Title
Verification and Synthesis of Clock-Gated Circuits

Author
Dai, Yu-Yun

Publication Date
2017

Peer reviewed|Thesis/dissertation

Verification and Synthesis of Clock-Gated Circuits

by

Yu-Yun Dai

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Robert K. Brayton, Chair
Professor Sanjit Seshia
Professor Alper Atamtürk

Summer 2017

Verification and Synthesis of Clock-Gated Circuits

Abstract

Verification and Synthesis of Clock-Gated Circuits

by

Yu-Yun Dai

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Robert K. Brayton, Chair

As system complexity and transistor density increase, the power consumed by digital integrated circuits has become a critical constraint for VLSI design and manufacturing. To reduce dynamic power dissipation, clock-gating synthesis techniques are applied to circuits to prune register updates by modifying the next-state functions of the registers. Hence to verify this kind of synthesis, sequential equivalence checking (SEC) of clock-gated circuits is required.

In this thesis, we examine the application of reverse engineering and control logic extraction to assist in the analysis and verification of clock-gated circuits. The proposed methodology also enables sequential clock-gating synthesis to further reduce dynamic power. A secondary focus is on recognizing circuit functionalities with deep learning techniques.

The first part of the work deals with the use of transparent logic to recognize control and data paths of gated-level circuits. We invent abstraction models (dependencies graphs, DGs) of sequential circuits and then explain how they can be used to formulate sufficient conditions for legal clock-gating. It is then demonstrated how to perform efficient sequential equivalence checking (SEC) between a circuit before and after clock-gating synthesis based on DGs. The proposed formulation is extended to allow sequential clock-gating synthesis to be done systematically and automatically.

The second part of the thesis introduces the use of neural networks to recognize circuit properties, which can be used to benefit and improve reverse engineering methods. We invent a representation of gate-level circuits to work with neural networks and build a framework for circuit recognition, including function classification and detection. The proposed framework can also be used to locate high-level constructs in the sea of logic gates.

To people who have made me a better person.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Robert Brayton for all the guidance, help, support and encouragement during my graduate study in Berkeley. He has been a wonderful mentor and living example for me to be a better researcher and human being. Many thanks to Ruth for sharing her observations of Bob and a lot of heartwarming moments. I wish I could become a person like Bob who has been an incredible role model for many people.

It has been my pleasure to work with Dr. Alan Mishchenko during the past four years. Without his ABC framework, my research projects might be significantly delayed. His selfless sharing has provided me plenty of ideas about not only researches on verification and synthesis, but several aspects of life.

My thanks goes to Professor Sanjit Seshia, Professor Alper Atamtürk and Prof. Andreas Kuehlmann for their valuable feedback to my thesis work. Their courses on verification, mathematical programming and logic synthesis have established essential foundations for my research works.

I have enjoyed the interactions with my colleagues in our group, Sayak Ray, Baruch Sterin, Jiang Long and Yen-Sheng Ho. Indeed, I need to express my special thanks to Baruch for all of his generous and patient helps. Also, I thank Tianshi Wang, Karthik Aadithya, Antonio Iannopollo, Baihong Jin, Shromona Ghosh, Pierluigi Nuzzo, Chung-Wei Lin and Ben Zhang from other groups for the many wonderful moments and conversations I have had with them in the DOP center.

Prof. Jajeet Roychowdhury has dramatically changed my graduate study since my first year in Berkeley. His courses on numerical methods, his challenging questions in my first preliminary exam, and his demands for my paper writing and presentation skills have sharpened me a lot. Without his push, I might not be able to proceed forward so quickly.

Working with Prof. Anant Sahai and Michel Maharbiz for the EE16B course has been a unique and unforgettable experience in Berkeley. I have learned a lot from my fellow GSIs and uGSIs, as well as my students. I also thank Prof. Stavros Tripakis for giving me the chance to teach in EE244 as the way I wanted.

I would like to express my special thanks to Kei-Yong Khoo, who was the manager for my internship in Cadence. He introduced the concept of clock-gating to me, which has been the main focus of this thesis. I have had a great time with my mentor Danny Ho and other members from the Conformal team. I would recommend this team to everyone who might want to work on combinational equivalence checking or ECO problems.

The weekly lunches and chats with members in WICSE have been significant entertainments during the four years. I thank Qie Hu, Fanny Yang, Vidya Muthukumar, Mindy Perkins, Regina Eckert, Sandy Huang, Rashmi Vinayak, Cathy Wu, Alice Ye, Penporn Koanantakool and Mangpo Phitchaya Phothilimthana for many marvelous conversations we have had together.

I would like to give a special note of thanks to Shirley Salanio, for her prompt and cheerful help for all concerns or issues I have during my graduate study in the EECS department.

I thank my family members, especially my parents, for shaping me to the person who I am. Their influences on all of my characteristics can hardly be reciprocated. I hope I have made them proud.

I want to express my deepest gratitude to my husband, Wei-Hsun (Wish) Lin, for his consistent and unconditional support, even before I applied for the Ph.D. program. We have been the best friend to each other, and I believe we will share more monumental moments in the future.

Chapter 1

Introduction

This dissertation is a study of how formal methods and reverse engineering can be used to verify and synthesize digital circuits for minimizing dynamic power consumption. In the course of this study, several new concepts and techniques are introduced to analyze and extract circuit properties to build a framework for verification and synthesis to achieve low power circuit designs.

Before moving to the main part of this work, we begin with some background about sequential clock-gating in Section 1.1, and existing methods for sequential equivalence checking in Section 1.2.

It is assumed throughout that the reader has some familiarity with algorithmic notation, the vocabulary and terminology of digital design, Boolean logic and temporal logic.

1.1 Preliminaries

A sequential circuit, as shown in Figure 1.1, consists of a combinational logic part (C_L), and sets of primary inputs (PIs), outputs (POs) and memory components (flip-flops, FFs). The combinational part represents a functional mapping from PIs and current states of FFs, to POs and the next state of each FF.

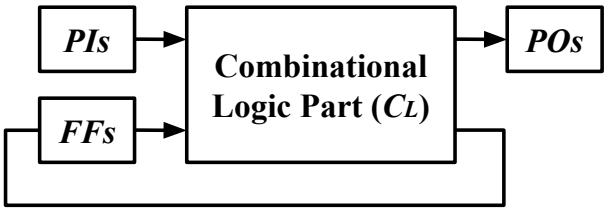


Figure 1.1: Standard representation of a sequential circuit.

In modern VLSI design flows, combinational and sequential synthesis techniques [23] are applied to sequential circuits to minimize chip area, reduce power consumption, optimize

the clock period, etc. Combinational synthesis preserves the sequential behavior of FFs to reduce the cost functions, while sequential synthesis provides more flexibility by possibly changing the next state functions and thereby producing further reductions in the power consumption.

To manufacture chips while considering power consumption, static and dynamic power of running the circuits are analyzed. Static power refers generally to the power required to maintain the state of a circuit, while dynamic power refers to power needed during switching activity. In this thesis, we only focus on dynamic power consumption.

To reduce the power consumed for updating FFs, synthesis tools apply *satisfiability (forward)* and *observability (backward) clock-gating* [20]. Satisfiability clock-gating is used to disable the clock of a FF when its input data remains unchanged during the current clock period. Thus, a FF need not be updated if the incoming data remains the same as that saved in the register. Observability clock-gating turns off the clocks to FFs when the current values of the FFs input can never be observed at the POs.

Generally, satisfiability and observability clock-gating techniques modify the clocks of FFs by using enabling signals. These are sequentially redundant in that they can be removed without modifying any observed sequential behavior. These redundancies are used to minimize the frequency of updating some of the FFs, hence reducing dynamic power consumption. Therefore, the clock-gated circuit will keep the same sequential properties but with fewer updates of the memory components.

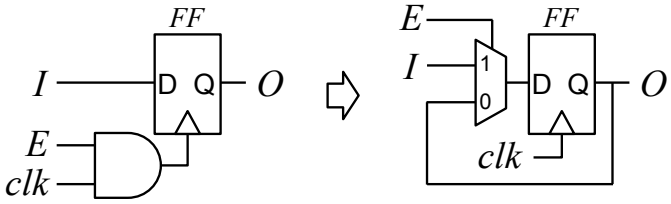


Figure 1.2: An equivalent model for clock-gating a FF.

As shown in Figure 1.2, clock-gating a FF can be modeled with a feedback loop through a multiplexer, where the enable signal, E, controls the switching of O between I and its old value. This allows the modeling of clock-gated circuits to be done with only generic FFs. Notice that in practice, the circuit on the left-hand side of Figure 1.2 can result some latch issues, so in practice, real clock-gating construct can be more complicated; however its modeling is the same as shown in Figure 1.2.

Once synthesis is applied to a circuit, sequential equivalence checking (SEC) is used between the circuits before and after synthesis [24]. As the need for SEC techniques increases, efficient methods become not only more necessary but also enable the use of sequential synthesis in the first place as such synthesis has usually avoided due to verification complexity. The complexity of general SEC is P-SPACE complete, and hence more complex than combinational equivalence checking (CEC), which is only NP-complete.

1.2 Background

After synthesis, we need to ensure that the circuits before and after synthesis are sequentially equivalent [2]. SEC for general circuits is typically formulated as a model checking problem on the miter between two sequential circuits (where the PI pairs are merged and the PO pairs are XORed to form the outputs of the miter circuit). Thus, sequential model checking techniques, including induction [42], bounded model checking (BMC) [5] and property-directed reachability (PDR) [14], can be applied to check if the outputs of the two circuits are identical forever. If an output of the miter can ever become *1*, meaning this pair of POs are evaluated to distinct values under the same input sequences, sequential equivalence is violated. In this case, model checking can provide an input sequence leading to the violation. Otherwise, the two circuits are proved sequentially equivalent.

Due to the P-SPACE complexity of model checking, applying this to SEC problems may be too hard. Of course, CEC can be tried and if successful, the two circuits are also sequentially equivalent. However, most effective sequential synthesis techniques tend to change the next state functions. For circuits under retiming and resynthesis, when the history of sequential synthesis is kept, the SEC problems can be reduced and solved more easily [21]. Similarly, to verify circuits after clock-gating synthesis, the SEC problem can be reduced to a CEC problem by an existing technique. Savoj at el. [41, 40] proposed a combinational approach to SEC for clock-gating synthesis. This approach aimed at circuits synthesized using satisfiability and observability don't cares (SDC and ODC) [39]. Although it worked well compared to existing methods, this approach has several weaknesses: (1) it cannot conclude non-equivalence, and therefore cannot supply a witness to help understand the reason for this, (2) it requires unrolling and there is no suggested number of timeframes for unrolling, (3) it still has a scalability problem, especially when the combinational logic is extremely complicated.

It is therefore important to develop a systematic approach to verify proposed clock-gating conditions on circuits. Additionally, an automatic method is proposed to find further clock-gating conditions for synthesis to reduce dynamic power consumption.

1.3 Contributions

In this thesis, we first propose an SEC method to apply to clock-gated circuits, based on the fact that sequential clock-gating synthesis usually only adds extra control logic to disable clocks when applicable. Given a pair of circuits, golden (**G**) and revised (**R**), where **R** is clock-gated from **G**, we build abstraction models, *characteristic graphs*, for both. Then for each candidate of a set of clock-gated FFs, we formulate sufficient conditions for legal clock-gating based on the characteristic graph. Once the sufficient properties are justified and the clock-gating condition is proved legal, the extra control logic is indeed a sequential redundancy for the original circuit. By reducing all proved sequential redundancies on the input circuits, the two circuits become more similar to each other, so the SEC problem gets

easier and hence can be solved usually by existing general SEC engines quickly.

Moreover, to resolve more complicated clock-gating conditions, we should consider more properties of the circuits which are excluded by characteristic graphs. To extract detailed control logic and data dependencies from circuits, we introduce a concept called *transparent logic* to model data flow under control. We also propose a functional approach to identify transparent logic from gate-level circuits. Unlike structure approaches which rely on structure matching, the proposed method can recognize more transparent logic and provide more insights of the target circuit. This concept can be used widely in verification and reverse engineering.

Based on recognized transparent logic, we invent another abstraction model, *dependency graphs*, to describe clock-gating structures and data dependencies for sequential circuits. Those graphs represent the essential information of clock-gating, but bypassing irrelevant combinational logic blocks. Then we formally formulate sufficient properties of legal clock-gating on dependency graphs using temporal logic. Those properties can be represented as circuits and verified by hardware model checkers. Dependency graphs are compatible with the proposed SEC flow that relies on characteristic graphs.

Moreover, the formulation of dependency graphs leads to clock-gating synthesis algorithms. We propose an automatic flow to synthesize enable signals which turn off the clocks of target FFs to reduce the frequency of their updating. This flow can be adopted into a modern VLSI design flow to achieve low power circuits.

To retrieve more high-level information from gate level circuits, we also experiment with deep learning techniques to recognize functionalities (like datapath operators) of circuit blocks. We invent a new representation for circuits to work with machine learning techniques and demonstrate how it can capture essential features for circuit recognition. To the best of our knowledge, this is probably the first work applying neural networks to recognize circuit types.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents *characteristic graphs* and the usage in verification of clock-gating conditions. Chapter 3 introduces transparent logic and a functional approach to recognize transparent logic in hardware designs. The definition and construction of dependency graphs are stated in Chapter 4, while Chapter 5 shows how to formulate legal clock-gating conditions on dependency graphs. The proposed clock-gating verification and synthesis flows are demonstrated in Chapter 6 and 7, respectively. Chapter 8 explains how to recognize functional blocks with deep learning techniques. Lastly, Chapter 9 concludes this thesis.

Chapter 2

Verification with Characteristic Graphs

The methodology introduced in this chapter is based on the observation that some sequential synthesis methods only modify a circuit by introducing control structures that are sequentially redundant [20]. Hence equivalence checking can be based on detecting and proving these redundancies, eliminating them, and then doing SEC between the resulting (simplified) circuit and a *golden model*. We propose such a method that uses an abstraction, characteristic graph (CG), of a circuit to formulate legal clock-gating conditions. We apply them to sequentially clock-gated circuits, and give some experiments comparing the new method against existing techniques.[1]

In the next two chapters we further extend and generalize these ideas with the concepts of dependency graphs (DGs) and transparent paths. DGs address more detailed data dependencies than CGs do. Recognizing transparent logic blocks can decompose larger combinational blocks into smaller sub-circuits and provide more detailed data flow information. The use of DGs complements CGs in that they provide more information, but it takes more time to analyze and build DGs for circuits. However, DGs are able to provide more precise properties for legal clock-gating conditions, which can benefit clock-gating synthesis as well as verification.

This chapter is organized as follows: In Section 2.1, a method for representing some essential features of a clock-gated circuit using a *characteristic graph* is described. Section 2.2 discusses the connection between the CG and sequential redundancy in clock-gated circuits. The overall flow of our SEC method for clock-gated circuits is given in Section 2.3, and Section 2.4 compares the performance of the proposed method with previous works, using several sets of experiments. Section 2.5 summarizes the results and discusses some limitations of the proposed method.

[1]The full version of this chapter has been published in Design Automation Conference 2015 [12].

2.1 Characteristic Graphs

A characteristic graph (CG) is an abstraction of its corresponding circuit. It is a high-level description, which represents only essential characteristics needed for equivalence.

2.1.1 Characteristic Graph

A characteristic graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is a directed graph, where a vertex stands for a group of PIs, POs, FFs or internal signals in the corresponding circuit. Each directed edge represents a signal dependency from one group to another. There are three types of edges: *selection-edge*, *on-edge* and *off-edge*. A selection-edge connects exactly one signal to its target group, to indicate the conditional switch of the group dependency. On-edges and off-edges connect the different sets of support groups to the target group if the selection signal is 1 or 0, respectively. The selection-edge of each PI is driven by a constant *True*, which means the value of each input signal is unconditionally updated. A group containing PIs cannot be driven by other groups. Some FFs can be in the same group with POs because those FFs are also combinational ouputs and controlled by the same selector. Hence this type of group can have edges to other vertices. An algorithm is given in Algorithm 2.1 for constructing the CG of a sequential circuit.

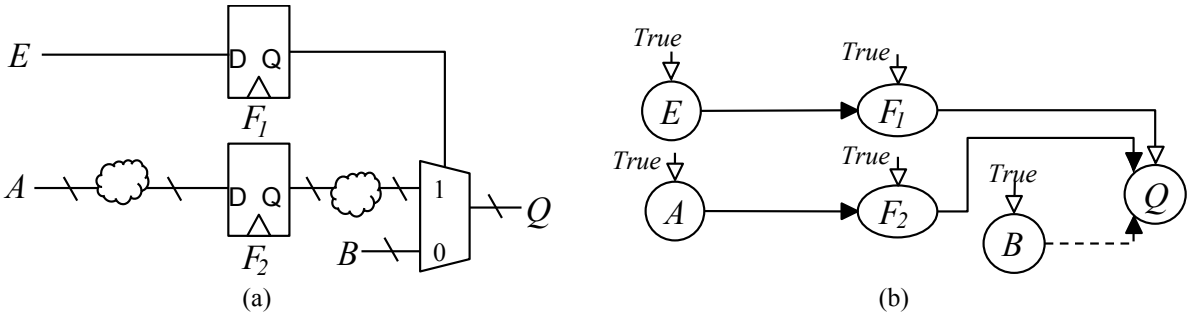


Figure 2.1: A sequential circuit (a) with its characteristic graph (b). Each circle stands for a group of signals, while *selection-edges*, *on-edges* and *off-edges* are represented respectively by solid lines with white arrows, solid lines with black arrows, and dotted lines with black arrows. Q is switched between F_2 and B, where F_1 is the selection signal. The selection-edges of the inputs, E, A and B, are connected to $True$. The selection-edge of F_1 is driven by $True$, because F_1 has no off-edge.

Figure 2.1 depicts a sequential circuit with its corresponding characteristic graph. A line with a white arrow represents a selection-edge, while solid and dotted lines with black arrows are on-edges and off-edges. The circles (vertices) stand for sets of signals in the original circuit, including PIs, E, A and B, the PO, Q, and FFs, F_1 and F_2. Note that there is no on-edge or off-edge into each PI, and their selection-edges are driven by constant *True*. Thus the value of each PI is not driven by any other signal, and it is updated at every clock tick. For the output, Q, the value of F_1 (selection-edge) determines if its value is driven by

B (on-edge) or F_2 (off-edge). Finally, F_1 is only driven by E, so it has no off-edge and its selection is $True$.

To sum up, a characteristic graph abstracts the data dependency among 'essential' signals, while ignoring combinational logic parts, which are usually irrelevant for proving clock-gating equivalence. Although this abstraction is especially motivated by the SEC problem for clock-gated circuits, the idea might be used in similar problems.

2.1.2 Construction of Characteristic Graph

Given a sequential gate-level circuit, the characteristic graph is constructed in three steps: (1) recognize selection signals (2) create vertices (3) build dependencies.

Recognize selection signals: When the input circuit is generated from a synthesis tool, in which 2-to-1 multiplexers (MUXes) are supported and explicitly expressed, the selector inputs of these MUXes are easily recognized and designated as selection signals. If the input circuit is an and-inverter-graph (AIG), the instances of a MUX structure shown in Figure 2.2 needs to be identified. The output signal O is controlled by S and conditionally switched between A and B. Therefore, the signal S here is recognized as the selection signal for the group consisting of output O. This structural matching can be performed over the AIG very quickly. However, it is possible that some essential MUX controls could be missed. Also, in this chapter, it is assumed there is at most one MUX in front of each FF, so it is possible that some clock-gating conditions might be overlooked, which would make the final equivalence check harder.

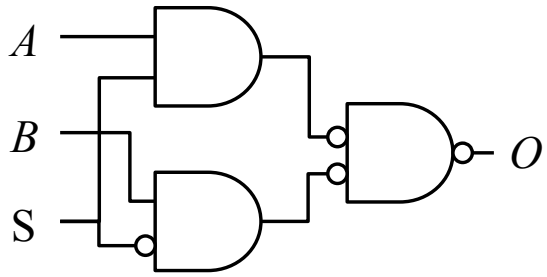


Figure 2.2: A typical 2-to-1 multiplexer represented as an AIG, where A and B are the two inputs, S is the selector, and O is the output: $O = SA + S'B$.

Create vertices: Initially, all POs and FFs are grouped by their common selection signals, while signals without selection conditions are put into individual groups. For example, in Figure 2.1, E, A, B and F_1, F_2 are put in individual vertices. Each selection signal occupies an individual vertex. Each signal must be in no more than one vertex.

Build dependency: Edges in the CGs represent data dependency in the sequential circuits. Starting with each signal covered by a vertex, we backtrack the original circuit for one time frame to find its set of supports (PIs or FFs), and then connect the vertices

containing those supports with edges to the target signal. If a vertex (a set of signals) is driven by a selection-edge (controlled by a selection signal), we need to find the two groups of supports, which determine the target value when the selection signal is 1 or 0, and then connect through on-edges and off-edges, respectively. After all vertices have been processed, a complete characteristic graph has been constructed.

Algorithm 2.1 Characteristic Graph Construction

Require: **Cir**: a gate-level sequential circuit with the sets of primary inputs **PI**, primary outputs **PO** and flip-flops **FF**

Ensure: $\mathbf{G} = (\mathbf{V}, \mathbf{E})$: characteristic graph for **Cir**, with the sets of vertices, **V** and edges, **E**

1: $\mathbf{V} = \emptyset$ and $\mathbf{E} = \emptyset$
2: $\mathbf{S} = recognize(\mathbf{Cir})$ ▷ S is the set of selection signals
3: **for all** PI_i in **PI do**
4: $\quad \mathbf{V} = \mathbf{V} \cup \{PI_i\}$
5: **for all** s_i in **S do**
6: $\quad \mathbf{V} = \mathbf{V} \cup \{s_i\}$
7: $\quad \mathbf{T} = findTarget(\mathbf{Cir}, s_i)$ ▷ **T** is the set of FFs and POs controlled by s_i
8: $\quad \mathbf{V} = \mathbf{V} \cup \{\mathbf{T}\}$
9: **for** ff_i in **PO** or **FF** not covered by any v in **V do**
10: $\quad \mathbf{V} = \mathbf{V} \cup \{ff_i\}$
11: **for all** v in **V do**
12: $\quad$ **if** v is controlled by selection signal s **then**
13: $\quad\quad connect(v, getVertex(\mathbf{V}, s), selection)$
14: $\quad\quad$ **for all** t covered by v **do**
15: $\quad\quad\quad \mathbf{Sup}_{on} = \mathbf{Sup}_{on} \cup backtrack(\mathbf{Cir}, t, s, on)$
16: $\quad\quad\quad \mathbf{Sup}_{off} = \mathbf{Sup}_{off} \cup backtrack(\mathbf{Cir}, t, s, off)$
17: $\quad\quad$ **for all** each support sup_{on} in $\mathbf{Sup}_{on}$ **do**
18: $\quad\quad\quad \mathbf{E} = \mathbf{E} \cup connect(v, getVertex(\mathbf{V}, sup_{on}), on)$
19: $\quad\quad$ **for all** each support sup_{off} in $\mathbf{Sup}_{off}$ **do**
20: $\quad\quad\quad \mathbf{E} = \mathbf{E} \cup connect(v, getVertex(\mathbf{V}, sup_{off}), off)$
21: $\quad$ **else**
22: $\quad\quad$ **for all** t covered by v **do**
23: $\quad\quad\quad \mathbf{Sup} = \mathbf{Sup} \cup backtrack(\mathbf{Cir}, t)$
24: $\quad\quad$ **for all** support sup in **Sup do**
25: $\quad\quad\quad \mathbf{E} = \mathbf{E} \cup connect(v, getVertex(\mathbf{V}, sup), on)$

Given a sequential circuit, **Cir**, with the sets of primary inputs **PI**, outputs **PO** and flip-flops **FF**, the algorithm for constructing the characteristic graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is shown in Algorithm 2.1. The function $recognize(\mathbf{Cir})$ at Line 2 is used to detect the MUX structures

in **Cir** and collect their set of selection signals **S**. Based on **S**, **PI**, **PO** and **FF**, vertices are created and added into **V** through Line 3 to Line 10. The function $findTarget(\mathbf{Cir}, s_i)$ is used to collect the set of signals, which are controlled by s_i. From Line 11 to 25, the vertices are connected according to the data dependencies in **Cir**. For the function *connect*(...) at Line 13, 18, 20 and 25, the first argument is the target vertex, the second is the support vertex, and the third is the edge type. The function $getVertex(\mathbf{V}, sup)$ returns the vertex which covers *sup*.

