraction of this power is consumed by the modules for which we found and formally verified some activity triggers. Note that this is an upper bound on the power reduction and the real savings depend on the relative duration of the idle periods. The ‘#registers’ column contains the total number of registers in the circuit while the column ‘reg-covered’ shows the percentage of registers that fall within the scope of the triggers.

As discussed before, the percentage of automatically proven triggers is quite small compared to all opportunities detected by the statistical engine. We learned from the SENDS experience that in most cases additional constraints are needed for a successful formal proof. However, the main observation to be taken from Table 4.1 is the relatively good scalability of the formal verification. We are able, within 15 minutes limit, to formally prove some activity triggers for design blocks that contain thousands of registers and consume enough dynamic power to be interesting for clock gating.

4.8 Limitations and future work

We developed and implemented a method for semi-automatic detection of activity triggers in electronic circuits as well as formal verification of their correctness. We demonstrated that the method can achieve a significant power reduction on an image processing architecture for which we had a good knowledge of design intent. We also applied the method in a fully automatic manner to a larger set of industrial designs with some more modest results. The lesson from this experience is that a proper modularization, specification of input constraints and some human intervention are, in most cases, crucial for a successful application of the method. The major current limitations of the approach are:

1. The sensitivity of the statistical detection procedure to interference, especially in designs that are not highly modularized;

2. The need for extensive simulation data in order to detect triggers. This prevents the use of the method in very early stages of the design flow when such simulation data may not be available.

3. The usual limitations related to the complexity of the formal check.

There is a room for improvement in addressing these issues. One idea is to use automatic modularization, e.g., every bus is analyzed separately, to cope with the noise that comes from a small part of the analyzed module. Furthermore, the trigger detection engine could be improved, possibly using machine learning (as in [5, 69]) or design intent detection heuristics inspired by [32]. Some triggers may be purely functional, independent on the environment. There is an initiative to work on structural analysis of such triggers that would allow us to identify them without simulation, much earlier in the design flow, contributing to the general 'shift to the left' trend in the silicon industry. We made some initial research in this direction but so far we were not able to process non-trivial designs. The success of the formal part of activity trigger detection strongly depends on the designers' readiness to provide detailed input constraints that can be quite complex. We observed that some natural constraints such as alternation of start and stop events are often present, and hence can be used even if they are not specified. If the formal proof succeeds under this assumption, it could be reported as a hint to the user. More constraint hints can be obtained by data mining the simulation traces.

Hardware: the parts of a computer that can be kicked.

Jeff Pesis

5 Conclusion

In this thesis we presented two novel methods for reducing switching power in digital circuits, power-aware synthesis and activity trigger detection.

Power-aware synthesis optimizes synthesis of combinatorial logic with respect to an input model. The optimized logic exhibits lower switching during typical computation. The optimization is performed on the AIG level by re-arranging AND gates within AND cones. We implemented the method using AIG manipulation tools from the ABC software and tested the method on a variety of synthetic models and two simple hardware designs. The tests on synthetic models suggest that the potential for power reduction is strongly related to amount of determinism in the input model with no savings for uniform input and large savings for models that are almost deterministic. The method has two main shortcomings. First the reduction is achieved on the AIG level and it is not clear that switching reduction is preserved after technology mapping unless we restrict ourselves to AND and inverter gates. We performed experiments in this direction, however the results are inconclusive. The second shortcoming is that the switching reduction is achieved on AIG networks that were not optimized via other methods. When we preprocessed the AIG by some space and time optimization tools from ABC, the switching reduction achieved by power-aware synthesis was much less significant.