The function *backtrack*(...) at Lines 15, 16 and 23 goes back one time frame from target t and returns the supports (PIs or FFs) on the boundaries. If the third argument, s, and fourth argument, *on* or *off*, are specified, this function will only backtrack the specified input side of each target MUX, and collect the corresponding supports.

Once the characteristic graph is constructed, it is used to detect sequential redundancy candidates.

2.2 Sequential Redundancy and Clock-Gating

Given a sequential circuit, *sequential redundancy* refers to a signal that can be replaced by another signal or a constant value (1 or 0), while preserving sequential equivalence to the given circuit. Thus, the fanouts of such signals can be moved to other existing signals or to constants without changing the observed behavior.

Clock-gating synthesis can be based on either satisfiability or observability of signals. During this, additional control signals are created in a sequential circuit to reduce the frequency of updating the FFs. These extra signals, by definition, must be sequentially redundant in order to preserve sequential equivalence.

In this section, sufficient conditions are proposed for legal satisfiability and observability clock-gating on sequential circuits that result in sequential redundancies. Those sufficient conditions of legal clock-gating are formulated using CGs and then proved on the original circuits. Here we only demonstrate standard examples for this thesis, while the proposed theorems and algorithms, as well as proofs are detailed in [12].

2.2.1 Satisfiability Clock-Gating

Satisfiability clock-gating aims at turning off clocks for FFs when the input data is identical to the value in the previous time frame. One special case is that their support FFs remain at their previous states. Thus, the clock-gating is legal if all target FFs are guaranteed to update their states when their support FFs are updated. Otherwise, if none of the supports are updated, it is immaterial (don't care) if the target FFs are updated. Under proper initial states, the signal for disabling a clock is redundant and can be set to a constant.

The circuit in Figure 2.3 is an example of satisfiability clock gating: F_1 (initialized as 1) is sequentially redundant because it can be replaced by constant *1*, while the behavior observed at the outputs, Q_1 and Q_2, remains the same. The sequential redundancy, F_1 to

the MUXes in front of F_5 and F_6, is added to avoid updating F_5 and F_6 when their support FFs (F_2, F_3 and F_4) remain in the same states.

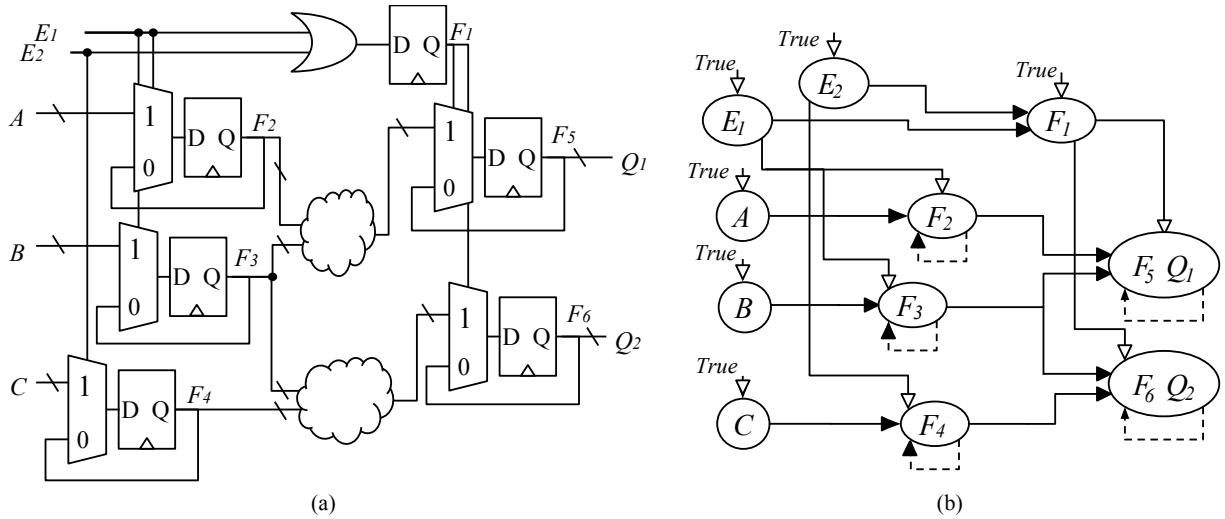


Figure 2.3: A clock-gated sequential circuit (a) with its characteristic graph (b). Each clock-gated FF is represented by a FF feeding back to a MUX controlled by a selection signal. The clock-gated FFs, F_2 to F_6, are updated only when their corresponding selection signals, E_1, E_2 or F_1 is 1; otherwise, they keep the same values as already saved in the corresponding FFs.

To verify if the clock-gating condition on F_5 and F_6 is legal, i.e. the connection from F_1 to the MUXes is sequentially redundant, a sufficient condition proposed by the CG method (detailed in [12]) is that both the following conditions are satisfied:

1. The initial value of F_1 is 1.
2. The LTL property
$$\mathbf{G}(E_1 \vee E_2 \Rightarrow \mathbf{X}F_1) \tag{2.1}$$
must hold all the time.

Proof: For time frame 0, because the initial state of F_1 is 1, it is safe to replace the selector of each MUX in front of F_5 and F_6 with constant 1. Then, the LTL property in Equation 2.1 guarantees F_5 and F_6 must be updated in the next time frame whenever any support FFs, F_2, F_3 or F_4 get updated ($E_1 = 1$ or $E_2 = 1$) in the current time frame. In other words, if all support FFs are unchanged from the previous clock cycle ($E_1 = 0$ and $E_2 = 0$), the input value for F_5 or F_6 is the same as the old one, and is immaterial if F_5 or F_6 is updated or not. Thus the selector can be 1 or 0 in those cases. By choosing it to be 1 in those cases as well, F_1 becomes constant 1, i.e. F_1 is stuck-at-1 sequentially redundant. **Q.E.D.**

Note that the above condition is not a necessary condition because even if F_2, F_3 or F_4 may change to new states, it might be that the combinational logic actually computes a next

state for F_5 or F_6 which is the same as its current state. In general, such a condition would be difficult to identify and/or expensive to implement, although an easy case would be a multiplier with an argument equal to zero.

The above example demonstrates the legality of a single time frame satisfiability, which can be formulated on the CG entirely. Moreover, given a target control signal C_{sat}, which is the updating condition of a set of FFs, the CG method can propose sufficient conditions of legal clock gating that go across multi-time frames. Based on the characteristics of the input circuit, there is a natural number N, such that for each k between 1 and N ($1 \leq k \leq N$), the CG method can propose a corresponding k-time frame sufficient condition for C_{sat} being stuck-at-1 sequentially redundant. That is, once any of the N conditions can be justified, C_{sat} is sequentially redundant.

2.2.2 Observability Clock-Gating

Observability clock-gating is used to disable updating FFs when these updates are not observable at the POs. In other words, the differences (updating or not updating) of gated FFs cannot be propagated to any POs before the FFs are updated. Recall the example in Figure 2.1, a set of FFs, F_2, can be gated by using observability as shown in Figure 2.4. If the connection from E to the MUXes before F_2 can be proved as sequential redundancy, the observability clock-gating applied on F_2 is legal and can be reduced.

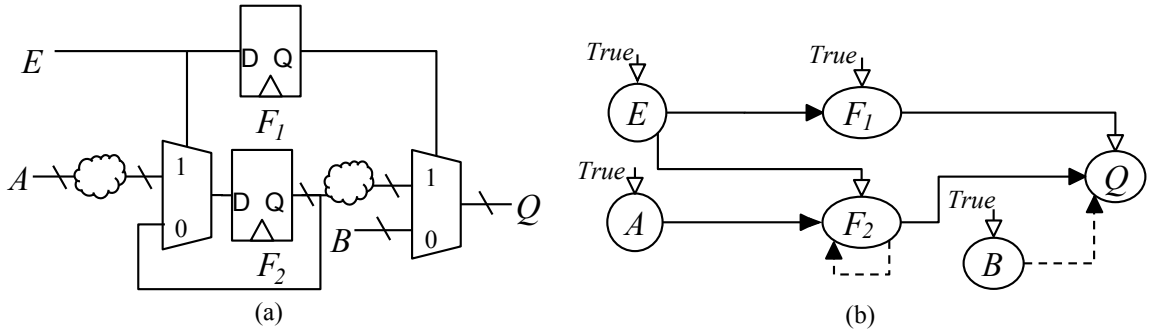


Figure 2.4: The revised circuit for Figure 2.1 with its characteristic graph, where the inputs, outputs and FFs are perfectly mapped to those in the golden design.

To verify if the clock-gating condition applied to F_2 is legal, a sufficient condition proposed by the CG method is that the following LTL property

$$\mathbf{G}(\mathbf{X}F_1 \Rightarrow E) \tag{2.2}$$

holds all the time.

Proof: The LTL property in Equation 2.2 guarantees F_2 is updated when its target Q depends on the value of F_2 in the next time frame. If E is 0, this property implies that Q is independent of F_2 in the next time frame due to $F_1 = 0$. This blocks any new state of

F_2 to be observed at any output. Thus the selector of the MUX in front of F_2 can be 0 or 1 in these cases. By choosing it to be 1, the selector is sequentially stuck-at-1 redundant. **Q.E.D.**

Similar to the satisfiability clock-gating cases described in Section 2.2.1, given a target control signal C_{obs}, which determines if a set of FFs is updated, we can also formulate sufficient conditions for observability clock-gating across multiple time frames, which justify the target signal, C_{obs}, as sequential stuck-at-1. According to the properties of the input circuit, there is a natural number N, such that the proposed CG method can formulate a corresponding k-time frame condition for each k between 1 and N. If one of the N conditions can be proved, C_{obs} is sequentially redundant.

2.2.3 Using the Characteristic Graph

A characteristic graph exposes the essential properties of the corresponding circuit, including signal dependency and control signals. It contains information required to formulate properties for the legality of a clock-gated circuit. The on-edges and off-edges of a CG connect the targets and supports across each time frame, while each selection-edge indicates the updating condition for a group of signals. As discussed in Section 2.2.1 and 2.2.2, sufficient conditions for the legality of satisfiability and observability clock-gating can be formulated with LTL properties. Each selection signal associated with a proved condition has its corresponding signal in the original circuit which can be replaced by 1.

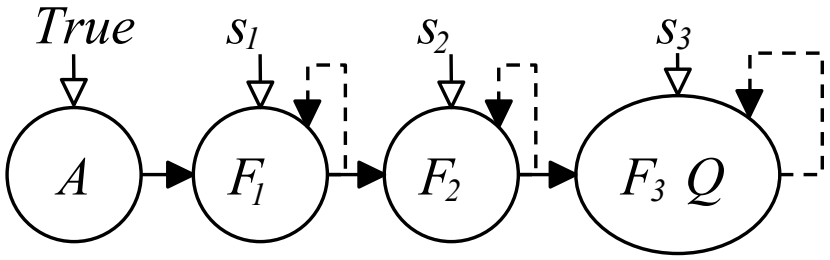


Figure 2.5: A CG with three stages of FFs. The vertex A is a PI, where the selection-edge is driven by $True$. The signals, s_1, s_2 and s_3 represent the updating conditions for FFs F_1, F_2 and F_3, respectively.

Consider the CG in Figure 2.5, where F_1, F_2 and F_3 are FFs, Q is a PO and s_1, s_2 and s_3 are their corresponding control signals. Control signal s_1 only can be from an observability clock-gating case, where a sufficient property for s_1 being sequential stuck-at-1 is $\mathbf{G}(\mathbf{X}s_2 \Rightarrow s_1)$. The CG method cannot formulate observability clock-gating conditions across more than one time frame for F_1 because there is a sequential loop due F_2.

For s_2, it can be a satisfiability clock-gating case with the property $\mathbf{G}(s_1 \Rightarrow \mathbf{X}s_2)$ and s_2 is 1 in the first time frame, or else an observability clock-gating case with the property $\mathbf{G}(\mathbf{X}s_3 \Rightarrow s_1)$.

s_3 can only be a satisfiability clock-gating case, and the sufficient condition requires (1) $\mathbf{G}(s_2 \Rightarrow \mathbf{X}s_3)$ holds and (2) the initial condition of s_3 is 1. The proposed algorithm terminates when trying to go across F_2, because its selection-edge is not driven by $True$.

The sufficient conditions proposed by the CG method can be proved on circuits by general hardware model checkers. As implied by the LTL safety properties, only the fanin cones of those control signals need to be considered by model checking, and hence irrelevant combinational logic will be excluded automatically by state-of-the-art model checking methods. Thus, when model checking the generated properties on the original circuit, the problem size is effectively much less.

There are some limitations on finding sequential redundancy on CGs. Currently each vertex on a CG covers all signals controlled by the same selection signal, so some sequentially redundant points may be missed. For example, if a set of FFs is clock-gated by a satisfiability condition with a control signal, s, while another set of FFs is clock-gated by an observability condition also with s, both cases could be legal but cannot be proved by the current formulation. This issue can be resolved by separating all FF into different vertices, but that may result in duplicate properties to be proved.

2.3 Overall Flow

Given two sequential circuits, golden (**G** and clock-gated, **R**), with a mapping correspondence between PIs and POs of the circuits, SEC verifies if the output sequences are identical when the same input sequences are applied. If **R** is known to be a clock-gated version of **G**, instead of applying general SEC methods, the difficulties of SEC can be reduced using the CG approach. Here the overall flow of this method is outlined.

As the use-model, it is assumed that the golden model **G** may be already be clock-gated in the RTL, possibly manually by the designer. Therefore in comparing **G** and **R**, we propose to reduce them both with the CG method and apply the proved redundancies to get **G'** and **R'**, which will be compared.

Algorithm 2.2 outlines an algorithm to perform SEC between two circuits, **G** and **R**, and report if they are sequentially equivalent(**EQ**) or not(**NON-EQ**).

The function $CEC(...)$ at Line 1 performs general combinational equivalence checking between corresponding signals for each pair and returns a set of unproved pairs. The function $charGraph(...)$ returns the corresponding characteristic graph for the specified circuit.

The loop between Lines 7 and 15 verifies each candidate and revises the corresponding CG according to the proved redundancy one by one. At Line 7, the function $comparison(...)$ analyzes these unresolved pairs and proposes a candidate for sequential redundancy by comparing their CGs. The function $defineProperty(...)$ applies the ideas in Section 2.2 for each target signal, and generates the set of corresponding LTL properties (**P**) to be proved. Notice that not only **R** but also **G** can contain sequential redundancies, so from Line 8 to 11, properties are defined for C_G or C_R, respectively.

Algorithm 2.2 Proposed SEC Flow

Require: **G** and **R**: two circuits with mapped PIs, POs and FFs.
Ensure: SEC result: **EQ** or **NON-EQ**

```
1: nonEQ = CEC(G, R)
2: if nonEQ = ∅ then
3:     return EQ
4: C_G = charGraph(G)
5: C_R = charGraph(R)
6: proved = ∅
7: while candidate = comparison( nonEQ, C_G, C_R) do
8:     if candidate ∈ G then
9:         P = defineProperty(candidate, C_G)
10:    else
11:        P = defineProperty(candidate, C_R)
12:    proof = multiProve(P)
13:    if isLegal(proof) then
14:        revise(candidate, C_G, C_R)
15:        proved = proved ∪ candidate
16: (G', R') = simplify(proved, G, R)
17: return SEC(G', R')
```

The function *multiProve*(...) at line 12 verifies a circuit with multiple outputs. Given a set of properties **P**, *multiProve*(**P**) verifies all properties simultaneously, and then returns the resulting set **proof**, which lists both proved and disproved properties. At Line 13, *isLegal*(**proof**) analyzes the result and determines if **candidate** is sequentially redundant using theorems stated in [12]. When any of the sufficient conditions proposed by the theorems is satisfied, the candidate is proved. Due to the possible dependencies among candidates, the corresponding CG should be revised using proved redundancies before the next run. All proved candidates are used to simplify **G** and **R** into **G'** and **R'** at once (Line 16).

Finally, we perform *SEC*(**G'**, **R'**) to check if **G'** and **R'** are sequentially equivalent. If **G'** and **R'** are proved to be **NON-EQ**, *SEC*(**G'**, **R'**) can return a counter-example, which is also valid for **G** and **R**. Therefore, the proposed algorithm can provide counter-examples to users for debugging.

A limitation of the structural approach used is that previous synthesis done on **R** might have destroyed some of the MUX structures in the circuit. Then some MUXes might not be recognized during the CG construction and thus the algorithm might fail to identify all sequentially redundant points. In any case, as much redundancy as possible is identified. The resulting simplification might be enough for SEC to still be able to prove the property.

To show how this algorithm works, consider the circuits in Figure 2.1 and 2.4 as **G** and **R** respectively. At line 1, only the pair for F_2 fails CEC and is added into **nonEQ**. At Line

7, E, the selection signal of F_2, is a candidate of sequential stuck-at-1 redundancy. Since it can only be observability clock-gating, the corresponding property, $\mathbf{G}(\mathbf{X}F_1 \Rightarrow E)$, is created and proved. Then **R** is simplified into **R'** by replacing the selector of the MUX of F_2 by constant 1, while **G'** is the same as **G**. Therefore **G'** and **R'** become identical to Figure 2.1, and can be proved equivalent by SEC easily. Finally, the algorithm returns that **G** and **R** are sequentially equivalent.

2.4 Experimental Results

We compare the CG method against two state-of-the-art methods.

1. Model checker *super_prove* [7]. This is a general purpose gate-level model-checker, which won the single-output track in the Hardware Model Checking Competition 2014 (HWMCC'14) [17].

2. *Absec*, a command implemented in ABC [8] which uses the algorithm in [40]. This is specific to checking clock-gated circuits. It unrolls the circuit a determined number of clock cycles and uses CEC to prove the desired result.

The CG method, $SEC(\mathbf{G}, \mathbf{R})$ is implemented in ABC. The $multiProve(...)$ function used is *multi_prove*, which won the multi-output track in HWMCC'13 [16] (not held since). We also apply *super_prove* to the final SEC between **G'** and **R'**.

All experiments were performed on a 16-core 2.60GHz Intel(R) Xeon(R) CPU with a 1500 second time limit. The example circuits were clock-gated manually at the RTL, and then synthesized into AIGs to create **R**. Each input for *super_prove* is a multi-output miter between a golden design (**G**) and its clock-gated circuit (**R**). The inputs for *Absec* and the CG method are **G** and **R** are given separately before mitering.

2.4.1 Performance for General Clock-Gated Cases

First the applicability and efficiency of the three methods applied for general clock-gated cases are compared. Table 2.1 lists five cases with their circuit sizes, along with how they were clock-gated. The first three circuits were downloaded from OpenCores [33](**G**) and modified (**R**) manually, while the last two cases were created manually (both **G** and **R**) for this comparison. The CG results are separated into two stages: *Simplify* includes Line 1 to 16 in Algorithm 2.2, while *SEC* refers to the final SEC at Line 17.

Because *Absec* is implemented in ABC only for observability clock-gating, it is not applicable to the satisfiability clock-gating cases (indicated with N/A in Table 2.1).

As can be seen in Table 2.1, the proposed CG method significantly outperforms the other two methods. Although the general model checking method *super_prove* can prove some of the satisfiability and observability clock-gating cases, it requires much more run time than the specialized methods. We see that *Absec* can reduce the sequential complexity and prove

Table 2.1: Comparisons with *super_prove* and *Absec* on three OpenCores [33] cases and two synthetic cases.

Circuit	Clock-Gating Techniques	AND #	FF #	*super_prove* (s)	*Absec* (s)	CG method(s) *Simplify*	CG method(s) *SEC*
aes.Round	Observability	125k	645	208.2	5.31	0.67	2.95
Md5Core	Satisfiability	95k	40k	80.33	N/A	0.92	7.92
CLA_fixed	Observability	3k	97	T.O.	1169.78	0.66	1.97
Synthetic_1	Observability	4k	73	T.O.	166.28	0.56	0.23
Synthetic_2	Both	877	74	177.06	N/A	0.65	0.43

equivalence for backward cases. The final two columns of Table 2.1 show that the CG method is very efficient in both the redundancy finding phase (*simplify*) and the final SEC proof after the redundancies have been removed.

2.4.2 Comparisons of Scalability

The second set of experiments compares the scalability of the three above methods by applying them to the same design (*qmult* taken from OpenCores [33]), but with varying bit-widths (8-16). As the widths increase, the combinational part gets more complex. All of these cases were modified using only observability clock-gating in order to allow *Absec* to be applied.

Table 2.2: Comparisons with *super_prove* and *Absec* on *qmult*, a design from OpenCores [33], with varying bit-widths.

Circuit	AND #	FF #	*super_prove* (s)	*Absec* (s)	CG method(s) *Simplify*	CG method(s) *SEC*
qmult_8	487	25	0.35	0.90	0.58	0.33
qmult_9	632	28	4.09	3.61	0.64	0.34
qmult_10	791	31	23.14	10.50	0.65	0.34
qmult_11	964	34	61.28	98.28	0.65	0.35
qmult_12	1151	37	113.92	153.96	0.65	0.35
qmult_13	1352	40	T.O	229.61	0.66	0.36
qmult_14	1567	43	T.O.	1308.66	0.65	0.37
qmult_15	1796	46	T.O.	425.79	0.54	0.36
qmult_16	2039	49	T.O.	620.23	0.65	0.37

Table 2.2 shows that as the complexity of the circuit increases, the runtimes of *super_prove* and *Absec* increase sharply. In contrast, the CG method is not affected by the increased complexity because the complicated combinational logic is effectively excluded by the CG method.

2.5 Summary

This chapter presented a novel SEC method for clock-gated circuits. The proposed method is based on constructing a characteristic graph (CG) to model only essential signals of a circuit deemed necessary to prove equivalence. It uses control graphs, CGs, to formulate sufficient properties for sequential redundancies. These properties are proved and used to simplify the circuit after which SEC becomes easy. The experimental results show that the CG method is scalable and effective, and substantially outperforms existing techniques.

The contributions of this chapter are summarized below:

1. We formulate graph representations (CGs) of AIG circuits, (**G** or **R**). The CG expresses the essential underlying control structure of the circuit. An algorithm is provided that constructs a CG from the corresponding circuit.

2. We show how to use the CG to formulate LTL properties (**P**) about the flow control of the original circuit. These properties can justify both satisfiability and observability clock-gating conditions spanning several time cycles. **P** can be model checked on the circuit easily because their supports rely only on signals highlighted in the CG.

3. We prove that each property in **P**, if proved, implies that an associated control signal directing the flow is sequentially stuck-at-1 redundant. This redundancy is used to simplify **R** into **R'**. Similarly, **G** is simplified to **G'**.

4. **R'** and **G'** are SEC checked. Since **R'** and **G'** are typically structurally more similar to each other, the subsequent model checking becomes very efficient.

5. This method was implemented and applied to a number of academic and industrial clock-gated circuits to check (SEC) them against the original circuits. Experimental results show that this method is much more efficient than existing methods for performing SEC on clock-gated circuits.

The proposed SEC flow works well when only the control condition over FFs is used for clock-gating synthesis. However, there are some drawbacks of this method:

1. The MUX structures for clock-gating might be destroyed by synthesis applied after clock-gating. Unrecognized clock-gating conditions might cause the final SEC to be too difficult and cause the verification of legal cases to fail.

2. The construction of CGs assumes there is at most one MUX in front of each FF. In real cases, multiple clock-gating conditions can be applied together, resulting in a series of MUXes. Thus, the CG method cannot be used to address a case where multiple clock-gating conditions are applied to the same set of gated FFs.

3. The CG construction flow does not distinguish FFs gated by the same control value but under different data dependencies and gating conditions. This issue could result in a false negative when proving legal clock-gating conditions.

4. The proposed CG method only checks the cases where the control signals are sequential redundancies. It is possible that another signal in the fanin cones of controls is redundant but the CG method cannot identify it and reduce it. Hence the clock-gating condition remains and the final SEC is still challenging.

5. The CG method only considers the cases where only control signals of MUXes in front of FFs are used to perform clock-gating. This might exclude some legal clock-gating conditions that rely on more sophisticated analysis of control logic.

6. The CG method cannot handle conditions with sequential loops properly. It is incapable of analyzing satisfiability clock-gating conditions with branches across time frames.

All issues above can be resolved by extracting all control paths and analyzing data dependencies in detail. To recognize more control logic in circuits, we propose to use a functional approach presented in the next chapters and to analyze data dependencies thoroughly using a "dependency graph" (DG). While this might be more computationally expensive, it does lead to a more complete theory of clock-gating and in fact to automatic synthesis methods. It is still efficient and can be combined with the methods of this chapter to lead to an overall very efficient hybrid CG/DG method.

Chapter 3

Transparent Logic in Hardware Designs

In hardware, control logic regulates the data flow and dictates circuit functionalities. A logic that simply moves data from one part of a circuit to another without modifying it can be referred to as *transparent* logic. Another category of logic *transforms* data by some word-level operator, e.g. a bit-vector operator defined in Verilog [34]. A third category, *control*, determines which data is moved and when, or which operation is applied and when.

The basic example of transparent logic is a multiplexer (MUX) structure, which selects from several data signals and forwards it unaltered towards the outputs. Efficient identification of MUXes can be performed over gate-level circuits using structural matching, but this can be unreliable, especially if synthesis has been applied.

In this chapter, we focus on functional approaches which do not depend on the actual gate-level structure of the circuit. These can augment structural methods and provide a much more reliable technique as we show in the experiments.

In general, identifying transparent logic is widely applicable to other applications like reverse engineering but we develop it here to be used in Chapter 4 to identify controlled dependencies for constructing dependency graphs for synthesizing and verifying clock-gating in circuits.

3.1 Introduction

Functional methods, which rely only on functional dependencies, have been used generally to augment structural approaches. Examples are:

- Li and Subramanyan et. al. [27, 45] identified internal words based on *bitslice aggregation* (functional approach) and *shapehashing* (structural approach). The candidate words found were used as boundaries of operators for further recognition. However, they did not recognize found boundaries as inputs or outputs of operators before identifying functionality.

- Li et. al.[29, 27] identified functional operators in gate-level circuits, based on an existing library of blocks. Word-level information at the primary inputs was assumed available, but in many applications such information is not given.

- Sterin et. al. [44] extracted word-level operators functionally, given a library of operators and a slice of logic containing inputs and outputs of such operators. The possible location and ordering of the inputs and outputs of an operator and word-level information were not required. However, the slice of logic cannot include transparent logic, based on the algorithm.

We present methods to identify *functional transparent logic*. This is inherited from *functional isomorphism*. Using this approach, an algorithm to identify words, word-level operator boundaries and control logic in gate-level circuits is proposed and applied to a variety of test cases. Indeed, we can rewrite a gate-level circuits hierarchically (i.e. hierarchical Verilog format) with the recognized logic as sub-circuits. Once operator boundaries are (roughly) located, techniques like [44] can be used to identify the precise location of the operators and their functionalities.

The remainder of this chapter is organized as follows. Section 3.2 introduces *functional* isomorphism. In Section 3.3, we describe the definition and propagation of *transparent* logic. Proposed algorithms for identifying transparent logic are given in Section 3.4. Some practical challenges of identifying transparent logic are discussed in Section 3.5. Experimental results are shown in Section 3.6, while Section 3.7 concludes this chapter.

3.2 Overview

Roughly, a transparent path in a circuit has width n and a set of controls $s = \{s^i\}$, which when evaluated appropriately at a minterm $s = \{s^i\} = \{m_{s^i}\}$, moves a data-word (width n) from the beginning of the path to the end. Different minterms for s select different input words to be transported. Such paths can fork and join in the circuit, and can begin and end at a set of inputs, outputs or internal signals. A path is *maximal* if there is no transparent path that can extend it. The terminals of maximal transparent paths are of interest because they likely delineate the input or output of an operator, e.g. an arithmetic function.

A sink terminal can have many source terminals. Each data signal at a sink terminal is a Boolean function of a) data signals $D_k = \{d_k^j\}$ at the source terminals and b) the set of associated controls $\{s^i\}$ of transparent segments of any path from source to sink. Such a set of functions at a sink forms an NPN equivalence class (or equivalently an NPN isomorphism class). The isomorphism between the inputs of any two signals f_p and f_q (where $p, q \in [1, n]$) in the sink terminal is $d_p^j \leftrightarrow d_q^j$, i.e. different bit positions in the same data word are isomorphically mapped to each other, while control signals s^i are isomorphically mapped into themselves. It is possible that some bits of a terminal have been inverted, hence NPN equivalence is considered in the subsequent discussions.

Thus, the outputs of a transparent path are a subset of an NPN isomorphism class.

3.2.1 NPN Isomorphism

Two graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are *isomorphic*, if there exists a bijective mapping, $\mathbf{M}_{12}$: $V_1 \rightarrow V_2$, such that any two vertices u and v are adjacent in G_1, if and only if $\mathbf{M}_{12}(u)$ and $\mathbf{M}_{12}(v)$ are adjacent in G_2 [1]. Two circuits, C_1 and C_2, are isomorphic to each other if their logic gates and connections form two isomorphic graphs, while any gate g of C_1 and the mapped gate $\mathbf{M}_{12}(g)$ in C_2 are the same type. The relation between C_1 and C_2 is called *structural isomorphism*, which has been applied to reverse engineering [15]. In contrast, *NPN isomorphism* is a *functional isomorphism*, which is a relation between two signals in a circuit.

A signal f in a circuit, supported by a set of other signals, S_f, is a Boolean function of these inputs: $f : \mathbf{B}^{|S_f|} \rightarrow \mathbf{B}$, for $\mathbf{B} = \{0, 1\}$.

In the following sections, for a Boolean variable x_i with its polarity p_i, $(x_i)^{p_i}$ represents the function: $p_i = 0 \rightarrow (x_i)^{p_i} \equiv x_i$ and $p_i = 1 \rightarrow (x_i)^{p_i} \equiv inv(x_i)$.

Definition 3.1 A pair of Boolean functions $f(x_1, \ldots x_n)$ and $g(y_1, \ldots, y_n)$ are *NPN isomorphic*[1], if there exists a permutation π of size n and polarities p_{out} and $\{p_1, \ldots, p_n\} \in \mathbf{B}^n$ such that

$$f(x_1, \ldots, x_n) = g^{p_{out}}(x_{\pi(1)}^{p_1}, \ldots, x_{\pi(n)}^{p_n}) \tag{3.1}$$

i.e., g can be made equivalent to f by selectively negating inputs, permuting inputs, and negating the output. The implied isomorphic mapping between the supports of g and f is $\{y_i, x_{\pi(i)}^{p_i}\}$ and p_i is said to be the relative polarity between inputs y_i and $x_{\pi(i)}$.

A set of signals in a circuit, in which every pair is functionally NPN isomorphic is called an *NPN isomorphism class*.

3.2.2 Composition of NPN Isomorphism

Although improved methods for computing NPN equivalence can be found in Soekin et. al. [43], this calculation can still be time-consuming. This effort can be reduced immensely by proving NPN isomorphism on smaller logic blocks and then composing proved classes to obtain larger ones. Larger classes help extend paths of transparency (discussed in Section 3.3) in a circuit and to more reliably find transparency boundaries, and hence the input/output boundaries of word-level operators.

The following discussion provides details on when compositions lead to larger NPN isomorphisms.

Definition 3.2 (*polar consistency*) Let $(f(s), g(t))$ be a pair of NPN isomorphic functions with sets of supports $s = \{s_i\}$ and $t = \{t_j\}$, respectively. Suppose each pair of mapped input supports $s_i \leftrightarrow t_j$ are NPN isomorphic functions, i.e. $s_i(x)$ is NPN isomorphic to $t_j(y)$. Let p_{out}^{ij} be the relative output polarity between $s_i(x)$ and $t_j(y)$, and p_{ij} be the relative

[1] or Negation-Permutation-Negation (NPN) equivalent

input polarity between inputs s_i and t_j in the NPN isomorphism between $f(s)$ and $g(t)$. The compositions $f(s(x))$ and $g(t(y))$ are *polar consistent*, if $p_{out}^{i\pi(i)} = p_{i\pi(i)}$, where π is the permutation in the isomorphism mapping of $(f(s), g(t))$.

Theorem 3.1 The compositions of $(f(s(x)),\ g(t(y)))$ are polar consistent if and only if $f(s(x))$ and $g(t(y))$ are NPN isomorphic.

3.3 Extending Transparent Logic

As already stated, the identification of maximal *transparent logic* can be used to identify input/output boundaries of arithmetic operators.

3.3.1 Transparent Words

Intuitively, a *transparent word* is a set of signals, $\{w^k\}$, with supports, $\{S^k\}$, where under some evaluation of $\cap^k S^k$ (*common control*), $\{w^k\}$ is equivalent to a subset (*data-word*) of $\cup^k S^k$. In other words, the control evaluation makes the word transparent from some input data-word.

Example: An m-bit word from a set of 2-to-1 multiplexers (MUX) controlled by the same selector signal s,

$$C[m-1:0] = s?A[m-1:0] : B[m-1:0], \tag{3.2}$$

comprises a transparent word C, where $\forall j \in [0, m-1]$, $(C[j] = sA[j] + s'B[j])$. For this case, word C is transparent from word A or word B, depending on the value assigned to s.

Definition 3.3 Functions $W = \{w^k | k \in [1, m]\}$ of an NPN isomorphism class comprise an m-bit *transparent word*, if:

1. Each function $w^k : \mathbf{B}^{S^k} \rightarrow \mathbf{B}$, has support $S^k = (Control, Data^k)$, i.e. *Control* is the set of common signals, and each bit of *Control* is isomorphically mapped into itself.

2. Formula 3.3 below is *True*, where m_c is a minterm of *Control*, $\equiv$ denotes functional equivalence, and $w^k_{m_c}$ denotes the co-factor of function $w^k(Control, Data^k)$ with respect to m_c.
$$(\exists_{m_c} \forall_k \exists_{d_i^k \in Data^k} \exists_{p_i^k} (w^k_{m_c}(Data^k) \equiv (d_i^k)^{p_i^k})). \tag{3.3}$$

3. For any $(w^x, w^y) \in W$, the associated isomorphic support mapping $\mathbf{M}_{xy}$, satisfies $\mathbf{M}_{xy}(Data^x) = Data^y$.

Thus a transparent word W is conditionally (by m_c) equivalent to an input data word $[(d_i^1)^{p_i^1}, \ldots, (d_i^m)^{p_i^m}]$. Based on the above definition, the vector of conditionally equivalent

data support bits that have a common condition m_c, $D_i = \{d_i^1, \ldots, d_i^m\}$ is called an *input word*.

Given a transparent word, $W = \{w_k\}$, with the corresponding support partitions $\{(Control, Data^k)\}$, the entire support set of W can be partitioned into **Control** and $\mathbf{Data}^W = \bigcup^i D_i$. The definition of transparent words can be restated as follows:

Definition 3.4 A transparent word W is a set of NPN isomorphism functions supported by control **Control** and data $\mathbf{Data}^W = \bigcup^i D_i$, such that the following formula is *True*:

$$\forall_{D_i \in \mathbf{Data}^W} \exists_{m_c} \exists_{P_i} (W_{m_c}(\mathbf{Data}^W) \equiv (D_i)^{P_i}), \tag{3.4}$$

where P_i is the set of polarity bits for D_i.

Although, for an input word D_i, there could be multiple minterms of $Control$ satisfying Formula (4), the assignments of $m_c \in Control$ for different D_is are disjoint.

Example: Consider Equation 3.2: for each $C[j]$, the support set $\{s, A[j], B[j]\}$ can be partitioned into $Data^j = \{A[j], B[j]\}$ and $Control = \{s\}$, such that $(s = 1) \Rightarrow (C[j] = A[j])$ and $(s = 0) \Rightarrow (C[j] = B[j])$. Hence a common (control) assignment applied to all bits of the transparent word, makes them simultaneously transparent from the corresponding supports. The supports of C can be partitioned into $\mathbf{Data}^C \equiv \{A[m-1:0], B[m-1, 0]\}$, and $\mathbf{Control} \equiv \{s\}$.

Since negations of some bits of transparent words might occur during synthesis, it seems reasonable to consider the logic still as "transparent". Note that in the example: $C[j] = sA[j] + s'B[j]$ the negation of bit $C[j]$ can be done by negating the data inputs, $A[j]$ and $B[j]$:

$$\begin{aligned} inv(C[j]) &= inv(sA[j] + s'B[j]) \\ &= s\ inv(A[j]) + s' inv(B[j]). \end{aligned} \tag{3.5}$$

but C (with some phase changes) can still be considered transparent from A and B because the assignments to the control bits are unchanged.

In Section 3.3.2 and 3.3.3, as we compose transparent sections to form a larger transparent path, we will need to resolve cases where only some bits of a transparent word are negated. However, for composing transparencies to find larger ones i.e.$f(g(x))$ it is required that the polarities of the inputs to the fanout logic (f) and outputs (g) of the fanin logic are consistent as defined in **Definition 3.2**. This can be done by negating some of the inputs (x) of the path (using NPN isomorphism) to get a polarity of (g) that is compatible with the input polarity of (f).

Theorem 3.2 Given a transparent word W, the negation of any output bit w^k can be effected by negating the corresponding input data support bits, without changing any control assignment.

The upshot is that when finding another transparent section of logic and composing it to extend a transparent path, this can *always* be done simply by negating the inputs to get compatible polarities at the point of composition.

3.3.2 Composition of Transparency

Similar to the composition of NPN isomorphism, larger transparent functions are frequently created by composing smaller transparent blocks.

Example: In Figure 3.1, word C is transparent from A and B under the control of s_1, while a second transparent block consists of word E, transparent from C and D under the control of s_2. Thus $(s_1 = 1, s_2 = 1) \rightarrow E \equiv A$, while $(s_1 = 0, s_2 = 1) \rightarrow E \equiv B$ i.e. transparency of E from A and B is obtained by composing of smaller transparent blocks. If some bits of C are negated before feeding into the MUXes controlled by s_2, the composition can be done by pushing the negation to the corresponding bits of A and B to maintain polar consistency.

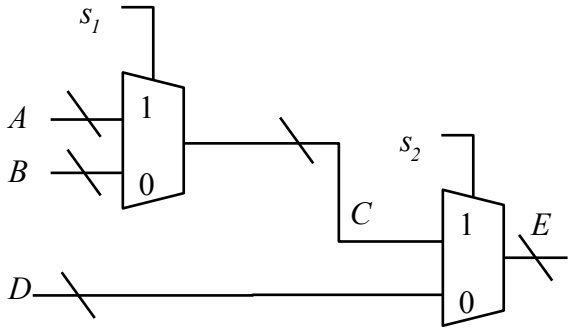


Figure 3.1: A transparent word can be implemented by composing smaller transparent words.

Definition 3.5 Let $\mathbf{W} = \{W^k(X)|k = [1, n]\}$ be a set of n m-bit transparent words, and let $Y = \{y^j|j = [1, m]\}$ be another transparent word with support $Data^Y = \mathbf{W} \bigcup V$ and common control $Control^Y$. Suppose each input word of Y is exactly one transparent word in $\mathbf{W}$ or one word in V. The set of compositions,

$$Z = \{z^j\} = \{y^j(\mathbf{W}(X), V, Control^Y)\} \tag{3.6}$$

forms a *compound word*, and is denoted as $Z = Y \circ \mathbf{W}$.

Theorem 3.3 Assume Y is a transparent word and $\mathbf{W}$ is a set of transparent words. Let $\{\alpha_i^k\}$ be the set of minterms of $Control^k$, which enable W^k to be transparent from an input word $x_i^k \in Data^k$, and $\{\beta^k\}$ be the set of minterms of $Control^Y$ for $(Y \equiv W^k)$. Using the notation:

$$Control^Z = Control^Y \bigcup [\cup^k Control^k],$$
$$Data^Z = V \bigcup [\cup^k Data^k],$$

a compound word, $Z \equiv Y \circ \mathbf{W}$ is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha}^{k_i}\} \cap \{\hat{\beta}^k\} \neq \emptyset) \tag{3.7}$$

is *True*, where $\{\hat{\alpha}^{k_i}\}$ and $\{\hat{\beta}^k\}$ are $\{\alpha_i^k\}$ and $\{\beta^k\}$ extended to cubes of the larger space of $Control^Z$, respectively.

Proof:

1. Based on Theorems 1 and 2, Z can be an NPN isomorphism class by flipping the polarities of W^kwhenever its output polarity is not compatible with the input polarities of y^k.

2. Because Y is a transparent word, for each input word in V, there must exist an assignment of $Control^Y$ to enable the transparency from V.

3. Conditions satisfying Formula 3.7 imply that for each input word x_i^k of W^k, there exists an assignment of $Control^Z$ such that a) W^k is transparent from x_i^k, b) Y is transparent from W^k, and c) Y is transparent from x_i^k. Therefore, $Z \equiv Y \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

3.3.3 Propagation of Transparency

Example: Figure 3.2 illustrates how a longer transparency can be obtained from non transparent sections of logic. C is transparent from A when $s_1 = 1$, and D is transparent from B when $s_2 = 1$, but the logic block from C and D to E is not transparent (there is no common control support for each bit of E). However, E is transparent from A when $(s_1 = 1, s_2 = 0)$, while $(s_1 = 0, s_2 = 1)$ makes E transparent from B.

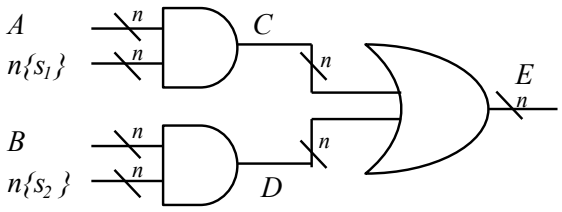


Figure 3.2: A longer transparent word may be obtained from smaller transparent words and an NPN isomorphism class.

When a transparent function block is obtained partly non-transparent sections, it is called *propagation of transparency*. The conditions when this can happen are stated in the following.

Definition 3.6 (*proceeding word*) Let $\mathbf{W}$ be a set of n m-bit transparent words, and let $Y(\mathbf{W}) = \{y^j(\mathbf{W})\}$ be an NPN isomorphism class. Suppose each y^j is supported by exactly one bit of each W^k, and the isomorphically mapped supports of y^j are always from the same word of $\mathbf{W}$. We say that Y is a proceeding word of $\mathbf{W}$ and denote this by $Z = Y \circ W$.

Theorem 3.4 Assume Y is a proceeding of $\mathbf{W}$ as in **Definition 3.6** and the supports of W^k are $supp(W^k) = (Control^k, Data^k)$. Let $\{\alpha_i^k\}$ be the set of minterms of $Control^k$ which cause $(W^k \equiv x_i^k)$, and $\{\beta^k\}$ be minterms of $\cup^k Control^k$ which cause $(Y \equiv W^k)$. Using the notation $Control^Z = \cup^k Control^k$, and $Data^Z = \cup^k Data^k$, a proceeding word, $Z \equiv Y \circ \circ \mathbf{W}$, is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha}_i^k\} \cap \{\beta^k\} \neq \emptyset), \tag{3.8}$$

where $\{\hat{\alpha}^{k_i}\}$ refers to $\{\alpha_i^k\}$ extended to cubes of $Control^Z$.

Proof:

1. Similar to the proof of Theorem 3, Z can be an NPN isomorphism class by flipping the polarities of W^k if needed.

2. For each input word x_i^k in $Data^Z$, Formula 3.8 implies that there exists an assignment of $Control^Z$, such that $W^k \equiv x_i^k$, $Y \equiv W^k$, and thus, $Y \equiv x_i^k$, implying $Z \equiv Y \circ \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

Example: In Figure 3.2, $\mathbf{W} = (C, D)$ and $\{\hat{\alpha}_1^k\} = s_1 s_2 + s_1 \overline{s}_2$ makes C transparent from A, while $\{\hat{\alpha}_2^k\} = s_1 s_2 + \overline{s}_1 s_2$ makes D transparent from B. $\{\beta^k\} = s_1 \overline{s}_2$ $(\overline{s}_1 s_2)$ causes $E \equiv C$ $(E \equiv D)$. Note that $\{\hat{\alpha}_1^k\} \cap \{\beta^k\} = s_1 \overline{s}_2 \neq \emptyset$ and $\{\hat{\alpha}_2^k\} \cap \{\beta^k\} = \overline{s}_1 s_2 \neq \emptyset$. Thus the conditions for propagation of transparency are met, and therefore E is transparent from A and B.

3.4 Transparency Identification

The functional approach proposed for transparency identification relies only on dependencies among signals. It can be used to complement a structural approach, leading to a method that is still efficient but with more reliable results.

In general, we want to identify transparent logic anywhere it occurs in the circuit - from inputs to internal words (forward), from internal words to outputs (backward), and between internal words. A general problem is formulated and solved in this paper: Given a combinational circuit, find:

1. all (disjoint) transparent logic blocks, specified by corresponding support and output boundaries,

2. transparent words on each output boundary,

3. input words on each support boundary, and
4. assignments of **Control** for moving each input word to the output transparent word.

The proposed algorithm can be decomposed into four parts:

1. collect candidate controls,
2. find transparent words controlled by one signal,
3. find proceeding words and
4. rearrange proved words.

3.4.1 Find Transparency with Given Controls

A sub-process, *findTransparency(...)*, is developed to identify all transparent words controlled by a set of signals. Algorithm 3.1 shows the proposed algorithm for this sub-process.