Activity trigger detection searches signal traces provided by simulation in order to find signals that are statistically related to a temporary cease of activity in sub-circuits. If the relation is not spurious, we can use these signals to efficiently clock gate these sub-circuits. We verify the validity of the proposed clock gating by model checking. The user is able to interact with the process by adding constraints if model checking returns an unrealistic counter example. The main advantage over state-of-art clock gating condition detection is that our method can detect conditions for larger sub-circuits and the provided conditions are simple and usually related to design intent. Therefore they are

less expensive to implement and more understandable to human designers who may be more comfortable using them. The main limitations are the need for extensive simulation data, the sensitivity of the statistical detection to interference, insufficient modularization at the HDL level in many designs and the usual limitations related to the complexity of formal verification. The method was implemented as a new feature of an industrial EDA software tool.

Acronyms

AIG And-Inverter Graph. 12, 15–17, 19, 20, 26, 29, 45, 46, 78, *Glossary:* And-Inverter graph

ASIC application specific integrated circuit. 8

ATD Activity Trigger Detection. *Glossary:* activity trigger detection

CMOS complementary metal-oxide-semiconductor. 3

DAG Directed Acyclic Graph. 15, 20

EDA Electronic Design Automation. 12, 15

HDL hardware description language. 79

IC integrated circuit. 3

RTL Register Transfer Level. 8, 10, 12, 15, 50, *Glossary:* register transfer level

Glossary

ABC A System for Sequential Synthesis and Verification [63]. EDA software tool developed at Berkeley Verification and Synthesis Research Center . 15, 45, 46, 78

activity An average number of switches in a signal or (sub)circuit per clock cycle . 11, 17

activity trigger detection Our method to reduce the switching activity in sequential circuits by using activity triggering conditions for clock gating. 2, 10, 77, 78

activity triggers Conditions triggering activity or stability in a design block. Used for clock gating in activity trigger detection. 52

And-Inverter graph And-Inverter graph A technology independent netlist . 12, 15

circuit design A stage in the ASIC design flow. Transforms an RTL description of an IC into a netlist. Technology mapping is performed in this phase. 8, 12

clock gating A technique that reduces the dynamic power in circuits by disabling the clock signal for sub circuits that currently do not perform a useful computation. 50, 53

clock tree A structere that routes the clock signal to individual sequential elements . 53, 54

cone A tree of ANDgates in AIG that can be merged into one functionally equivalent ANDgate with multiple inputs . 19, 78

cut point Border nodes of a cone. Places where AIG can be cut into cones . 19

dynamic power The power that is consumed by charging the IC gate in direct proportion to the switching rate of the gate. Main components are the switching power and short circuit power. 6, 12, 50

gate A logical element performing a simple boolean function. Can be either abstract or standard cells coming from a standard cell library . 8

Gnd the ground in the IC. 3

input vector Stimuli applied to the circuit's input's. Sequence of input vectors determines the behavior of the circuit (together with initial values of registers) . 11

logic design A stage in the ASIC design flow. Transforms a behavioral description of an IC into RTL by performing various steps including *logic minimization*. 8

netlist A data structure capturing abstract topology of a design. Contains gates from a technology library. 8, 12, 13, 15

physical design A stage in the ASIC design flow during which a plan for the physical layout of the chip is created .The gates in the netlist are mapped to physical destinations on the chip, wires are routed, clock trees generated . 8

power dissipation transformation of electrical energy into heat during IC operation. 3

power-aware synthesis Our method to reduce the switching activity in combinatorial circuits by optimizing their structure at the AIG. 2, 10, 16, 37, 78

register transfer level Register transfer level - a description of the function circuit by using only ports, registers, buses and simple expressions for combinatorial logic. Can be expressed in HDL languages . 8

short circuit power The power that is consumed by a temporal short circuit current that flows through the transistors for short time when the switch is performed due to the continuous nature of the real non-ideal transistors. Component of the dynamic power. 5

standard cell A group of physical structures (transistors and wires) implementing a logical *gate*. The information about physical properties of a standard cell are provided in standard cell librarys . 8, 15

standard cell library A list of standard cells associated with information about delay, power, space taken by the gate and other physical qualities. Standard cell libraries are provided by the foundries where the chips are going to be manufactured . 15

switching activity The ratio between the number of switches in a gate and the clock rate. Also can be seen as a probability that a given gate value will switch in a given clock cycle.. 6

switching power the power that is consumed by charging the IC gate capacitance and dissipated by discharging, component of the dynamic power. 3

technology mapping Process of mapping a netlist containing abstract gates to a netlist of standard cells. 15