Algorithm 3.1 Find Transparency

Require: Circuit, **Controls**
Ensure: TransparentWords = (Inputs, Outputs, Minterms)
1: **Minterms** = *enumerateControls*(**Controls**)
2: **for all** *minterm* in **Minterms do**
3: **Candidates** = *applyMinterm*(*minterm*, **Circuit**)
4: **TransparentWords** = *analyzeWords*(**Candidates**)
5: *splitWords*(**TransparentWords**) **return TransparentWords**

In Line 1, the function *enumerateControls(...)* enumerates all possible minterms for the input control set. For each minterm, the function *applyMinterm(...)* finds the co-factor of the input circuit. It returns a set of conditionally transparent paths, **Candidates**, where each sink signal is functionally equivalent to the corresponding source. The output of each transparent path must be supported by all signals in **Controls**. Note that a sink signal can be driven by multiple transparent paths controlled by different minterms of the same set of controls.

In Line 4, the function *analyzeWords(...)* examines all candidate paths and merges paths with the same sink signal. Then this function partitions those sink signals into several transparent words. To match the requirement of NPN isomorphism, in each transparent word, all output signals are controlled by an identical set of minterms, and the depths (number of signals along each path, excluding sources) of all transparent paths are the same. Also, if some input bits are primary inputs, while the other mapped bits (under the same set of minterms) are internal signals, they are classified into separate words.

The function *splitWords(...)* partitions a word if some output bits of it are primary outputs, while others are internal signals.

3.4.2 Overall Algorithm Flow

Based on the function in Algorithm 3.1, Algorithm 3.2 outlines the steps for finding all transparent words and identifying words on support or output boundaries for an input circuit.

Algorithm 3.2 Functional Approach

Require: Circuit
Ensure: TransparentBlocks = (Outputs, Supports, Words)
1: **ProvedWords**= $\emptyset$
2: **CandidateControls** = $findHighFanoutSignals$(**Circuit**)
3: **for all** *control* in **CandidateControls** **do**
4: **NewWords** = $findTransparency$(**Circuit**, {*control*})
5: **ProvedWords** $\bigcup$ = **NewWords**
6: **for all** *word* in **ProvedWords** **do**
7: **if** $notFullyTransparent$(*word*) **then**
8: **ControlSets** = $findPossibleCombinations$(*word*)
9: **for all** *controlSet* in **ControlSets do**
10: **if** $newCombination$(*controlSet*) **then**
11: **NewWords** = $findTransparency$(**Circuit**, *controlSet*)
12: $extend$(**ProvedWords**, **NewWords**)
13: $cleanMultiplyDrivenWords$(**ProvedWords**)
14: $partitionWords$(**ProvedWords**)
15: $disposeWords$(**ProvedWords**)
16: **TransparentBlocks** = $analyzeBlock$(**ProvedWords**)
17: **return** **TransparentBlocks**

To find control candidates, the function *findHighFanoutSignals(...)* in Line 2 uses the fact that all bits of a transparent word must be controlled by the same condition. It collects all signals with more than 3 immediate fanouts.

Lines 3 to 5 find all transparent words controlled by a single signal, including the standard 2-to-1 MUXes and depth-one words.

To recognize proceeding words, Lines 8 to 12 work on words which so far are not fully feeding into transparent paths. For each candidate word, the function *findPossibleCombination(...)* collects its depth-one fanouts and finds other transparent words which also support those fanouts. If the signal dependencies of supporting words and fanouts satisfy **Definition 3.6**, they may result in proceeding words. Note that if the input circuit is an and-inverter-graph (AIG), when only depth-one fanouts are considered, each proceeding word can only come from two proved words.

Then the union of the control sets of the two words is a candidate control set. The algorithm executes *findTransparency(...)* if this control set has never been considered before. The newly found words are appended at the end of **ProvedWords** for being examined by the same flow later.

Lines 13 and 14 rearrange all proved transparent words to achieve a legal word dependency graph, in which multiply-driven signals are absent, and all bits of one word are driven by the same set of words. The function *cleanMultiplyDrivenWords(...)* examines every signal driven by more than one transparent path and assigns it to the most *preferred* word. This process favors wider words first, and then deeper ones. The function *partitionWords(...)* partitions each word into smaller words if some bits are supported by different input words. Also, outputs driving different sets of words are grouped into different words.

The function *disposeWords(...)* discards transparent words with bad properties; it excludes all words with fewer than 4 bits after the above processes have been executed; it can also discard some depth-one words if they are suspected as *bad* transparent words - more details are discussed in Section 3.5.2.

Finally, the function *analyzeBlocks(...)* finds (1) input words on support boundaries which are not directly supported by transparent signals, and (2) output boundaries where words are not fully feeding into transparent signals. This information can be used later in other applications like reverse engineering.

3.4.3 Running Examples

The following three examples demonstrate how the proposed algorithm works on various unconventional cases.

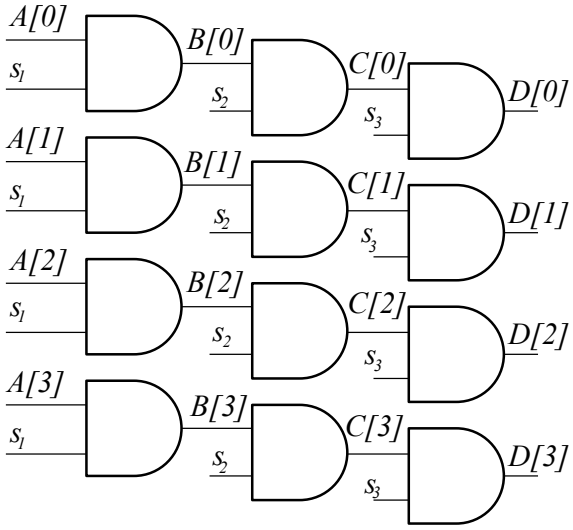


Figure 3.3: A compound word composed of three depth-one words.

Example 1: Figure 3.3 shows a compound word composed of three depth-one transparent words. First, high-fanout signals, s_1, s_2 and s_3, are collected as control candidates. Then depth-one transparent words, $s_1 \rightarrow (B \equiv A)$, $s_2 \rightarrow (C \equiv B)$ and $s_3 \rightarrow (D \equiv C)$ are proved by Lines 3 to 5 in Figure 3.2. Lines 7 to 14 are skipped because all words found above are

either fully connected to another word or support nothing. The function *disposeWords(...)* might drop some depth-one words in this circuit when certain strategies are applied. See Section 3.5.2 for a further discussion. If no word gets dropped, the whole circuit is reported as one transparent logic block, where A and D are input and output boundaries, respectively.

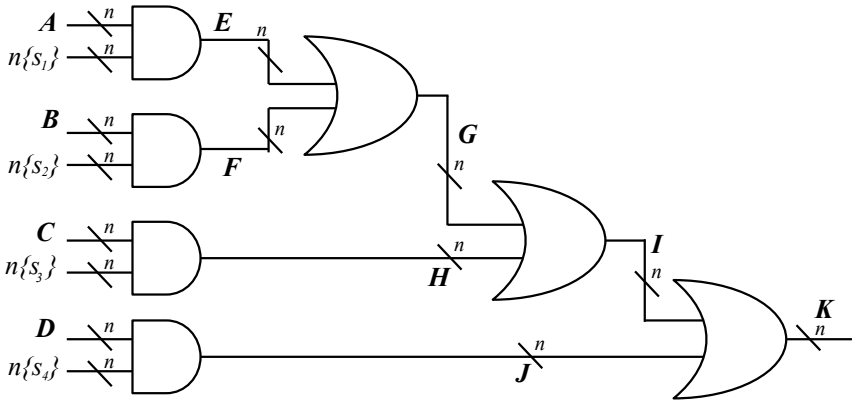


Figure 3.4: An example containing proceeding words.

Example 2: Figure 3.4 demonstrates how proceeding words can be found by the proposed algorithm. As before, s_1, s_2, s_3 and s_4 are recognized as control candidates. Then $s_1 \rightarrow (E \equiv A)$, $s_2 \rightarrow (F \equiv B)$, $s_3 \rightarrow (H \equiv C)$ and $s_4 \rightarrow (J \equiv D)$ are proved as depth-one transparent words.

Starting from E, *findPossibleCombinations(...)* finds that the combination of E, F and G satisfies the definition of a proceeding word, so $\{s_1, s_2\}$ is a new combination of controls. According to this control set, the function *findTransparency(...)* returns $s_1 s_2' \rightarrow (G \equiv A)$ and $s_1' s_2 \rightarrow (G \equiv B)$. Following the similar procedure, I is proved as transparent from A $(s_1 s_2' s_3')$, B $(s_1' s_2 s_3')$ and C $(s_1' s_2' s_3)$. Finally I and J are associated together and K is transparent from A $(s_1 s_2' s_3' s_4')$, B $(s_1' s_2 s_3' s_4')$, C $(s_1' s_2' s_3 s_4')$ and D $(s_1' s_2' s_3' s_4)$. The only word on the output boundary, K, is a depth-four transparent word supported by A, B, C and D.

Example 3: Consider Figure 3.5 as the input circuit. First, s_1 is recognized as a high-fanout signal. Then *applyMinterm(...)* in *findTransparency(...)* (Figure 3.1) finds $s_1 \rightarrow (\{E, G\} \equiv \{A, F\})$ and $s_1' \rightarrow (\{E, G\} \equiv \{B, D\})$. Then *analyzeWords(...)* in Figure 3.5 merges the two sets of paths into one word, $\{E, G\}$, and then partitions the word into $(E \equiv s_1 A + s_1' B)$ and $(G \equiv s_1 F + s_1' D)$, because sources of E (bits of A) are primary inputs, while those of G (bits of F) are internal signals. There are no more transparent words in this circuit.

There are two disjoint transparent blocks, because all paths through the adder are not transparent. For the block with output E, A and B are reported as support words, while F and D are support words for the other block with G as the output.

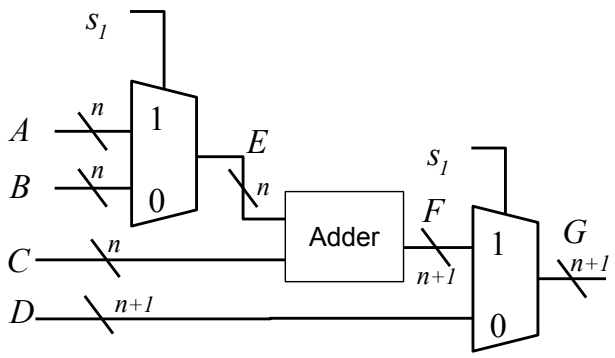


Figure 3.5: An example with disjoint transparent blocks.

3.5 Practical Challenges

Finding transparent words and performing perfect reverse engineering for real circuits can be challenging, but the proposed algorithm provides more possibilities to address issues that cannot be resolved by structural approaches.

3.5.1 Generalized Transparent Words

Often a word is transparent from words with different bitwidths - for example,

$$C[m-1:0] = s?A[m-2:0] : B[m-1:0]. \tag{3.9}$$

The most significant bit of C, $C[m-1] = s'B[m-1]$, is not NPN isomorphic to other bits of C. Hence this word is partitioned into one word with $m-1$ bits and one with 1 bit by the proposed algorithm. Therefore, some reverse engineering algorithms based on precise word boundaries can fail due to the decomposition of the whole word.

Also, it is possible that control signals can be part of input data words, i.e.

$$C[m-1:0] = s?A[m-1:0] : B[m-1:0], \tag{3.10}$$

where $s \equiv A[m-1]$. That is, $C[m-1] = sA[m-1] + s'B[m-1] = A[m-1] + B[m-1]$. Thus, $C[m-1]$ is different from other bits of C. These excluded bits are discarded by *disposeWords(...)* because of bit-width differences, and then the transparent boundaries are imperfect.

The above cases can be handled by modifying *analyzeWords(...)* in Algorithm 3.1. For both the above cases, when s is 1, $C[m-1]$ is constant, which is excluded from **Candidates**. When s is 0, $C[m-1]$ is transparent from $B[m-1]$, through a path with depth one, while other bits of C are through paths of depth two. Hence, *analyzeWords(...)* can merge all bits of C into one word even though some bits are controlled by only one minterm, and their depths are different. In other words, we can relax the requirement of NPN isomorphism to achieve *generalized transparent words*.

However, it is possible that the bits with different depths are indeed different words, so a practical reverse engineering process should consider both cases and use other information to revise the word boundary.

3.5.2 Ambiguity of Transparency

Some data signals are recognized as control candidates because they drive more than three fanouts. For example, a word-level multiplier can be synthesized as a set of adders among internal words,

$$\begin{aligned} C[2m-1:0] &= A[m-1:0] \times B[m-1:0] \\ &= A[m-1:0]B[0] + A[m-1:0]B[1] << 1 \\ &+ \cdots + A[m-1:0]B[m-1] << (m-1). \end{aligned} \tag{3.11}$$

These words are depth-one transparent words from one input word of the multiplier, controlled by bits of the other input word. These depth-one transparent words can be recognized and excluded by *disposeWords(...)*, considering they are directly feeding into non-transparent paths.

However, it is excessive to discard all depth one transparent words which do not support other transparent words. Consider the example in Figure 3.3. Bits of D are sinks of transparent paths, while they do not support other transparent words and can be discarded.

Also, if a word (a set of MUXes) is switching between one constant word and one variable, these MUXes can be synthesized as AND or OR gates, which comprise depth-one transparent words. Discarding these would lead to overlooking some transparencies.

Moreover, compared to the flow in Algorithm 3.2, the resulting words and boundaries are different when the function *disposeWords(...)* (discarding some depth-one words) executes before *partitionWords(...)*.

To resolve this issue for reverse engineering, it is preferable to run the proposed algorithm with different settings of *disposeWords(...)*, and combine the information of recognized operators to decide suitable boundaries.

3.5.3 Limitations of Proposed Algorithms

The proposed algorithm only considers composition and propagation of transparency, so it cannot recognize transparent blocks outside these categories.

Consider the circuit in Figure 3.6. It is a transparent block because $s_1 s_2 s_3 \rightarrow (B \equiv A)$, but it cannot be found by the proposed algorithm, which will find several depth-one transparent words controlled by s_1, s_2 and s_3, but then signal dependencies cannot satisfy the definition of proceeding words.

Note that this logic block still satisfies the definition of a transparent word, so it can be recognized by finding NPN isomorphism functions and permuting supports. However,

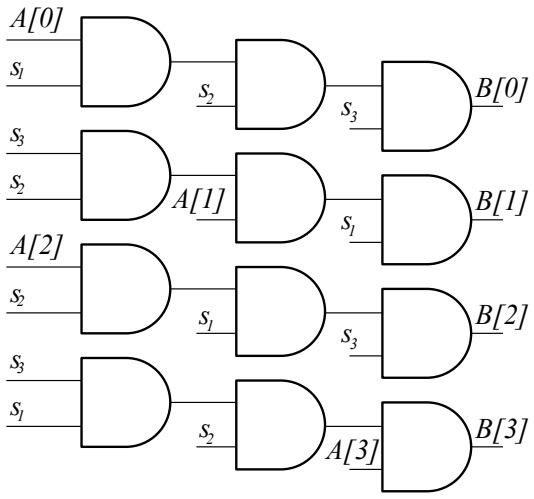


Figure 3.6: A transparent block which is not the result of compositions of NPN isomorphism classes.

searching for all NPN isomorphism classes can be time-consuming, especially if finding and revising ideal support and output boundaries is done.

In real (industrial) benchmarks, these types of cases do not seem to happen a lot, so they can be overlooked by the proposed algorithm without much loss.

3.6 Experimental Results

The proposed algorithms were implemented in ABC [8]. All experiments were performed on a 16-core 2.60GHz Intel(R) Xeon(R) CPU with no time limit. All cases were processed as AIGs as inputs. Sequential circuits were converted into combinational designs by replacing flip-flops inputs and outputs with primary outputs and inputs respectively.

As a reference for the functional approach, we implemented a purely structural approach: 1) structural matching is used to locate all 2-to-1 MUXes in the AIGs, 2) signals with the same control are grouped into one word, and these connected words are collected into larger transparent blocks and 3) words are partitioned into sub-words if they are supported by different input words or drive different output words.

We wanted to compare the efficiency and effectiveness of the structural algorithms versus our functional algorithms applied to highly-transparent cases. To select these, the proposed functional algorithm was applied to all 230 cases of the single-output track in the Hardware Model Checking Competition 2014 [17] after their conversion to combinational circuits. For each case, some POs were proved conditionally equivalent to some primary inputs. We computed the proportion of those POs to all POs, and ran experiments on the top 10 cases with the highest percentages of transparent POs. Among the 230 cases, there are 20 cases with more than 50% transparent POs, while another 38 cases have more than

25% transparent POs. Table 3.1 shows the statistics of the selected cases after they were converted to combinational circuits. The last column of Table 3.1 indicates the percentages of transparent POs to all POs. The *6sxxx* cases are industrial problems from IBM and the *beem* examples come from different applications areas such as protocols, planning, scheduling, communication, or puzzles.

Table 3.1: Statistics of the selected benchmarks from HWMCC'14 [17].

Case Name	PI #	PO #	AND #	Trans. PO %
6s195.aig	1344	1258	8046	87.1
beemfrogs1b1.aig	323	159	8493	86.0
6s171.aig	1357	1263	8074	84.6
beemloyd3b1.aig	237	118	3970	82.1
6s282b01.aig	1977	1934	10264	81.2
6s384rb024.aig	22367	14953	47933	79.0
6s206rb103.aig	37847	28644	103375	71.4
6s302rb09.aig	36962	27777	100571	70.3
6s348b53.aig	15797	15561	89567	70.1
beemldelec4b1.aig	2559	1215	34252	67.5

3.6.1 Comparison between Functional and Structural Approaches

Table 3.2 shows the comparisons between the structural approach and the proposed functional method. Column 2 indicates the total number of signals (AIG nodes or primary inputs) that were classified as belonging to words. Columns 3-8 (labeled *Structural Approach*) give the statistics of the transparencies found using the reference structural approach. Column 3 lists the total number of structural MUXes recognized. Column 4 lists the number of AIG nodes plus inputs covered by all the transparent logic blocks found; Column 5 gives the (minimum, maximum) widths (the number of MUXes grouped together as a word) of found and partitioned words, and Column 6 shows the (minimum, maximum) depths of transparent words on boundaries. The depth of each word is the total number of AIG nodes between itself and the primary inputs, where one MUX is counted as depth 2. Column 7 (labeled *Forward*) lists the total number of signals which are in the transparent block where all input words are primary inputs. Column 8 shows the run-time of the overall structural approach. Here we only identify 2-to-1 MUXes and MUXes with negation on outputs or inputs. We omit counting words (after partitioning) with less than 4 bits. Columns 9-13 (labeled Proposed Functional Approach) show similar statistics for the proposed algorithm. Here we include all depth-one transparent words when counting signals or depths of transparent blocks.

Observation of Structural Results Table 3.2 shows that most benchmarks contain wide transparent words. The run times show that this approach is very efficient as expected.

Although these cases have high percentages of transparent POs, for some cases (beemldelec4b1.aig) the structural approach cannot find any transparent words reachable from pri-

Table 3.2: Experimental results of the structural and functional approaches on ten selected cases from HWMCC'14 [17].

Case Name	Total Sig. #	Structural Approach						Proposed Functional Approach				
		Mux #	Sig.#	Widths	Depths	Forward	time(s)	Sig. #	Widths	Depths	Forward	time(s)
6s195.aig	9390	2357	8090	4, 512	2, 12	4620	0.133	8820	4, 512	1, 16	7221	0.183
beemfrogs1b1.aig	8816	2016	5381	6, 8	2, 32	520	0.142	6365	4, 18	1, 34	827	0.267
6s171.aig	9431	2362	8135	4, 512	2, 12	4737	0.161	8847	4, 512	1, 16	7249	0.246
beemloyd3b1.aig	4207	985	2771	6, 8	2, 28	360	0.125	3071	4, 13	1, 29	1512	0.167
6s282b01.aig	12241	2472	8490	4, 982	2, 54	6175	0.142	10369	4, 982	1, 66	8072	0.291
6s384rb024.aig	70300	14492	53380	4, 3723	2, 8	46083	0.183	57717	4, 3663	1, 11	51918	0.591
6s206rb103.aig	141222	28684	106822	4, 737	2, 8	87328	0.258	116996	4, 651	1, 10	101388	2.365
6s302rb09.aig	137533	27818	103864	4, 590	2, 8	84943	0.250	113803	4, 471	1, 10	98888	2.250
6s348b53.aig	105364	28775	66356	4, 427	2, 16	52107	0.217	78757	4, 427	1, 17	69525	1.274
beemldelec4b1.aig	36811	8458	14561	5, 41	2, 76	0	0.167	24169	4, 9	1, 104	9199	1.857

Table 3.3: Experimental results of the functional approaches on unrolled cases from HWMCC'14 [17].

Case Name	Two Time Frames					Three Time Frames				
	Sig. #	Widths	Depths	Forward	time(s)	Sig. #	Widths	Depths	Forward	time(s)
6s195.aig	16149	4, 512	1, 18	9693	0.300	23484	4, 512	1, 20	10112	0.533
beemfrogs1b1.aig	12750	4, 18	1, 41	1674	0.591	19135	4, 18	1, 41	2521	0.874
6s171.aig	16194	4, 512	1, 18	9740	0.358	23547	4, 512	1, 20	10178	0.533
beemloyd3b1.aig	6195	4, 13	2, 29	3055	0.217	9319	4, 13	1, 29	4598	0.308
6s282b01.aig	18782	4, 881	1, 68	9722	0.500	25332	4, 881	1, 72	11584	1.074
6s384rb024.aig	101059	4, 2993	1, 13	85193	1.915	144299	4, 2691	1, 17	115368	4.205
6s206rb103.aig	207172	4, 585	1, 12	164619	6.553	297312	4, 585	1, 19	219035	13.580
6s302rb09.aig	201639	4, 455	1, 12	160876	6.303	289451	4, 455	1, 19	214364	12.931
6s348b53.aig	141452	4, 256	1, 21	117946	3.789	203975	4, 256	1, 31	160923	7.547
beemldelec4b1.aig	48308	4, 9	1, 116	18368	4.438	72447	4, 9	1, 116	27537	7.012

mary inputs. Many MUXes are recognized but there are several reasons why the structural approach misses many transparent words:

1. Structural matching only considers standard 2-to-1 multiplexers, while there are other types of transparent functions.

2. Many of the identified MUXes are controlled by different selection signals, and thus lead to words of less than 4 bits, which are excluded in the analysis. Moreover, some words are partitioned into small words because their output or input dependencies are different.

3. Forward transparent words are required to be reachable from primary inputs through fully transparent paths. If a transparent word originates from the output word of an arithmetic operator (e.g. words G and F in Figure 3.5) or a depth-one transparent word, it would not be reported, yet many MUXes would be involved in such a transparency.

Although quite fast, this approach itself is not enough for finding many of the whole transparent blocks that exist in these benchmarks as shown in the columns which show the forward and total signals found by the functional approaches.

Comparing Functional and Structural Approaches Based on Table 3.2, we observe the following:

1. According to the signals covered by transparent blocks, the proposed functional approach can find more and larger transparent blocks. Note that for some cases the differences are not huge, which leads to a conclusion that most transparent logic in those cases are comprised by standard 2-to-1 MUXes.