Vdd the source of charge in an IC or the voltage associated with this source. 3

Bibliography

[1] Henrik Andersen and Henrik Hulgaard. “Boolean expression diagrams”. In: *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*. IEEE. 1997, pp. 88–98.

[2] Eli Arbel, Cindy Eisner, and Oleg Rokhlenko. “Resurrecting infeasible clock-gating functions”. In: *Proceedings of the 46th annual Design Automation Conference (DAC)*. IEEE. 2009, pp. 160–165.

[3] Eli Arbel, Oleg Rokhlenko, and Karen Yorav. “SAT-based synthesis of clock gating functions using 3-valued abstraction”. In: *Formal Methods in Computer-Aided Design (FMCAD), 2009*. IEEE. 2009, pp. 198–204.

[4] Pietro Babighian, Luca Benini, and Enrico Macii. “A scalable ODC-based algorithm for RTL insertion of gated clocks”. In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Vol. 1. 2004, pp. 500–505.

[5] Pietro Babighian, Gila Kamhi, and Moshe Vardi. “Interactive presentation: PowerQuest: trace driven data mining for power optimization”. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*. EDA Consortium. 2007, pp. 1078–1083.

[6] A Bellaouar and Mohamed I Elmasry. *Low-power digital VLSI design: Circuits and systems*. Kluwer, 1995.

[7] L. Benini et al. “Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers”. In: *ACM Transactions on Design Automation of Electronic Systems* 4.4 (1999), pp. 351–375.

[8] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. “A survey of design techniques for system-level dynamic power management”. In: *IEEE Transactions on VLSI* 8.3 (2000), pp. 299–316.

[9] Luca Benini and Giovanni DeMicheli. *Dynamic power management: design techniques and CAD tools*. Springer Science & Business Media, 2012.

[10] Luca Benini et al. “Glitch power minimization by selective gate freezing”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 8.3 (2000), pp. 287–298.

[11] Armin Biere et al. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS*. 1999, pp. 193–207.

[12] Mark Bohr and Kaizad Mistry. *Intel’s Revolutionary 22 nm Transistor Technology*. 2011.

[13] Aaron R Bradley. “SAT-based model checking without unrolling”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2011, pp. 70–87.

[14] Robert K. Brayton and Alan Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool”. In: *Proceedings of the 22nd Computer Aided Verification International Conference (CAV)*. 2010, pp. 24–40.

[15] Robert K Brayton et al. *Logic minimization algorithms for VLSI synthesis*. Springer, 1984.

[16] Anantha P Chandrakasan and Robert W Brodersen. *Low power digital CMOS design*. Kluwer, 1995.

[17] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. “Low-power CMOS digital design”. In: *IEICE Transactions on Electronics* 75.4 (1992), 371–382.

[18] Satrajit Chatterjee et al. “On Resolution Proofs for Combinational Equivalence”. In: *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*. 2007, pp. 600–605.

[19] D. Chiou. “Keynote talk II: Accelerating data centers using reconfigurable logic”. In: *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*. 2015, pp. 60–60.

[20] Jason Cong, Bin Liu, and Zhiru Zhang. “Behavior-level observability don’t-cares and application to low-power behavioral synthesis”. In: *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*. ACM. 2009, pp. 139–144.

[21] O. Coudert. “Gate sizing for constrained delay/power/area optimization”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 5.4 (1997), pp. 465–472.

[22] Jack Edmonds. “Maximum matching and a polyhedron with 0, l-vertices”. In: *J. Res. Nat. Bur. Standards B* 69 (1965), pp. 125–130.