2. The minimum and maximum widths of transparent words are different for the structural and functional approaches.

3. For most cases, the proposed functional approach can find deeper transparent paths, because the functional approach can find depth-one transparent words and compose them into larger transparent blocks.

4. In general, the proposed algorithm can find many more transparent words reachable from primary inputs. It might be that some transparent paths start with MUXes between a constant integer and an input word, which cannot be recognized by the structural approach. Therefore all transparent words supported by that will not be reported as forward transparent words by the structural approach.

5. Although the functional method takes more time than the structural approach, the run times for the selected cases do not exceed 3 seconds. Hence the proposed algorithm is efficient enough for most applications.

For real applications, the particular final usage of the found words might dictate a suitable balance between performance and the number of proved words.

Even though some transparent words found by the functional approach may be discarded when combined with other techniques for recognizing arithmetic operators, it is better to have more candidates words for more refined reverse engineering applications.

3.6.2 Experiments on Unrolled Circuits

Table 3.3 shows the experimental results of running the proposed algorithm on circuits unrolled for two and three time frames. The columns are similar to those in Table 3.2.

Observation of Unrolled Circuits

Table 3.2 shows that, for each circuit, the number of signals covered by transparent blocks grows as the number of time frames increases. The changes of other statistics vary among the different circuits:

1. The maximal width of words remains the same for most of the cases, but for some, the maximal width decreases after unrolling. The reason is, after unrolling, more words are partitioned into smaller words because some bits support different words in their fanout cones.

2. For many cases, the maximal depths of transparent blocks increase as the numbers of time frames increases. These deeper paths indicate that some transparent paths continue from the first time frame to the second, and some continue into the third time frame. If the maximal path is not connected to another transparent path in the next time frame, the maximal depth remains the same.

3. The total number of signals in transparent blocks supported by primary inputs (labeled Forward) increases for all circuits, but the growth rate is distinct for each circuit. For some cases, many transparent POs are connected to other transparent paths in the next time frame, so after unrolling, there are more internal transparent words reachable from primary inputs.

The distinct statistics of transparent blocks found in different circuits show that finding transparent logic may be important for understanding circuit properties, and by finding as many as possible transparent blocks the proposed approach may be very useful in providing useful information about a circuit.

3.7 Summary and Possible Applications

This chapter presented an algorithm to identify transparent logic, which can be used to extract word-level information from gate-level circuits. Some challenges for finding the most

accurate boundaries for the transparencies were discussed. Experimental results demonstrated that the proposed algorithms can be very effective in extracting words as well as some control logic.

The concept of transparent logic and the proposed functional approaches can be applied to, but not limited to the following directions:

1. Reverse engineering: the proposed method can be integrated with other reverse engineering techniques that can identify word-level operators. Through iterations between different approaches, word-level information and found boundaries can be revised, benefiting both, by finding more transparencies and identifying more operators.

2. Datapath synthesis: to minimize areas of datapath modules in modern microprocessors and embedded systems, some high-level optimization technique, such as resource sharing, have been applied to gate-level circuits [47]. Finding transparent logic blocks can extend the existing MUX-based method to identify more common specification logic and to minimize the area.

3. Abstraction for model checking: the transparent words found by the proposed method could be input or output of word-level operators. Identifying arithmetic operators, such as multipliers, enables black-boxing and using uninterpreted function (UIF) constraints [18] for more efficient model checking. Also, knowing control logic boundaries can guide co-factoring and abstraction of hard verification problems. In hardware security, identification of datawords and datapaths is critical for validating safety of data propagation [37].

In the following chapters, transparent logic is used to build abstraction models of sequential circuits to assist in clock-gating verification and synthesis.

Chapter 4

Dependency Graphs

The goal here is more aggressive than in Chapter 2. It is to derive "maximal" sufficient conditions for a data signal to be not influencing the computation (*satisfiability*), or a computation result to be never observable at any output (*observability*). It is to be maximal, given the constraint that combinational logic, other than transparent logic, should not be considered. This is because we want our verification algorithms to be able to abstract this logic away, making verification easier.

A major part of this effort is identifying all *transparent logic* in the design, as discussed in Chapter 3. Considering transparent blocks of logic can strengthen clock-disabling conditions for some FFs, and hence save more dynamic power. For *satisfiability* clock-gating cases, more detailed information about data dependency can assist in defining more general conditions when input data need not be updated. For *observability* clock-gating cases, paths to observable outputs can be blocked by transparent logic and result the clock being disabled more often.

Example: Figure 4.1 contains a set of gated FFs, F_2, which are processed by a word-level square root operation ($\sqrt{}$) and then fed into a transparent block (words A, B, C to I.) I depends on A (through F_2) only when $F_1 = s_1 = 1$. Hence, F_2 need not be updated when $s_1 = 0$ in the next time frame; when $s_0 = 0$, the clock to F_2 can be disabled. This example of legal observability clock-gating can be identified only when transparent logic is considered.

To formulate legal clock-gating conditions on circuits that may have transparent blocks, we propose to construct a *dependency graph*, DG, and then formulate a set of properties sufficient for legal clock-gating.

A dependency graph (DG) is an abstraction of a circuit. The DG addresses clock-gating conditions related to data dependencies at a high-level, providing essential information for synthesis and verification.

The definition of dependency graphs is introduced in Section 4.1. Section 4.2 states the proposed algorithm to construct DGs. Section 4.3 compares DGs with CGs and summarizes this chapter.

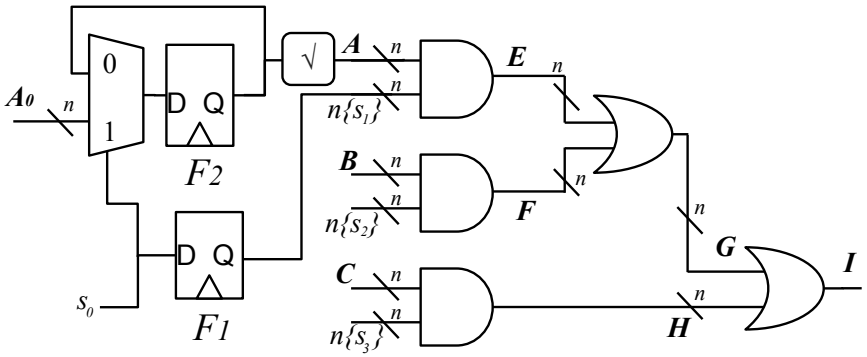


Figure 4.1: Considering transparent logic in clock-gating.

4.1 Dependency Graph

A dependency graph G = (E, V) is a directed graph, where each vertex associates a set of signals with a certain sub-circuit in the corresponding circuit. Each directed edge represents a data dependency.

We define eight types of vertices: (1) primary inputs, (2) constants, (3) primary outputs, (4) standard flip-flops, (5) transparent blocks, (6) combinational clouds, (7) gated flip-flops, and (8) signal branches. The eight types with related signals are shown in Figure 4.2.

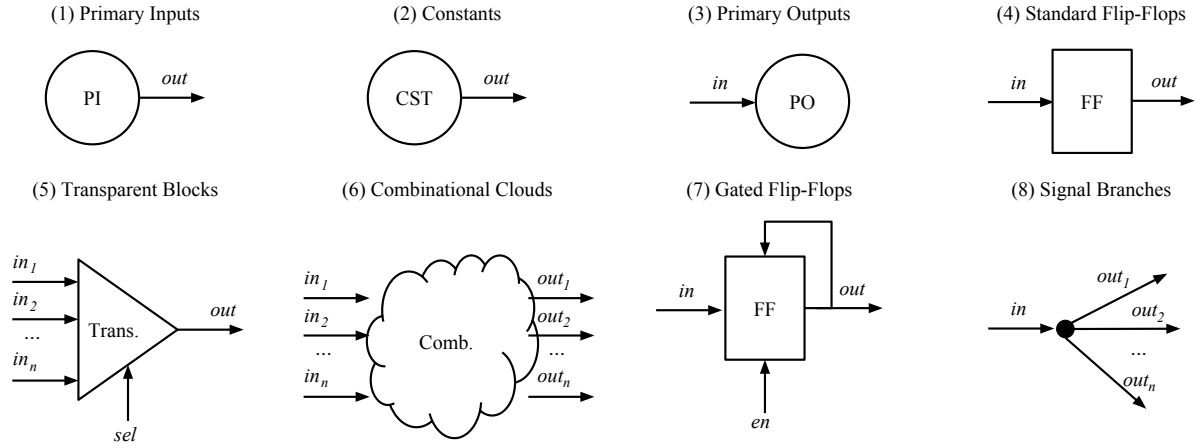


Figure 4.2: Eight types of vertices used in a dependency graph

- Vertices for primary inputs and constants have no inputs, while primary outputs only receive data from other vertices.
- A standard flip-flop vertex receives input data and propagates it to its output in the next time frame.

- A transparent block receives multiple inputs and forwards one of them to its output according to the value of *sel*. *sel* can represent multiple signals. For example, a set of n-to-1 MUXes controlled by MUX inputs $in_1, ..., in_n$ can be mapped into one DG vertex labeled with a single selector signal with $\lceil log_2(n) \rceil$ values.

  For incompletely specified transparent blocks, as in Figure 4.1, the assignments for transparency as well as all possible assignments for selector signals should be represented in the DG. When constructing a DG for analyzing clock-gating conditions, it is a must to represent the complete functionality of an incompletely specified transparent block, because (1) there is no guarantee that only the control assignments for transparency can happen, and (2) the other assignments can still result output updating, which should be considered for satisfiability clock-gating conditions.

  Example: In Figure 4.1, the complete specification for the data dependency of I includes that it depends on the value of A when $s_1 = 1$, regardless of the assignments for s_2 and s_3. The same idea applies to B and C with $s_2 = 1$ and $s_3 = 1$, respectively. But when more than one of (s_1, s_2, s_3) is 1, I depends on multiple words. Also, when all (s_1, s_2, s_3) are 0, $I = 0$. All these data dependencies should be represented with a transparent block vertex having an extra constant vertex as input, in the case shown in Figure 4.3.

- A combinational cloud represents a sub-circuit with no transparent logic. It can cover arithmetic operators and other complicated computational units. This type of vertex is used to abstract away irrelevant logic when formulating clock-gating properties. It is possible that some of this logic is relevant to a more detailed understanding of dependencies, but we chose to ignore these complications. This approach is conservative but still produces strong enabling signals, while making verification and synthesis easier.

- A gated FF vertex represents a set of already gated FFs, gated by the same signal *en*. These FFs are updated to the data input only when $en = 1$. In practice, a set of FFs can be clock-gated by a sequence of MUXes. These FFs and all MUXes are collapsed and modeled as one gated FF vertex with one enable signal. More details for MUX collapsing are discussed in Section 4.2.

- A signal branch is used for signals with multiple fanouts.

Example: Figure 4.3 demonstrates the dependency graph constructed for the circuit in Figure 4.1. The square root operator is represented by a combinational cloud. The transparent block, which is from words A, B, C to I, is mapped to a single transparent vertex, controlled by (s_1, s_2, s_3). A constant node is introduced to represent the condition when all (s_1, s_2, s_3) are zero, and thus $I = 0$. The set F_2 of gated FFs is covered by one gated FF vertex, which includes the MUXes controlled by s_0. Note that a FF F_1 is not represented, because it is irrelevant to the overall data flow. The edges in the DG represent data dependencies but not the control signals for gated FFs or transparent blocks.

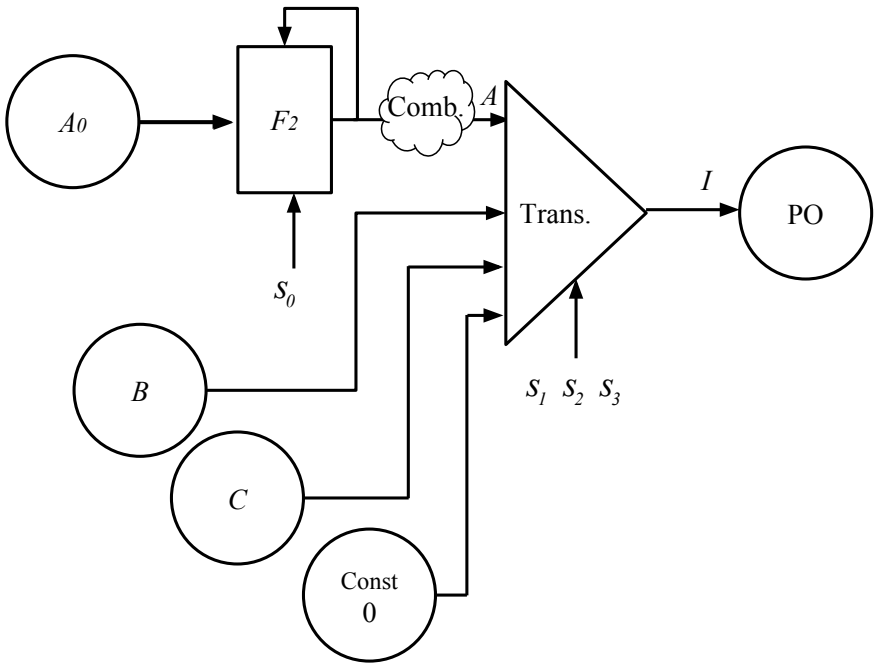


Figure 4.3: Dependency graph for the circuit in Figure 4.1.

4.2 Construction of Dependency Graph

A DG is constructed from a sequential circuit and its identified transparent blocks as follows: (1) recognize gated FF vertices, (2) complete incomplete transparent block vertices, (3) create other vertices, including combinational clouds, and (4) build dependencies.

According to the algorithm for identifying transparent blocks in Chapter 3, initially a set of bits is grouped together because of identical control signals and assignments. Then for each group, if some bits are supported by different sources, or some of them drive different transparent words in their fanout cones, the group is decomposed into sub-groups as transparent words. Thus, the identified transparent blocks are examined and decomposed to ensure that 1) all bits of one word are supported by identical input words and 2) all bits support the same set of words their fanout cones. Hence for each DG, a vertex depends on all bits of each vertex supporting it, so there are no redundancies where a bit is connected to a vertex but cannot influence the vertex value.

Recognize gated FF vertices: MUXes that support FFs are investigated first. When a fully-specified transparent block forms a clock-gating structure, where the FF output drives one data input of a corresponding 2-to-1 MUX (a MUX feedback loop), the other input word is examined. If it comes from another transparent word, and each bit of this is supported by the FF in its fanout cone under the same control condition, then this transparent block is regarded as part of a clock-gated FF. The procedure continues until no such MUX connects to the target FFs can be found. Then all selector signals of the collected MUXes are combined

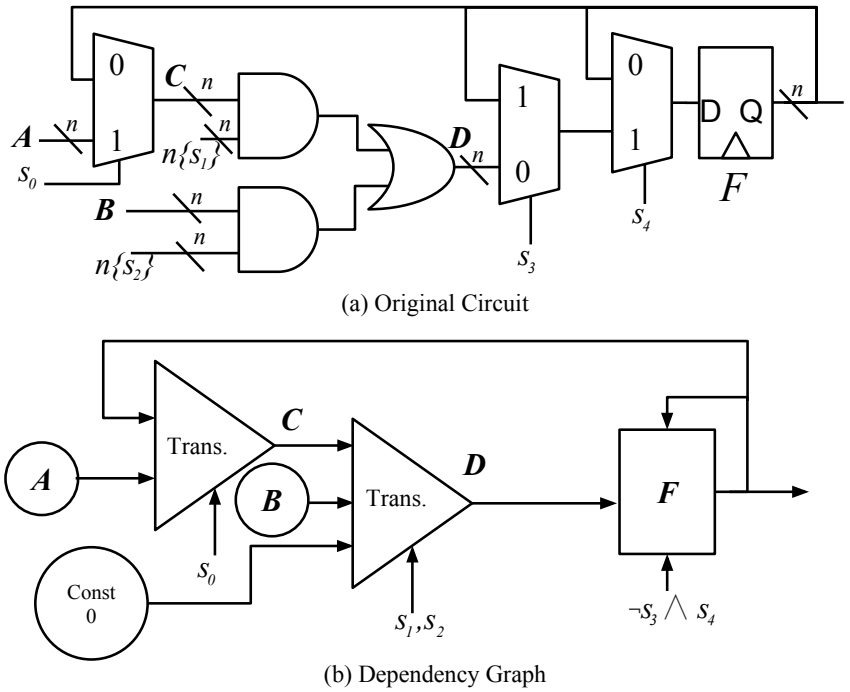


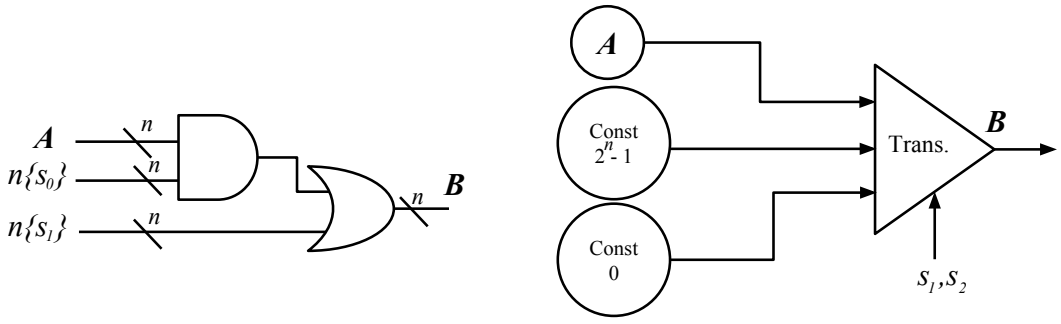
Figure 4.4: (a) Circuit with transparent blocks and gated FFs. (b) Corresponding DG.

into a single enable signal for this clock-gating vertex.

Example: In Figure 4.4(a), the MUXes (controlled by s_4) directly feeding FFs (F) are examined first. The process examines the other input side (not driven by F) and collects the MUXes controlled by s_3. Continuing, when it reaches transparent word D, it terminates because D is not directly in the support of F. Finally s_3 and s_4 are combined to form the enable signal of a gated FF vertex, with data input D. Note that although the MUXes controlled by s_0 are in the support of F directly, they are excluded from the gated FF vertex because there exists another transparent block (outside the clock-gating part) between its output C and F.

Complete transparent block vertices: After creating all gated FF vertices, the remaining transparent blocks are analyzed to create corresponding vertices. For an incompletely specified transparent block, some assignments of controls do not result in transparency, but dependencies between inputs and outputs still exist. Sometimes the transparent logic may be used to reset some FFs to constants and then make the output value different from the previous time frame. Hence, it is necessary to analyze all possible control assignments and create extra constant vertices to represent the entire functionality of the block.

Example: In Figure 4.5 B is transparent from A when $s_0 = 1, s_1 = 0$. It evaluates to zero when $s_0 = 0, s_1 = 0$. Also $B = 2^n - 1$ if $s_1 = 1$. Thus two constant vertices are added to complete the description of this block.



(a) Original Circuit (b) Dependency Graph

Figure 4.5: (a) Transparent block with two possible constant outputs. (b) Corresponding dependency graph.

Create other vertices: After creating all gated FF and transparent block vertices, all transparent words are collected and covered by vertices. If some input words are sets of FFs or PIs, then primary input or standard flip-flip vertices are created to cover these. Other words, consisting of internal signals (outputs of some combinational clouds), like A in Figure 4.1, are labeled temporarily as *words*. Then vertices for the remaining FFs, PIs and POs are created one by one, where each vertex only covers one signal. Note that if a FF or PI only controls transparent blocks, like F_1 in Figure 4.1, a vertex not needed for it because it is not part of the data flow.

The next task is to create combinational cloud vertices. First, each word or vertex accumulates a list of support vertices (words) by following the target fanin cone until reaching another vertex (word). One combinational cloud is created for each set of vertices that have identical support lists.

Example: In Figure 4.6, words A to F have been found and labeled. The lists of supports of words D, E, F are determined as $\{A, B\}$, $\{A, B\}$ and $\{B, C\}$, respectively. Hence one combinational cloud is created for D and E, and another is created for F. Before building edges from A, B, C to the two clouds, a signal branch for B is needed, no matter if it terminates as a vertex or a word.

The algorithm for constructing the dependency graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ for a sequential circuit, **Cir**, with its set of identified transparent blocks, is shown in Algorithm 4.1.

In Lines 2 to 5, gated FF vertices are constructed, where *collectGating*(...) performs backtracking to find clock-gating conditions for each set of gated FFs. Then the remaining transparent blocks are processed by Lines 6 to 8. During the two phases above, the vertices for some input data words are created if one word is a set of PIs or FFs.

Other vertices, except signal branches and combinational clouds, are created by *createVertex*(...). Then *createClouds*(...) explores the fanin cone of each word and collects the vertices or words supporting it. Finally this function creates one combinational cloud vertex for each set of words with the same support list.

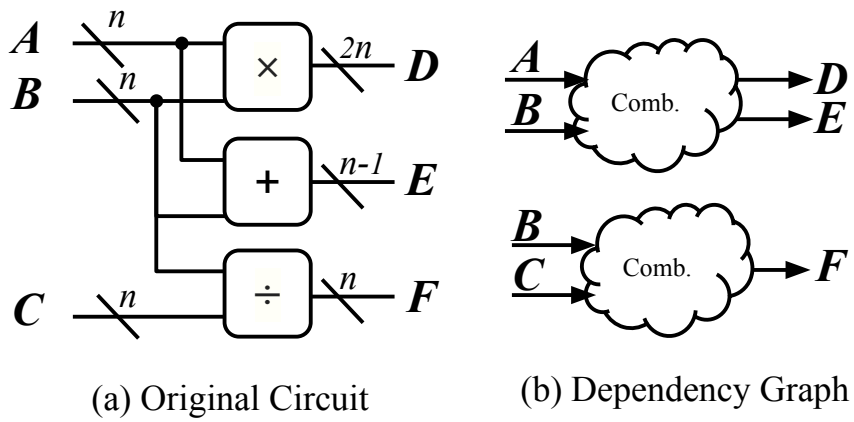


Figure 4.6: (a) Three arithmetic operators with shared input words. (b) Corresponding DG.

After topologically sorting all existing vertices, Lines 14 to 18 check each vertex and creates a signal branch if the vertex drives multiple vertices. For combination clouds, signal branches are created for all *output words* with multiple fanouts. Then each vertex is connected to all vertices supporting it. This cannot be done without the topological sort, because a branch vertex should be created before a signal is connected to its fanouts.

Once the DG is constructed, it can be used to formulate properties of legal clock-gating conditions, which will be discussed in the next chapter.

4.3 Summary

In this chapter, we introduced a second abstraction model, a dependency graph, to model a sequential circuit. Although both CGs and DGs aim at modelling control logic and data dependencies, they have significant differences as stated below:

1. Instead of different types of edges, DGs utilize different types of vertices to represent distinct functionalities of sub-circuits; DG edges are only used to describe data dependencies. The modelling is more compatible with an object-oriented programming paradigm. Also, if other types of logic blocks are identified and recognized in the future, the current architecture can be extended easily by adding new vertex types .

2. DGs can identify a series of MUXes for clock-gating and merge them into one gated-FF vertex, while CGs only accept the case where at most one MUX is in front of each FF. The feature of DGs is essential to handle FFs that have been gated multiple times.

3. DGs address more detailed data dependencies than CGs do. Recognizing transparent logic blocks can decompose big combinational blocks into smaller sub-circuits and can provide more detailed data flow, leading to stronger conditions for clock-gating.

Algorithm 4.1 Dependency Graph Construction

Require: **Cir**: a gate-level sequential circuit with the sets of primary inputs **PI**, primary outputs **PO** and flip-flops **FF**; **TransparentBlocks** = (**Outputs**, **Supports**, **Words**)
Ensure: $\mathbf{G} = (\mathbf{V}, \mathbf{E})$: dependency graph for **Cir**, with the set of vertices, **V** and edges, **E**.

1: $\mathbf{V} = \emptyset$ and $\mathbf{E} = \emptyset$
2: **for all** *block* in **TransparentBlocks do**
3: **if** *block* forms clock-gating structure **then**
4: $en = collectGating(block)$
5: $\mathbf{V} = \mathbf{V} \cup createGatedFF(block, en)$
6: **for all** *block* in $\{\mathbf{TransparentBlocks} - \mathbf{V}\}$ **do**
7: $(sel, \mathbf{ConstVs}) = analyzeTransparency(block)$
8: $\mathbf{V} = \mathbf{V} \cup \{\mathbf{ConstVs}, createTrans(block, sel, \mathbf{ConstVs})\}$
9: **for all** *signal* in **PO** or **FF** or **PI** **do**
10: **if** *signal* is not covered by any v in **V then**
11: $\mathbf{V} = \mathbf{V} \cup createVertex(signal)$
12: $\mathbf{V} = \mathbf{V} \cup createClouds(\mathbf{Cir}, \mathbf{V})$
13: $topologicalSort(\mathbf{V})$
14: **for all** v in **V do**
15: **if** v drives multiple fanouts **then**
16: $\mathbf{V} = \mathbf{V} \cup createBranch(v)$
17: **for all** *input* vertex supporting v **do**
18: $\mathbf{E} = \mathbf{E} \cup buildDependency(v, input)$

Due to the extra information DGs provide, it takes more time to analyze circuits and build DGs. However, DGs are able to provide more precise properties for legal clock-gating conditions and benefit clock-gating synthesis and verification.

Chapter 5

Legal Clock-Gating Conditions

Clock-gating synthesis strives to add extra control logic to reduce the frequency of updates FFs. To verify that the added clock-gating condition is legal, i.e. the revised circuit is sequentially equivalent to the original, properties of the circuit are formulated that are sufficient for the extra control to be legal.

In this chapter, we derive and prove sufficient conditions for legal satisfiability and observability clock-gating on sequential circuits and formulate properties for FFs that are targeted for clock-gating. DGs are used to formulate problems and derive properties, and then these properties are proved on the original sequential circuit. It is important to note that the properties are relatively easy to prove because they are formulated using the DG and are independent of complicated combinational logic in the circuit.

This chapter is organized as follows. Identification of gated FFs is described in Section 5.1. Section 5.2 reviews basic LTL and PLTL operators used in this chapter to describe legal clock-gating properties. Sections 5.3 and 5.4 explain legal conditions for observability clock-gating and satisfiability clock-gating, respectively. Section 5.5 uses more examples to demonstrate how clock-gating conditions can be associated with DGs. The circuit-based approaches to justify formulated properties are stated in Section 5.6. Finally, Section 5.7 summarizes this chapter.

5.1 Problem Formulation using DGs

For a sequential circuit, clock-gating synthesis on a set of FFs can be represented by adding MUXes with feedback loops. The same idea has several representations in a DG. Given a set of target FFs, which are covered by a single DG vertex (a standard FF or a gated FF vertex), two possible differences between a golden DG and its revised DG can indicate that clock-gating synthesis has been performed:

- Change from a standard to a gated FF vertex. This is done when a standard FF, updated to the input data at each time frame, now has a proposed control condition en; when $en = 0$, those FFs are kept at the same values as saved.

- Change the *en* signal of an existing gated FF vertex. The FFs were already gated by en_{old}, but clock-gating synthesis proposed en_{extra}, which is combined with en_{old} to build $en_{new} = en_{old} \wedge en_{extra}$; thus, $(en_{new} = 1) \Rightarrow (en_{old} = 1)$. Note that it is assumed that en_{old} is legal because it is given as part of the golden model and in general we may not have enough information to test this for legality.

Both of the above cases can be viewed as changing an enable signal from en_{old} to en_{new}, where in the first case, en_{old} = constant-1. To verify if the synthesis is legal, the following algorithms only need to check the legality of $en_{old} \wedge (\neg en_{new})$, because that is where the enabling signal has changed from 1 to 0.

In the following sections, a set of gated FFs is called *out-of-date* when the input can be different from the previous time frame (updating), but their values keep the same due to $en = 0$. If no such situation happens, the FFs are called *up-to-date*.

5.2 LTL and Past LTL

The following LTL operators are used in expressing observability conditions:

1. $\mathbf{X}\ a$ – "next": a holds in the next cycle.
2. $[a\ \mathbf{U}\ b]$ – "until": a remains $True$ at least until b becomes $True$, which can happen at the current or a future time frame.

Temporal formulas for satisfiability clock-gating conditions are expressed in LTL extended with *past operators* (PLTL). Only those used for constructing clock-gating properties in this thesis are explained. Detailed formal semantics of PLTL can be found in [4].

The following PLTL operators are used:

1. $\mathbf{Y}\ a$ – "yesterday": a was $True$ in the previous time frame and $False$ in the first time frame.
2. $\mathbf{Z}\ a$ – "invariant yesterday": a was $True$ in the previous time frame and $True$ in the current time frame.
3. $[a\ \mathbf{S}\ b]$ – "since": (1) b was $True$ at least once in a past (or the current) time frame, and (2) a was $True$ since the cycle after b was last $True$.

Note that **Y** and **Z** are past-duals of **X** (next), while **S** is the past-dual of **U** (until) in common LTL.

5.3 Observability Clock-Gating Conditions

Observability clock-gating disables updating FFs when their inputs are *never observable*. It depends on control conditions and data dependencies of their fanout signals from now and in the future. Hence the properties can be expressed by common LTL operators.

5.3.1 Observable Condition

A set of signals at a time frame is called *observable* when their current values can influence primary output values at the current or later time frames. Given a target FF vertex, in which the enable signal is modified from en_{old} to en_{new}, for observability clock-gating, we need to check if the inputs of the target FFs are not observable when $en_{old} \wedge (\neg en_{new})$ happens. Because the target FF can become out-of-date and observable in future time frames, we cannot just check the observable condition of the FF outputs in the next time frame.

For each type of DG vertex, the observable condition of each input depends on those of the vertex outputs and the control signals. We define $\mathbb{O}(s)$ as the observable condition for a signal s at a certain time frame. See the second column of Table 5.1 for the observable condition of each vertex type.

Vertex Type	Observable Condition for in: $\mathbb{O}(in)$
Primary Input	N/A
Constant	N/A
Primary Output	$True$
Standard FF	$\mathbf{X}(\mathbb{O}(out))$
Transparent Block	for in_i: $\mathbb{O}(out) \wedge (sel = assign_i)$
Combinational Cloud	$\bigvee_i \mathbb{O}(out_i)$
Signal Branch	$\bigvee_i \mathbb{O}(out_i)$
Gated FF	$[en_{old}] \wedge \{\mathbf{X}(\mathbb{O}(out)) \vee \mathbf{X}[(\neg en_{new})\mathbf{U}(\mathbb{O}(out))]\}$

Table 5.1: Observable condition for the input of each type of vertex.

- Each PO is observable all the time, so the observable condition is $True$.
- The observable condition is not applicable for PIs or constants.
- The input of a standard FF is observable only when its output is observable in the next time frame.
- For each possible input of a transparent block, it is observable only when the output currently depends on this input and the output is observable.
- Any input of a combinational cloud or branch vertex is observable when any output is observable. Hence we take the union of the observable conditions of all output signals. As mentioned, this is, by definition, independent of the actual logic in the cloud on purpose.

- The observable condition for the input of a gated FF vertex depends on the control condition (en) and the observable condition of the output from now on. To make sure that the input data is taken and saved in the FF, en must be 1 now, and in the next time frame either (1) the output is observable ($\mathbf{X}(\mathbb{O}(out))$), or (2) the output becomes observable before the FF updates to another value, i.e. $\mathbf{X}[(\neg en)\mathbf{U}(\mathbb{O}(out))]$. Note that $[(\neg en)\mathbf{U}(\mathbb{O}(out))]$ can be $False$ only when $(en = 1)$ happens before $\mathbb{O}(out) = True$. The $\mathbf{X}$ outside the $\mathbf{U}$ formula avoids considering the first time frame where $en = 1$.

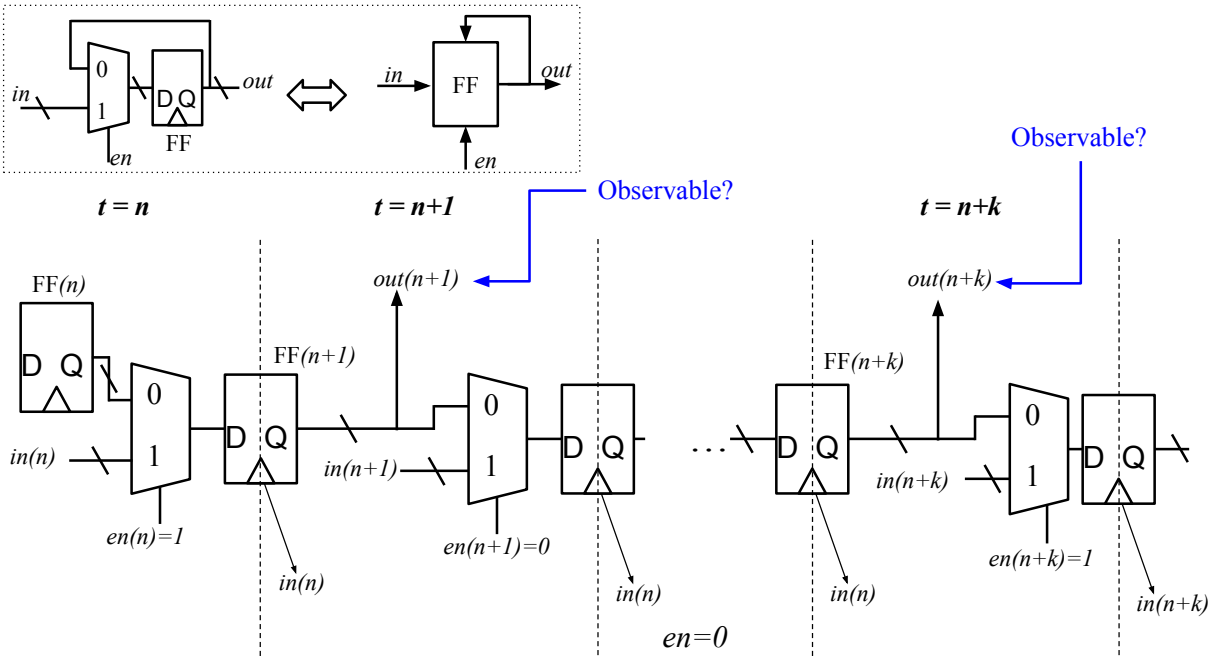


Figure 5.1: An example to demonstrate the observable condition of the input for a set of gated FFs at the n^{th} time frame, labelled as $in(n)$. The bottom diagram represents the top diagram unrolled k time frames when en again becomes 1.

Example: (Refer to Figure 5.1) Consider a gated FF where $en = 1$ at the n^{th} time frame, such that the FF receives and saves the input value, $in(n)$, at the end of this cycle. If the FF output is observable at the $(n+1)^{th}$ time frame, the input data from the previous time frame $in(n)$ is observable. If en is 0 at at the $(n+1)^{th}$ time frame and remains 0 before the $(n+k)^{th}$ time frame, the FF keeps the same value, i.e. $in(n)$, before the end of the $(n+k)^{th}$ time frame. If $\mathbb{O}(out)$ is $True$ at a certain $(n+l)^{th}$ time frame $(n < n+l \leq n+k)$, the saved value $(in(n))$ is observable. One extreme case is when $en = 1$ and $\mathbb{O}(out)$ both happen together only at the $(n+k)^{th}$ time frame, which means $en = 0$ holds until the point where $\mathbb{O}(out)$ holds. Then the input at the n^{th} time frame is observable at the $(n+k)^{th}$ time frame because it has been kept by the FF until this point.

When verifying the clock-gating condition on a target gated FF vertex, in which the enable signal has been changed from en_{old} to en_{new}, the observable condition of the input of the target gated FF is

$$\mathbb{O}(in) = [en_{old}] \wedge \{\mathbf{X}(\mathbb{O}(out)) \vee \mathbf{X}[(\neg en_{new})\mathbf{U}(\mathbb{O}(out))]\}.$$

This depends on both en_{old} and en_{new}. This LTL formula has two parts: the term $[en_{old}]$ before the conjunction ensures that the current input value is received, while the term $\{\mathbf{X}(\mathbb{O}(out)) \vee \mathbf{X}[(\neg en_{new})\mathbf{U}(\mathbb{O}(out))]\}$ guarantees the value saved in the FF can be observed at POs before being replaced by newer values. For the first part, although en_{new} might reduce the frequency of receiving new input, it can also significantly modify the sequential behavior, so using en_{old} here is safely conservation when a FF input is observable. Because there is no extra information for precise conditions of essential updates, we need to use en_{old} here to make sure we capture all essential updates on the target FF. Also, we must use en_{new} in the second part of this formula to verify the circuit after clock-gating. Because en_{new} can keep the saved value longer than en_{old} does, we need to examine if the out-of-date situation resulted by en_{new} can be observable. That is, $\mathbb{O}(in)$ cannot be constructed based only on en_{new} or only on en_{old}.

The example in Figure 5.2 demonstrates how a false positive can occur unless both en_{old} and en_{new} are used.

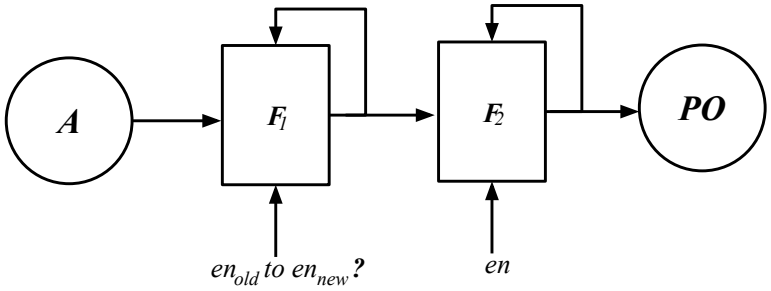


Figure 5.2: Example DG demonstrating an observable condition for the target gated FF.

For the target FF, F_1, with input in, where the enable signal is modified from en_{old} to en_{new}, the proposed synthesis is valid only if the "observable" property

$$\mathbf{G}((en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbb{O}(in)) \tag{5.1}$$

holds.

Example: In Figure 5.2, the enable signal for gated F_1 has been changed from en_{old} to en_{new}. To check legality, we need to prove the observable property,

$$\mathbf{G}\{(en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbb{O}(A)\}$$

on the circuit.

Let en_{old} = constant-1 and en_{new} = constant-0. If we formulate $\mathbb{O}(A)$ with en_{new} only, then $\mathbb{O}(A) = (en_{new}) \wedge ... = False$. Therefore, the target property 5.1 always holds. However, this clock-gating is illegal because the I/O behavior has been changed; F_1 would hold its value forever. Hence for the target FF F_1, the observable condition for the input A needs to include en_{old}.

On the other hand, if $\mathbb{O}(A)$ is formulated only with en_{old}, then

$$\begin{aligned}\mathbb{O}(A) &= [en_{old}] \wedge \{\mathbf{X}(\mathbb{O}(F_1)) \vee \mathbf{X}[(\neg en_{old})\mathbf{U}(\mathbb{O}(F_1))]\} = \mathbf{X}(\mathbb{O}(F_1)) \\ &= \mathbf{X}\{[en] \wedge [\mathbf{X}(\mathbb{O}(F_2)) \vee ...]\} \\ &= \mathbf{X}(en),\end{aligned}$$

because en_{old} = constant-1, $[(\neg en_{old})\mathbf{U}(\mathbb{O}(F_1))]$ is always $False$. Also $\mathbb{O}(F_2) = True$. Thus the observable property 5.1 becomes

$$\mathbf{G}\{(en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbf{X}(en)\},$$

which fails to catch the case where the saved out-of-date data (caused by $\mathbf{X}(en)$) can be observable after the next time frame.

Therefore, $\mathbb{O}(A)$ needs to include both en_{old} and en_{new} and should be formulated as

$$[en_{old}] \wedge \{\mathbf{X}(\mathbb{O}(F_1)) \vee \mathbf{X}[((\neg en_{new}))\mathbf{U}(\mathbb{O}(F_1))]\}.$$

When en_{old} = constant-1 and en_{new} = constant-0, this evaluates to $True$. Thus, the observable property 5.1 becomes $\mathbf{G}(True \Rightarrow False)$, which is always violated. Therefore, for the example in Figure 5.2, the proposed clock-gating (from en_{old} = constant-1 to en_{new} = constant-0) can never satisfy observable clock-gating conditions.

5.3.2 Property Formulation

The algorithm in Algorithm 5.1, given a target set of FFs, formulates an observable condition for its input $\mathbb{O}(in)$. Its negation is a sufficient non-observable condition. It is expected to cover $en_{old} \wedge \neg en_{new}$, i.e. $(en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbb{O}(in)$. This can be used to verify that en_{new} only turns off the clock when the input for the target FF is not observable.

The function *collectObserve*(...) in Algorithm 5.1 recursively constructs an observable condition for a target signal at a certain time frame by exploring the extended fanout cone until reaching either the depth limit (depth), POs, or some visited vertices.

The term **depth** indicates how deeply the recursive algorithm can go. When **depth** $= 0$, the algorithm just goes across one time frame from a FF vertex input to its output. By returning $True$ immediately, the FF is interpreted as a primary output. When **depth** < 0, this algorithm keeps exploring fanout cones until reaching primary outputs. Some paths might not go to primary outputs, resulting in an infinite recursion and some visited vertices are examined again. Hence the function *analyzeLoop*(...) is used to determine how to handle FFs in loops. Additional discussion about infinite recursion is in Section 5.3.3.

Algorithm 5.1 Observable Condition Construction: *collectObserve*(**DG**, **target**, **depth**)

Require: **DG**: a dependency graph, **target**: a signal, **depth**: the number of explored time frames.

Ensure: **P**: an LTL property

1: **if** *visited*(**target**) **then**
2: **return** *analyzeLoop*(**target**)
3: **if** **depth** $= 0$ **then**
4: **return** $True$
5: **switch** *type*(*outputV*(**target**)) **do**
6: **case** Primary Output:
7: **return** $True$
8: **case** Standard FF:
9: $O_{out} = collectObserve(\mathbf{DG}, output(\mathbf{target}), \mathbf{depth\text{-}1})$
10: **return** $\mathbf{X}(O_{out})$
11: **case** Transparent Block: ▷ as in_i
12: $O_{out} = collectObserve(\mathbf{DG}, output(\mathbf{target}), \mathbf{depth})$
13: **return** $O_{out} \wedge (sel = assign_i)$
14: **case** Combinational Cloud or Signal Branch:
15: **for all** $output_i$ of **target do**
16: $\mathbf{O} = \mathbf{O} \cup collectObserve(\mathbf{DG}, output_i, \mathbf{depth})$
17: **return** $\bigvee(\mathbf{O})$
18: **case** Gated FF:
19: $O_{out} = collectObserve(\mathbf{DG}, output(\mathbf{target}), \mathbf{depth\text{-}1})$
20: **return** $(en_{old}) \wedge (\mathbf{X}(O_{out}) \vee \mathbf{X}[(\neg en_{new})\mathbf{U}(O_{out})])$

The observable condition formulated here is tight in the sense that it captures all possible data dependencies based on analyzing control logic only. It is insufficient for true observablity because data flow could be blocked also by some combinational logic.

Example. In the DG in Figure 4.3, assume en_{old} for F_2 is constant-1 and $en_{new} = s_0$. We need to check if $\mathbf{G}(\neg s_0 \Rightarrow \neg\mathbb{O}(A_0))$ i.e. $\mathbf{G}(\mathbb{O}(A_0) \Rightarrow s_0)$, where $\mathbb{O}(A_0) = en_{old} \wedge \{\mathbf{X}(\mathbb{O}(F_2)) \vee \mathbf{X}[\neg s_0\mathbf{U}(\mathbb{O}(F_2))]\}$, $en_{old} = True$ and $\mathbb{O}(F_2) = \mathbb{O}(A) = \mathbb{O}(I) \wedge s_1$. Because $\mathbb{O}(I) = True$, $\mathbb{O}(F_2) = s_1$. Because, $s_1 = \mathbf{Y}(s_0)$ or $s_1 = \mathbf{Z}(s_0)$, $[\neg s_0\mathbf{U}(\mathbb{O}(F_2))] = [\neg s_0\mathbf{U}(s_1)]$ is $False$. Hence $\mathbb{O}(A_0) = \mathbf{X}(\mathbb{O}(F_2)) = \mathbf{X}(s_1)$. Thus the property $\mathbf{G}(\mathbf{X}s_1 \Rightarrow s_0)$ holds (regardless of the initial condition of s_1.) Hence s_0 is a legal clock-gating condition for F_2.

5.3.3 Infinite Recursion

When formulating the observable condition without a depth limit, a loop of FFs can cause an infinite recursion. To ensure termination, all side paths along the loop need to be examined.

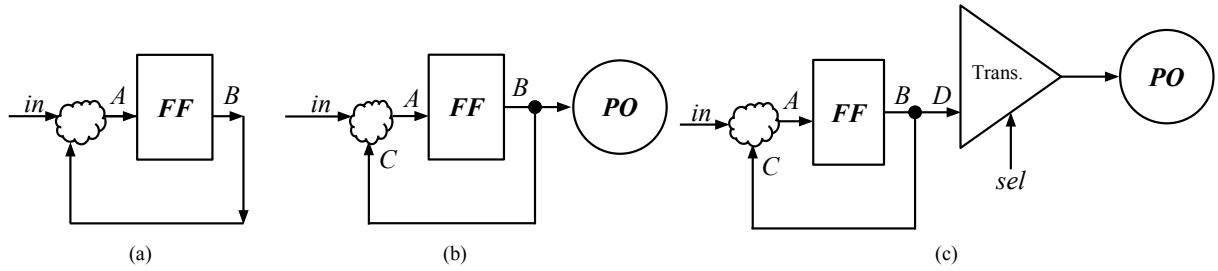


Figure 5.3: Three cases when deriving observable conditions for FFs in loops.

Consider the three examples in Figure 5.3. Each has a sequential loop. In the examples, different side paths are connected with branch vertices, resulting different properties.

For Example (a),

$$\mathbb{O}(in) = \mathbb{O}(A) = \mathbf{X}(\mathbb{O}(B)) = \mathbf{X}(\mathbb{O}(A)) = \mathbf{X}(\mathbf{X}(\mathbb{O}(B))),$$

and so on. This recursion cannot terminate by itself, because no primary output can be reached along this loop. Hence, it is correct to return constant $False$ when the algorithm reaches a visited vertex that is in a loop without a path to a primary outputs. However, if there is a sequence of FFs on this loop, where a visited vertex is not reached before *depth* is reached, the algorithm will return $True$ when reaching *depth*. That is, without analyzing the loop, the formulated observable condition is larger than the real situation.