[23] Niklas Een, Alan Mishchenko, and Robert Brayton. “Efficient implementation of property directed reachability”. In: *Formal Methods in Computer-Aided Design (FMCAD), 2011*. IEEE. 2011, pp. 125–134.

[24] Mahmoud Elbayoumi, Michael S Hsiao, and Mustafa ElNainay. “Novel SAT-based invariant-directed low-power synthesis”. In: *16th International Symposium on Quality Electronic Design (ISQED), 2015*. IEEE. 2015, pp. 217–222.

[25] Michael D Ernst et al. “Dynamically discovering likely program invariants to support program evolution”. In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.

[26] Michele Favalli and Luca Benini. “Analysis of glitch power dissipation in CMOS ICs”. In: *Proceedings of the 1995 international symposium on Low power design*. ACM. 1995, pp. 123–128.

[27] Richard Phillips Feynman, JG Hey, and Robin W Allen. *Feynman lectures on computation*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[28] Ranan Fraer, Gila Kamhi, and Muhammad K. Mhameed. “A New Paradigm for Synthesis and Propagation of Clock Gating Conditions”. In: *Proceedings of the 45th Annual Design Automation Conference (DAC)*. Anaheim, California: ACM, 2008, pp. 658–663. ISBN: 978-1-60558-115-6.

[29] Malay K. Ganai and Andreas Kuehlmann. “On-the-Fly Compression of Logical Circuits”. In: *in International Workshop on Logic Synthesis*. 2000.

[30] Gary D Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms*. Springer, 2006.

[31] Gary D Hachtel et al. “Markovian analysis of large finite state machines”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 15.12 (1996), pp. 1479–1493.

[32] Sudheendra Hangal et al. “IODINE: a tool to automatically infer dynamic invariants for hardware designs”. In: *Proceedings of the 42nd annual Design Automation Conference (DAC)*. ACM. 2005, pp. 775–778.

[33] Richard Herveille. *SPI Core*. [Online; accessed 9-March-2016]. 2014. URL: http://opencores.org/projects,simple_spi.

[34] Henrik Hulgaard, Poul Frederick Williams, and Henrik Reif Andersen. “Equivalence checking of combinational circuits using boolean expression diagrams”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 18.7 (1999), pp. 903–917.

[35] Aaron P Hurst. “Automatic synthesis of clock gating logic with controlled netlist perturbation”. In: *Proceedings of the 45th annual Design Automation Conference (DAC)*. ACM. 2008, pp. 654–657.

[36] “IEEE Standard for Standard SystemC Language Reference Manual”. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012).

[37] *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification (IEEE Std 1800–2005)*. 2005.

[38] “IEEE Standard for Verilog Hardware Description Language”. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006).

[39] “IEEE Standard VHDL Language Reference Manual”. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009).

[40] Hans Jacobson et al. “Stretching the limits of clock-gating efficiency in server-class processors”. In: *Proceedings of the International Symposium on High-Performance Computer Architecture*. 2005, pp. 238–242.

[41] Kurt Keutzer. “DAGON: technology binding and local optimization by DAG matching”. In: *Papers on Twenty-five years of electronic design automation*. ACM. 1988, pp. 617–624.

[42] Nam Sung Kim et al. “Leakage current: Moore’s law meets static power”. In: *computer* 36.12 (2003), pp. 68–75.

[43] Zvi Kohavi and Niraj K Jha. *Switching and finite automata theory*. Cambridge University Press, 2010.

[44] Thomas Koshy. “Catalan numbers with applications”. In: (2008).

[45] Jan Lanik and Oded Maler. “On Switching Aware Synthesis for Combinational Circuits”. In: *Hardware and Software: Verification and Testing*. Ed. by Nir Piterman. Vol. 9434. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 276–291.

[46] Jan Lanik et al. “Reducing power with activity trigger analysis”. In: *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*. IEEE. 2015, pp. 169–178.