For Example (b), the observable condition of in is formulated as follows:

$$\mathbb{O}(in) = \mathbb{O}(A) = \mathbf{X}(\mathbb{O}(B)) = \mathbf{X}(True \vee \mathbb{O}(C)) = True.$$

That is, the infinite loop has a side path to the primary output. Thus, when any of the side paths can reach primary outputs without any assignment requirements (for controls of transparent blocks or gated FFs), the infinite recursion can be resolved naturally.

Example (c) includes a transparent block on a side path of the loop. For this case,

$$\mathbb{O}(in) = \mathbb{O}(A) = \mathbf{X}(\mathbb{O}(B)) = \mathbf{X}(\mathbb{O}(D) \vee \mathbb{O}(C)),$$

where $\mathbb{O}(D) = (sel = assign_D) \wedge True$ and $\mathbb{O}(C) = \mathbb{O}(A)$. If the recursion keeps going, the property is

$$\mathbb{O}(in) = \mathbf{X}(sel = assign_D) \vee \ ... \vee \mathbf{X}^n(\mathbb{O}(A)),$$

which might never converge. If we formulate the property with a depth limit, it will reach a point where $\mathbb{O}(A)$ is assigned as $True$, so the whole property is $True$. Hence for infinite recursion cases, the observable condition for the visited vertex should be regarded as $True$.

In summary, when formulating the observable condition for a vertex involved in a loop, it can be $False$ only when there is no side path to the primary outputs; otherwise it should be constant $True$.

5.4 Satisfiability Clock-Gating Condition

Satisfiability clock-gating aims at turning off clocks for FFs when their input data is the same as that saved already in the FFs. This is related to control conditions and data dependencies of fanin cones from the past up to now. Common Linear Temporal Logic (LTL) operators are insufficient to describe this because some *past* operators must be included.

5.4.1 Update Condition

For a target FF vertex, where the enable signal is modified from en_{old} to en_{new}, satisfiability clock-gating requires that two properties need to be checked, *up-to-date* and *satisfiability*. Both of these properties use the notion of an update condition.

A set of signals is said to *update* when the signal values are different from those in the previous time frame. This is denoted by $\mathbb{U}(signal)$.

For each vertex of a DG, the update condition of each output depends on the update conditions of the vertex inputs, as well as related control signals. For each type of vertex, Table 5.2 lists the update condition for outputs in the current time frame, $\mathbb{U}(out)$. Note that this formulation leaves out the cases where signals may be "updated" but to the same values due to combinational logic, i.e. only control logic is used to determine the update condition.

Vertex Type	Update Condition for *out*: $\mathbb{U}(out)$
Primary Input	$True$
Constant	$\mathbf{Z}(False)$
Primary Output	N/A
Standard FF	$\mathbf{Z}(\mathbb{U}(in))$
Transparent Block	$\bigvee_i [(sel = assign_i) \wedge \mathbb{U}(in_i)] \bigvee [sel \neq \mathbf{Y}(sel)]$
Combinational Cloud	$\bigvee_i \mathbb{U}(in_i)$
Signal Branch	$\mathbb{U}(in)$
Gated FF	$\mathbf{Z}\{[en] \wedge [\mathbb{U}(in) \vee \mathbf{Y}([\neg en]\mathbf{S}[\mathbb{U}(in) \wedge \neg en])]\}$

Table 5.2: Update condition for the output of each type of vertex.

As shown in Table 5.2,

- PIs update in each cycle, independent of the actual input patterns.
- A constant is always identical to its previous cycle (not update), but for the first time frame, it updates from unknown to a certain value. Hence we use "invariant yesterday" $\mathbf{Z}$ to make sure it is $True$ in the first time frame.
- There is no output signal for PO vertices, so $\mathbb{U}(out)$ is not applicable.
- A standard FF updates in the current time frame depending on the update condition of its data input in the previous time frame. If the input data updates at cycle n

($\mathbb{U}(in) = True$ at cycle = n), then the output of the FF will update at cycle *n+1*, independent of the actual values. **Z** is used here because the initial conditions of FFs are not constrained in this formulation (the same idea will be applied to gated FFs.)

- The output of a transparent block updates when either (1) at least one of the support data inputs updates (the output can depend on more than one input data), or (2) the selection of input sources is different from that in the previous cycle. Here "yesterday" **Y** is used because the update conditions for other vertices (PIs, constants and FFs) guarantee $\mathbb{U}(in_i)$ must be $True$ in the first time frame; this also holds for the output of each transparent block.

- The update condition for each output of a combinational cloud is the union of the update conditions for each input to the combinational cloud, $\mathbb{U}(in_i)$; when any of these inputs update, the output must update, independent of the combinational logic. A signal branch vertex behaves like a combinational vertex.

- A gated FF can receive and update to a new value only when $en = 1$ in the previous time frame. Thus, either (1) the input data must update ($\mathbb{U}(in) = True$), or (2) the FF has not *updated* since its input was last updated, i.e. $\mathbf{Y}([\neg en]\mathbf{S}[\mathbb{U}(in) \wedge \neg en])$ is valid in the previous time frame.

When $en_{old} \neq$ constant-1, it is possible that the target FF can be out-of-date in the golden model. Therefore the target FF needs to be examined for being up-to-date. The following "up-to-date" property,

$$\mathbf{G}(\mathbb{U}(in) \Rightarrow en_{old}), \tag{5.2}$$

means anytime *in* updates, this FF receives and updates to the new value. Otherwise, this FF can be out-of-date.

If Formula 5.2 holds, en_{new} can be checked that it only additionally turns off the clock when the input data remains the same, i.e. the "satisfiability" property

$$\mathbf{G}((en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbb{U}(in)), \tag{5.3}$$

holds. Then the change from en_{old} to en_{new} is legal for satisfiability clock-gating.

To illustrate this, consider the gated FF shown in Figure 5.4: The output of the gated FF is updated at the n^{th} time frame only when $en = 1$ at the $(n-1)^{th}$ time frame. Then there are two cases allowing the FF to receive new values: (1) at the $(n-1)^{th}$ time frame, the input data updates and is different from its previous value, or (2) the input data updates at a previous $(n-k)^{th}$ time frame while $en = 0$ at that time. At this point, the FF is out-of-date. *en* keeps being 0 before the $(n-1)^{th}$ time frame, so the "since" property holds. Then the FF receives the input value due to $en = 1$ at the $(n-1)^{th}$ time frame, which can be different from the value kept in the FF since the $(n-k)^{th}$ time frame. Hence the output of this FF updates at the n^{th} time frame.

As mentioned, before verifying a clock-gating synthesis (from en_{old} to en_{new}) on a target FF, en_{old} needs to be examined. There are two cases:

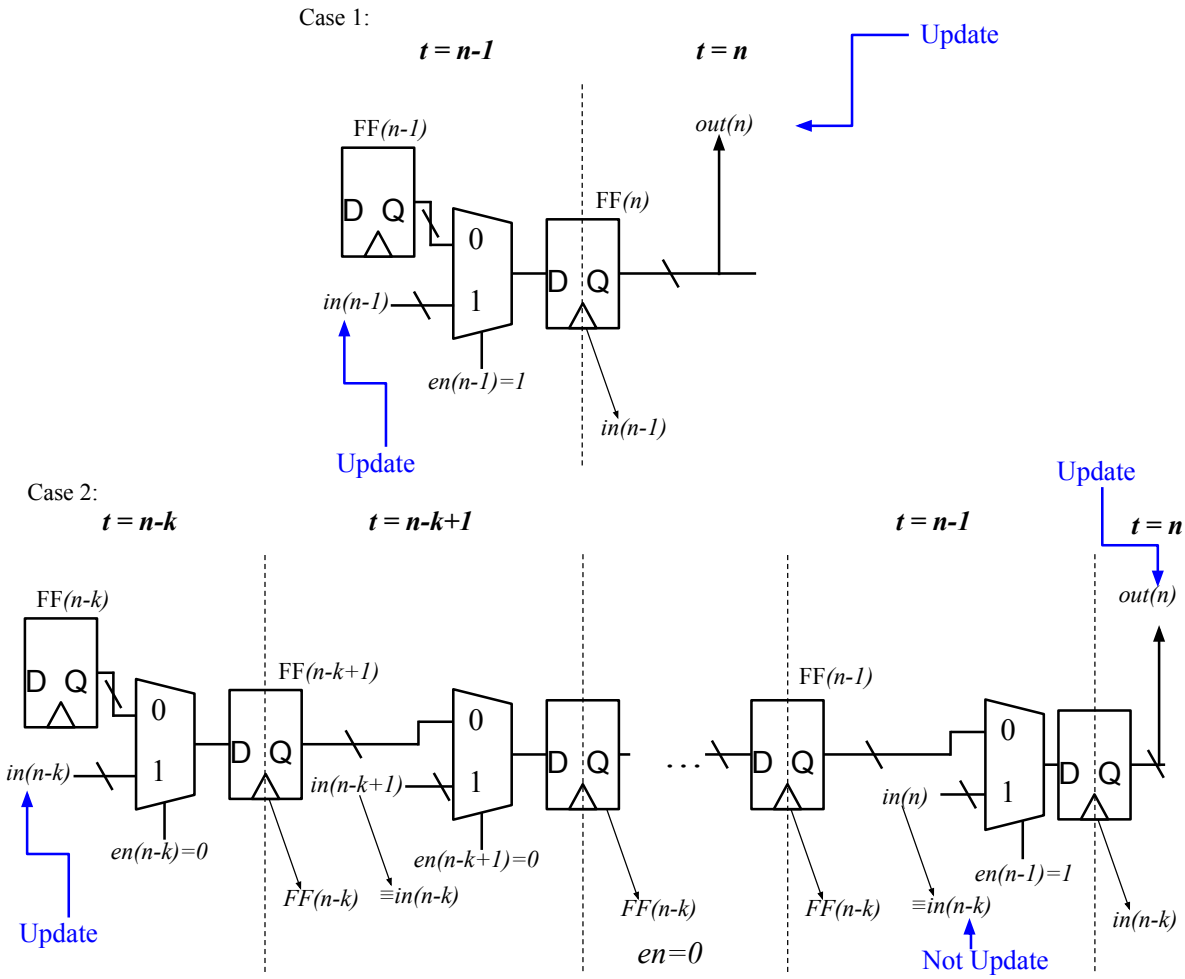


Figure 5.4: An example to demonstrate how the ouput of a gated FF can be updated.

1. $\mathbf{G}(\mathbb{U}(in) \Rightarrow en_{old})$ holds. Then $(en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbb{U}(in)$, i.e. en_{new} only additionally turns off the clock when $\mathbf{G}\{(en_{old} \wedge \neg en_{new}) \Rightarrow \neg\mathbb{U}(in)\}$ holds and the change from en_{old} to en_{new} is legal for satisfiability clock-gating. Also, the target FF is always up-to-date.

2. $\mathbf{G}(\mathbb{U}(in) \Rightarrow en_{old})$ fails. Even when en_{new} is proposed by a satisfiability clock-gating condition, it is necessary to verify if en_{new} results in extra observable out-of-date conditions (i.e. when $en_{new} = 0$, the FF can be out-of-date.) Hence, for a target FF which fails the up-to-date property, it needs to be verified that the proposed en_{new} satisfies Formula 5.1, the observable property. This will be illustrated in the examples discussed in Section 5.5.

5.4.2 Property Formulation

Algorithm 5.2 Update Condition Construction: *collectUpdate*(**DG**, **target**, **depth**)

Require: DG: a dependency graph, **target**: a vertex output, **depth**: the number of backtracking time frames.

Ensure: P: a PLTL property

1: **if** *visited*(**target**) or (**depth** = 1 and **target** belongs to FF types) **then**
2: **return** $True$
3: **switch** type(**target**) **do**
4: **case** Primary Input:
5: **return** $True$
6: **case** Constant:
7: **return** $\mathbf{Z}(False)$
8: **case** Standard FF:
9: $U_{in} = collectUpdate(\mathbf{DG}, inputV(\mathbf{target}), \mathbf{depth\text{-}1})$
10: **return** $\mathbf{Z}(U_{in})$
11: **case** Transparent Block:
12: **for all** $inputV_i$ of **target do**
13: $U^i_{in} = collectUpdate(\mathbf{DG}, inputV_i, \mathbf{depth})$
14: **return** $\bigvee_i [(sel = assign_i) \wedge U^i_{in}] \bigvee [sel \neq \mathbf{Y}(sel)]$
15: **case** Combinational Cloud or Signal Branch:
16: **for all** $inputV_i$ of **target do**
17: $\mathbf{U} = \mathbf{U} \cup collectUpdate(\mathbf{DG}, inputV_i, \mathbf{depth})$
18: **return** $\bigvee(\mathbf{U})$
19: **case** Gated FF:
20: $U_{in} = collectUpdate(\mathbf{DG}, inputV(\mathbf{target}), \mathbf{depth\text{-}1})$
21: **return** $\mathbf{Z}\{[en] \wedge [U_{in} \vee \mathbf{Y}([\neg en]\mathbf{S}[U_{in} \wedge (\neg en)])]\}$

The proposed algorithm of formulating the update condition for a target vertex (output) is shown in Algorithm 5.2. The function *collectUpdate*(...) constructs the update condition for a vertex output by recursively exploring the update conditions of all support vertices. Each type of vertex has a specialized process handled by one of the case values.

We also use the term **depth** to control how deep the recursive algorithm can go. When **depth** < 0, this algorithm terminates only when reaching primary inputs, constants, or already visited vertices (loops). When working on a loop, which can result in infinite recursion, this algorithm returns $True$ immediately. More explanation and examples for infinite recursion are discussed in Section 5.4.3.

Calling *collectUpdate*(...) for a vertex FF means that the exploration goes across one time frame, so the input depth should be reduced by one. When **depth** $= 1$, it explores combinational logic only and returns $True$ when reaching FFs, meaning those FFs are in-

terpreted as free primary inputs. Generally, a larger **depth** can result in more restricted (better) update conditions, which saves more power. However, a larger **depth** also implies possibly more effort is spent in formulating and proving the property. The parameter **depth** can be used to tradeoff power consumption and overall verification effort.

Example: Suppose for the circuit in Figure 5.5 that a clock-gating synthesis method proposes a control signal en_{new} for F_3. Since en_{old} is constant-1, Formula 5.2 obvously holds, so only Formula 5.3 needs to hold. This become $\mathbf{G}(\neg en_{new} \Rightarrow \neg\mathbb{U}(C))$ and if it holds, introducing en_{new} is legal.

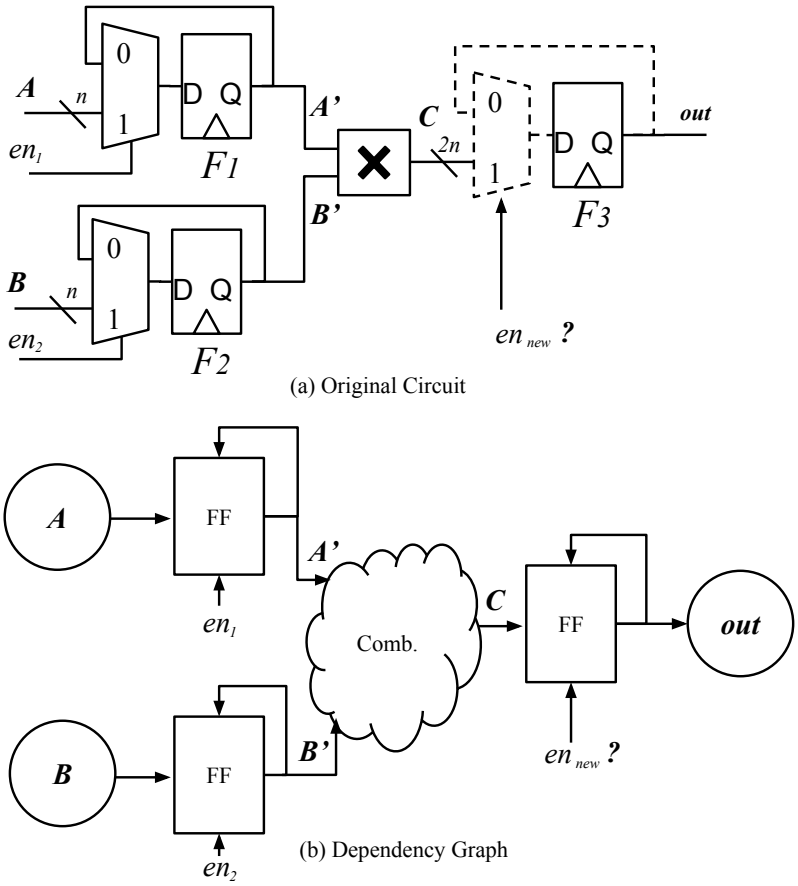


Figure 5.5: An example to demonstrate how a satisfiability clock-gating condition can be verified.

To formulate $\mathbb{U}(C)$, the algorithm needs to collect $\mathbb{U}(A')$ and $\mathbb{U}(B')$. The update condition $\mathbb{U}(A')$ depends on $\mathbb{U}(A)$, which is $True$ all the time, so

$$\mathbb{U}(A') = \mathbf{Z}\{en_1 \wedge [True \vee \mathbf{Y}(\neg en_1)\mathbf{S}[True \wedge \neg en_1]\} = \mathbf{Z}(en_1).$$

Similarly, $\mathbb{U}(B') = \mathbf{Z}(en_2)$. Therefore,

$$\mathbb{U}(C) = \mathbf{Z}(en_1) \vee \mathbf{Z}(en_2) = \mathbf{Z}[(en_1) \vee (en_2)].$$

Based on the formula, we can verify if $(\mathbb{U}(C) \Rightarrow en_{new})$ is $True$ for all time frames.

5.4.3 Infinite Recursion

Considering possible initial conditions and combinational logic, the update condition for a vertex in a loop should be assigned as $True$ to make sure it is sufficient.

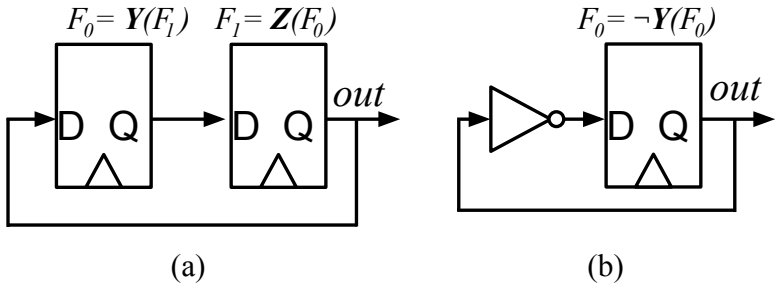


Figure 5.6: Two examples with FFs on sequential loops. The initial states and combinational logic are included.

Example: In Figure 5.6(a) there are two FFs with opposite initial values in a loop. Without information about the initial states, the update condition of *out* is formulated $\mathbb{U}(out) = \mathbb{U}(F_1) = \mathbf{Z}(\mathbb{U}(F_0)) = \mathbf{Z}(\mathbf{Z}(\mathbb{U}(F_1)))$, and so on. Because of the invariant yesterday $\mathbf{Z}$, the infinite formulation results $\mathbb{U}(out)$ as constant $True$. Also, looking into the original circuit, it shows that the values of F_0 and F_1 keep oscillating. That is, the values of F_1 and F_2 are always different from the previous time frame. Hence $\mathbb{U}(out)$ must be $True$. Figure 5.6(b) has a feedback loop with an invertor. The value of F_0 keeps updating in each time frame, regardless of its initial condition. Thus $\mathbb{U}(out)$ is $True$.

As mentioned before in this paper, initial conditions and combinational logic are not analyzed for formulating properties. To avoid false positives, we need to take the most conservative assumption, meaning FFs in sequential loops always update. Hence when reaching a revisited vertex in a loop, the algorithm in Figure 5.2 returns $True$ immediately.

5.5 Illustrative Examples

More examples are discussed to demonstrate how the proposed algorithms work in practice.

5.5.1 Update Condition for Redundant Controls

Generally, the proposed properties for disabling the clock are sufficient, but could be stronger. This is because the actual values of data are ignored. In some cases with redundant control paths, the proposed derivation can over-approximate the update condition even when input data values are not changed. Moreover, when there exists redundancy in control logic, the proposed algorithm will formulate weaker properties, where some non-updates or non-observable conditions are not considered.

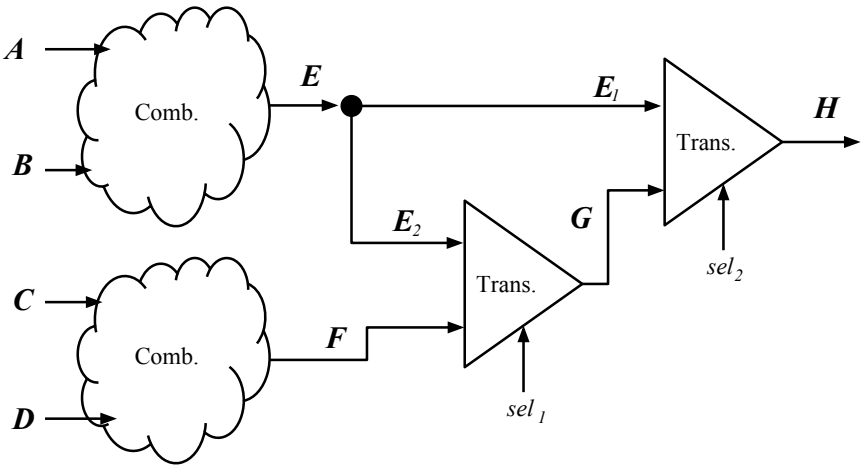


Figure 5.7: The formulated property includes more cases for updates than necessary.

Example: In Figure 5.7 $sel_2 = as_E$ is used to mean H is transparent from E_1 under the assignment $sel_2 = as_E$, while $sel_2 = as_G$ means H is transparent from G. The update condition of H is constructed as :

$$\mathbb{U}(H) = [(sel_2 = as_E) \wedge \mathbb{U}(E_1)] \vee [(sel_2 = as_G) \wedge \mathbb{U}(G)] \vee [sel_2 \neq \mathbf{Y}(sel_2)],$$

where $\mathbb{U}(E_1) = \mathbb{U}(E)$, and

$$\mathbb{U}(G) = [(sel_1 = as_E) \wedge \mathbb{U}(E_2)] \vee [(sel_1 = as_F) \wedge \mathbb{U}(F)] \vee [sel_1 \neq \mathbf{Y}(sel_1)].$$

Assume $\mathbb{U}(E)$ and $\mathbb{U}(F)$ are $False$; then $\mathbb{U}(G) = sel_1 \neq \mathbf{Y}(sel_1)$ and $\mathbb{U}(E_1) = False$. The update condition for H becomes

$$[(sel_2 = as_G) \wedge (sel_1 \neq \mathbf{Y}(sel_1))] \vee (sel_2 \neq \mathbf{Y}(sel_2)).$$

This formula implies that anytime when sel_2 changes, H updates. However, it is possible that sel_1 keeps selecting E_2 ($sel_1 = as_E$), $G \equiv E_2$ all the time, while sel_2 changes (H switches between E_1 and G). Hence H is identical to E and never updates ($\mathbb{U}(E) = False$). In this case, the update condition for H,

$$[(sel_2 = as_G) \wedge (sel_1 \neq \mathbf{Y}(sel_1))] \vee (sel_2 \neq \mathbf{Y}(sel_2)),$$

covers some cases where H does not update. Hence the resulting clock-gating condition is weaker.