[47] Lawrence L Larmore and Daniel S Hirschberg. “A fast algorithm for optimal length-limited Huffman codes”. In: *Journal of the ACM (JACM)* 37.3 (1990), pp. 464–473.

[48] Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Dover Publications, 1976.

[49] Eric Lehman et al. “Logic decomposition during technology mapping”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.8 (1997), pp. 813–834.

[50] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. “The glory of the past”. English. In: *Logics of Programs*. Ed. by Rohit Parikh. Vol. 193. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1985, pp. 196–218.

[51] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[52] Alan Mishchenko and Robert Brayton. “Faster logic manipulation for large designs”. In: (2013).

[53] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. “DAG-aware AIG rewriting a fresh look at combinational logic synthesis”. In: *Proceedings of the 43rd annual Design Automation Conference*. ACM. 2006, pp. 532–535.

[54] Alan Mishchenko et al. “A semi-canonical form for sequential AIGs”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2013, pp. 797–802.

[55] Trevor Mudge. “Power: A first-class architectural design constraint”. In: *Computer* 4 (2001), pp. 52–58.

[56] Rajeev Murgai, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. “Decomposition of logic functions for minimum transition activity”. In: *Proceedings of the 1995 European conference on Design and Test*. IEEE Computer Society. 1995, p. 404.

[57] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998.

[58] Joseph Paradiso, Thad Starner, et al. “Energy scavenging for mobile and wireless electronics”. In: *Pervasive Computing, IEEE* 4.1 (2005), pp. 18–27.

[59] Jan Rabaey. *Low power design essentials*. Springer Science & Business Media, 2009.

[60] Jan M Rabaey and Massoud Pedram. *Low power design methodologies*. Springer, 1996.

[61] Tsutomu Sasao. *Switching theory for logic synthesis*. Vol. 1. 0. Springer, 1999.

[62] Naveed A Sherwani. *Algorithms for VLSI physical design automation*. Springer Science & Business Media, 2012.

[63] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/*. 2015.

[64] Vivek Tiwari, Pranav Ashar, and Sharad Malik. “Technology mapping for low power”. In: *Design Automation, 1993. 30th Conference on*. IEEE. 1993, pp. 74–79.

[65] Vivek Tiwari, Sharad Malik, and Pranav Ashar. “Guarded evaluation: Pushing power management to logic synthesis/design”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.10 (1998), 1051–1060.

[66] Chi-Ying Tsui, Massoud Pedram, and Alvin M Despain. “Power efficient technology decomposition and mapping under an extended power consumption model”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.9 (1994), pp. 1110–1122.

[67] Chi-Ying Tsui, Massoud Pedram, and Alvin M Despain. “Technology decomposition and mapping targeting low power dissipation”. In: *Proceedings of the 30th international Design Automation Conference*. ACM. 1993, pp. 68–73.

[68] Neil HE Weste and Kamran Eshraghian. *Principles of CMOS VLSI design*. Vol. 2. Addison-Wesley Reading, MA, 1993.

[69] Roni Wiener, Gila Kamhi, and Moshe Y Vardi. “Intelligate: scalable dynamic invariant learning for power reduction”. In: *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Springer, 2009, pp. 52–61.

[70] Wikipedia. *Serial Peripheral Interface Bus — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-March-2016]. 2016. URL: https://en.wikipedia.org/w/index.php?title=Serial_Peripheral_Interface_Bus&oldid=704779254.

[71] Chingwei Yeh, Chin-Chao Chang, and Jinn-Shyan Wang. "Technology mapping for low power". In: *Design Automation Conference, 1999. Proceedings of the ASP-DAC'99. Asia and South Pacific*. IEEE. 1999, pp. 145–148.

[72] Hai Zhou and DF Wong. "An exact gate decomposition algorithm for low-power technology mapping". In: *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society. 1997, pp. 575–580.