In this example, the two transparent blocks (two groups of MUXes controlled by s_1 and s_2) can be merged into one transparent block, which is controlled by sel_1 and sel_2 together, and the output H switches between E and F based on the assignments of s_1 and s_2. Based on the simplified circuit, when formulating the update condition for H, the proposed algorithm can capture the exact condition based on the control logic.

Hence when the control logic can be simplified (with some redundant structure), the proposed algorithm might formulate an over-approximation of the update condition, i.e. there exists a case where $\mathbb{U}(signal)$ evaluates to $True$ but $signal$ keeps the same regardless of the actual value, which results in a weaker clock-disabling condition.

5.5.2 Dominating Gated FFs

When a data path crosses multiple gated FFs, the notion of *dominance* arises in the update condition of the target signal.

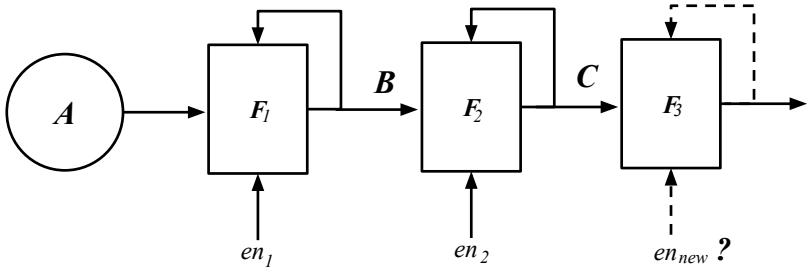


Figure 5.8: An example with a sequence of gated FFs before the target FF.

Example: In Figure 5.8, F_1 and F_2 have been clock-gated by en_1 and en_2, respectively. To formulate an update condition for F_3, the algorithm gives

$$\mathbb{U}(C) = \mathbb{U}(F_2) = \mathbf{Z}\{en_2 \wedge [\mathbb{U}(F_1) \vee \mathbf{Y}(\neg en_2 \mathbf{S}[\mathbb{U}(F_1) \wedge \neg en_2])]\},$$

where $\mathbb{U}(F_1) = \mathbf{Z}\{[en_1 = 1] \wedge [\mathbb{U}(A) \vee ...]\} = \mathbf{Z}(en_1 = 1)$ (because $\mathbb{U}(A) = True$.) Hence

$$\mathbb{U}(F_2) = \mathbf{Z}\{[en_2] \wedge [\mathbf{Z}(en_1) \vee \mathbf{Y}([\neg en_2]\mathbf{S}[\mathbf{Z}(en_1) \wedge \neg en_2])]\}.$$

Suppose $\mathbf{G}(\mathbf{Z}(en_1) \Rightarrow en_2)$. Then $([\neg en_2]\mathbf{S}[\mathbf{Z}[(en_1) \wedge \neg en_2])$ is $False$ because $[\mathbf{Z}(en_1) \wedge \neg en_2] = 0$. Therefore, $\mathbb{U}(F_2) = \mathbf{Z}[(en_2) \wedge \mathbf{Z}(en_1)] = \mathbf{Z}[\mathbf{Z}(en_1)]$ (due to $\mathbf{Z}(en_1) \Rightarrow en_2$.) For this case, the update condition for F_2 only depends on en_1 and thus, F_2 must be up-to-date when F_1 updates. Hence, en_1 *dominates* this data path.

On the other hand, suppose $\mathbf{G}(\mathbf{Z}(en_1) \Rightarrow en_2)$ fails. Then it is possible that F_1 updates ($\mathbf{Z}(en_1)$) but F_2 is out-of-date ($\neg en_2$). Thus, en_2 is more critical for update F_2. We could over-approximate the update condition as $\mathbb{U}(F_2) = \mathbf{Z}(en_2)$, because $en_2 = 1$ is required for updating F_2 in the next time frame, independent en_1. For this case, en_2 *dominates* the control of the data path from A to C.

The proposed algorithm inherently covers the concept of dominance, so analyzing adjacent gated FFs is unnecessary. However, this concept is useful in understanding the behavior of a data path.

5.5.3 Combining Satisfiability and Observability

As mentioned in Section 5.4.1, it is essential to verify the observable conditions when the target FF can be out-of-date with en_{old}. The next example illustrates this.

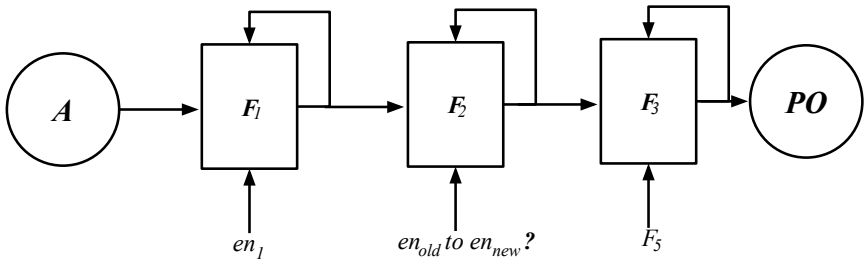


Figure 5.9: DG for all circuits in Figure 5.10.

Example: Figure 5.10(a) shows the original design. In 5.10(b), F_2 is clock-gated using satisfiability by $en_{new} = \mathbf{Z}(en_1)$ (the initial condition of F_4 should be $True$). Because $en_{old} = True$, this clock-gating condition can be verified with the property $\mathbf{G}(\neg en_{new} \Rightarrow \neg\mathbb{U}(F_1))$ where $\mathbb{U}(F_1) = \mathbf{Z}(en_1)$.

Figure 5.10(a) can be clock-gated also by an observability condition $en_{new} = \mathbf{X}(F_5) = en_2$ as in 5.10(c). Proving $\mathbf{G}(\neg en_{new} \Rightarrow \neg\mathbb{O}(F_1))$ justifies this. Here $\mathbb{O}(F_2) = F_5$ and
$\mathbb{O}(F_1) = [en_{old}] \wedge \{\mathbf{X}(\mathbb{O}(F_2)) \vee \mathbf{X}[(\neg en_2)\mathbf{U}(\mathbb{O}(F_2))]\}$.
Because $en_2 = \mathbf{X}(F_5)$, then $\mathbf{X}[(\neg en_2)\mathbf{U}(\mathbb{O}(F_2))]$ is $False$. Hence $\mathbb{O}(F_1) = \mathbf{X}(F_5)$ and the required property holds.

The two clock-gating conditions are valid when applied alone, but combining them can be problematic. Consider Figure 5.10(b), and let $en_{old} = \mathbf{Z}(en_1)$ and $en_{new} = \mathbf{Z}(en_1) \wedge en_2$. This can be justified by proving $\mathbf{G}([\mathbf{Z}(en_1) \wedge \neg en_2] \Rightarrow \neg\mathbb{O}(F_1))$. Here $\mathbb{O}(F_1) = [\mathbf{Z}(en_1)] \wedge \{\mathbf{X}(F_5) \vee \mathbf{X}[(\neg(\mathbf{Z}(en_1) \wedge en_2))\mathbf{U}(F_5)]\} = [\mathbf{Z}(en_1)] \wedge \{(en_2) \vee \mathbf{X}[(\neg(\mathbf{Z}(en_1) \wedge en_2))\mathbf{U}(F_5)]\}$, where $\mathbf{X}[(\neg(\mathbf{Z}(en_1) \wedge en_2))\mathbf{U}(F_5)]$ can be $True$. Therefore the property $\mathbf{G}([(\mathbf{Z}(en_1)) \wedge (\neg en_2)] \Rightarrow \neg\mathbb{O}(F_1))$ can fail, and the synthesis (from $\mathbf{Z}(en_1)$ to $\mathbf{Z}(en_1) \wedge en_2$) is illegal.

Now consider Figure 5.10(c) with $en_{old} = en_2$ and $en_{new} = \mathbf{Z}(en_1) \wedge en_2$. Verifying with just $\mathbf{G}([(en_2) \wedge (\neg\mathbf{Z}(en_1))] \Rightarrow \neg\mathbb{U}(F_1))$, where $\mathbb{U}(F_1) = \mathbf{Z}(en_1)$, leads to a false positive. Although this property holds, the proposed en_{new} is still illegal. The issue is, en_{old} has made F_2 out-of-date, so the extra constraint $\mathbf{Z}(en_1)$ can result in more out-of-date conditions; we also need to check $\mathbf{G}([(\neg\mathbf{Z}(en_1)) \wedge (en_2)] \Rightarrow \neg\mathbb{O}(F_1))$. This observable property can fail, because $\mathbb{O}(F_1) = en_2 \wedge \{\mathbf{X}(F_5) \vee \mathbf{X}[(\neg(\mathbf{Z}(en_1) \wedge en_2))\mathbf{U}(F_5)]\} = [\mathbf{Z}(en_1)] \wedge \{(en_2) \vee \mathbf{X}[(\neg(\mathbf{Z}(en_1) \wedge en_2))\mathbf{U}(F_5)]\}$, where $\mathbf{X}[(\neg(\mathbf{Z}(en_1) \wedge en_2))\mathbf{U}(F_5)]$ can be $True$. because en_1 and en_2 are independent. Hence $[\neg\mathbf{Z}(en_1) \wedge en_2] \Rightarrow \neg\mathbb{O}(F_1)$ can be $False$, the clock-gating synthesis (from en_2 to $\mathbf{Z}(en_1) \wedge en_2$) is therefore invalid.

Therefore, if the target FF can be out-of-date under en_{old}, it is necessary to use observable conditions to verify.

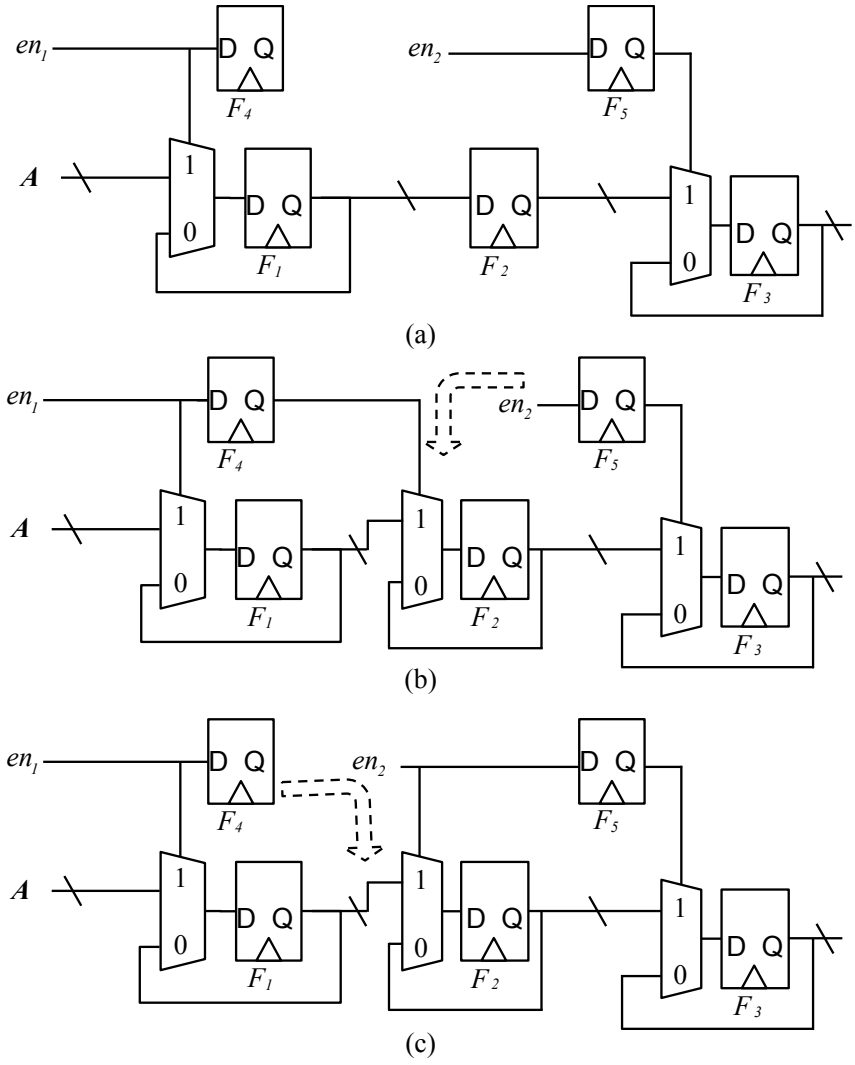


Figure 5.10: Circuit that can be clock-gated by satisfiability or by observable conditions separately.

5.5.4 Verification flow for Clock-Gating

A valid flow for verifying an arbitrary clock-gating synthesis following the discussion in the previous section is given in Figure 5.11.

For a gated FF vertex with input in and original enable signal en_{old}, to verify changing en_{old} to en_{new}, the first step is to check if the set of FFs are always up-to-date (Formula 5.2). If this passes, there is a chance that the proposed clock-gating is only based on the update condition of in (satisfiability clock-gating.) Hence it can be verified with Formula 5.3. If the target FF with en_{new} satisfies the satisfiability property, it is a legal clock-gating synthesis.

If the target FF vertex can be out-of-date with en_{old}, or it violates the satisfiability property, verification with the observability condition (Formula 5.1) is required. If this

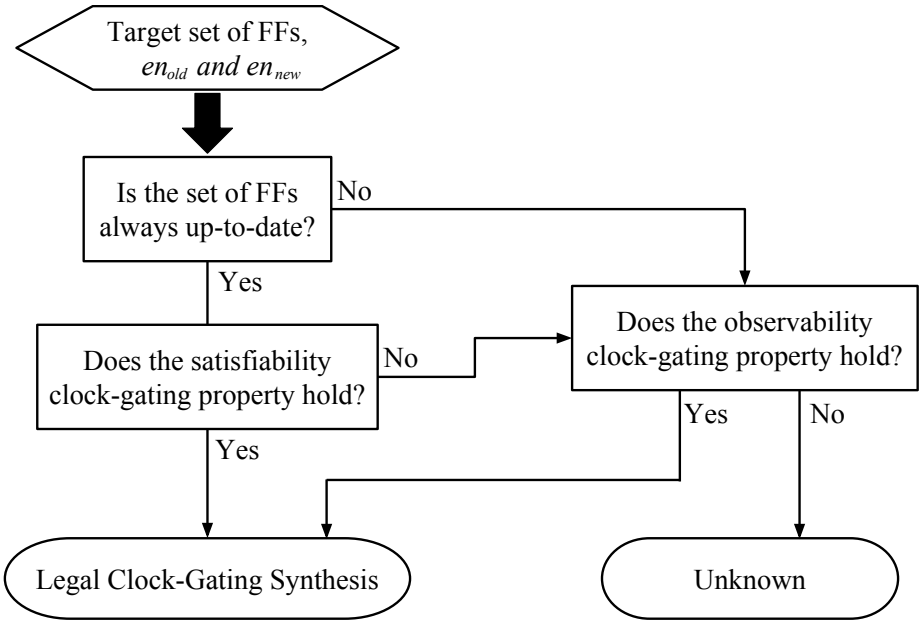


Figure 5.11: Verification flow for a target set of FFs.

holds, the clock-gating synthesis is valid; if it fails, the proposed synthesis still can be valid, but cannot be justified with only control signals and data dependencies; information about combinational logic is required.

Note that the update and observable conditions cannot be combined to make a single property as in

$$\mathbf{G}(en_{old} \wedge \neg en_{new} \Rightarrow \neg\mathbb{U}(in) \vee \neg\mathbb{O}(in)). \tag{5.4}$$

This cannot verify a case where the current input is not updating, but the target FF has been out-of-date before $en_{old} \wedge \neg en_{new}$ happens. Then the old out-of-date value can be kept for longer and become observable without checking.

Example: In Figure 5.10(c), $en_{old} = en_2$ and $en_{new} = \mathbf{Z}(en_1) \wedge en_2$. Based on the incorrect Formula 5.4, we could formulate $\mathbf{G}([\neg\mathbf{Z}(en_1) \wedge en_2] \Rightarrow \neg\mathbb{U}(F_1) \vee \neg\mathbb{O}(F_1))$, where $\mathbb{U}(F_1) = \mathbf{Z}(en_1)$. This property always holds, but the proposed clock-gating is illegal; using Formula 5.4 can lead to a false positive.

5.6 Proving on Circuits

The formulated sufficient properties for legal clock-gating, which fully consider the functionalities of control signals, must be proved for a sequential circuit. The LTL properties can be recast as new hardware property outputs to be proved using hardware model checkers.

Claessen et al. [10] provided methods for expressing LTL/PLTL formulae as circuits. including $\mathbf{G}(z \Rightarrow \{op\}(a)$ and $\mathbf{G}(z \Rightarrow [a\{op\}b])$, where $\{op\}$ is an arbitrary LTL or PLTL

operator. They constructed a hardware *monitor circuit* to represent each target property to be proved.

In this section, methods for deriving monitor circuits are reviewed only for those operators used in clock-gating properties: **X**, **Y**, **Z**, **S**, and **U**.

5.6.1 Formulating Past Properties on Circuits

For update conditions relating to the circuit's behavior from the past to now, monitor circuits for PLTL properties are constructed by adding FFs for some existing signals:

- $\mathbf{Y}a$: add a FF in front of a to delay it for one cycle, and initialize it to $False$.
- $\mathbf{Z}a$: add a FF in front of a to delay it for one cycle, and initialize it to $True$.
- $[a\ \mathbf{S}\ b]$: build a signal s which is delayed and supported by an extra FF with a feedback loop, such that $s = (\mathbf{Y}(s) \wedge a) \vee b$. The FF is initialized to $False$ because b must hold at least once. When b occurs, s becomes $True$ and starts to expect a. If the current b is $False$ and $\neg a$ happens before or now, this property fails. That is, only b can resolve the pending status.

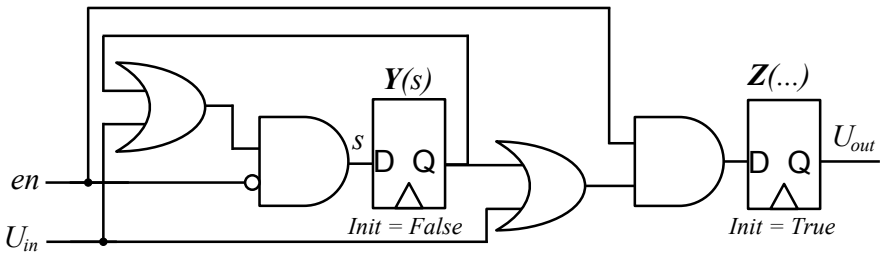


Figure 5.12: Circuit for $\mathbf{Z}\{[en] \wedge [\mathbb{U}(in) \vee \mathbf{Y}([\neg en]\mathbf{S}[\mathbb{U}(in) \wedge \neg en])]\}$ based on old enable en and the update condition of the input.

Example: Figure 5.12 shows the hardware for the update condition of a FF gated by en. The condition in PLTL is $\mathbf{Z}\{en \wedge [\mathbb{U}(in) \vee \mathbf{Y}([\neg en]\mathbf{S}[\mathbb{U}(in) \wedge \neg en])]\}$. This depends on U_{in}, the update condition of the data input of the gated FF, and en, the enable signal of the gated FF. Before combined with other parts, the "since" part, $([\neg en]\mathbf{S}[\mathbb{U}(in) \wedge \neg en])$, is constructed as follows: let $a = \neg en$ and $b = U_{in} \wedge \neg en$, then

$$\begin{aligned}[\neg en]\mathbf{S}[\mathbb{U}(in) \wedge \neg en] = a\ \mathbf{S}\ b &= (\mathbf{Y}(s) \wedge a) \vee b \\ &= (\mathbf{Y}(s) \wedge \neg en) \vee (U_{in} \wedge \neg en) \\ &= \neg en \wedge (\mathbf{Y}(s) \vee U_{in}),\end{aligned}$$

based on the distributivity of conjunction over disjunction. Hence the "since" term is built with a FF whose output represents $\mathbf{Y}(s)$ which is fed back to s, combined with $\neg en$ and U_{in}. After $\mathbf{Y}(s)$ is ORed with U_{in}, this is ANDed with en, and the whole formula is delayed by the other FF, which is initialized to $True$ to represent the $\mathbf{Z}$ in the front of the formula.

5.6.2 Formulating Future Properties as Circuits

Representing an observable condition is challenging because it relates to future circuit behaviors; thus extra FFs are added to be used to verify future properties.

In the following, we just describe the hardware constructions for the future properties, but refer to [10] for the more complicated constructions and their justifications. Free variables z_X and z_U are added to transform an LTL formula into an equi-satisfiable one. These need to satisfy conditions $\mathbf{G}(z_X \Rightarrow \mathbf{X}(a))$ or $\mathbf{G}(z_R \Rightarrow [a\mathbf{U}b])$ to represent something is pending in the future. Signals $pending_X$ or $pending_U$, are created, to indicate an event is expected (waited). To help in finding counterexamples, signals $failed_X$ or $failed_U$ are added as well. Signal $accept_U$ is created to present acceptance condition for the liveness concept of "until".Their functionalities are listed below:

- $\mathbf{G}(z_X \Rightarrow \mathbf{X}a)$: $pending_X = z_X$, while $failed_X = \neg\mathbf{Z}(False) \wedge \mathbf{Y}(pending_X) \wedge \neg a$. Thus, the "next" property fails when (1) it is not the first time frame, (2) $pending_X$ is $True$ in the previous time frame, and (3) a is $False$ in the current time frame.

- $\mathbf{G}(z_U \Rightarrow [a\mathbf{U}b])$: $pending_U$ comes with an extra FF as $[(z_U \vee \mathbf{Y}(pending_R)) \wedge \neg b]$, meaning when z_U is $True$ in the current time frame, or $pending_U$ was $True$ in the previous time frame, only b can relax the waiting condition. $failed_U$ can be expressed as $failed_U = pending_U \wedge \neg a$, referring to where $\neg a$ occurs when $pending_U$ is $True$. $accept_U = \neg pending_U$, meaning this property has never been activated or b has occurred.

For a gated FF,

$$\mathbb{O}(in) = [en_{old}] \wedge \{\mathbf{X}(\mathbb{O}(out)) \vee \mathbf{X}[(\neg en_{new})\mathbf{U}(\mathbb{O}(out))]\}.$$

An equi-satisfiable formula can be constructed by introducing a variable for each subformula as follows:

$$\begin{aligned}
&\mathbb{O}(in) = en_{old} \wedge (z_0 \vee z_1) \\
&\wedge \mathbf{G}(z_0 \Rightarrow \mathbf{X}(\mathbb{O}(out))) \\
&\wedge \mathbf{G}(z_1 \Rightarrow \mathbf{X}(z_2)) \\
&\wedge \mathbf{G}(z_2 \Rightarrow (\neg en_{new})\mathbf{U}(\mathbb{O}(out))).
\end{aligned}$$

To build O_{in}, the observable condition O_{out}, for the gated FF output, can be used with extra signals (z_i, $pending_i$, $failed_i$ and $accept_i$), en_{old} and en_{new} as shown in Figure 5.13. Using the formulation for each $pending$, $failed$ or $pending$ signal as described above, with

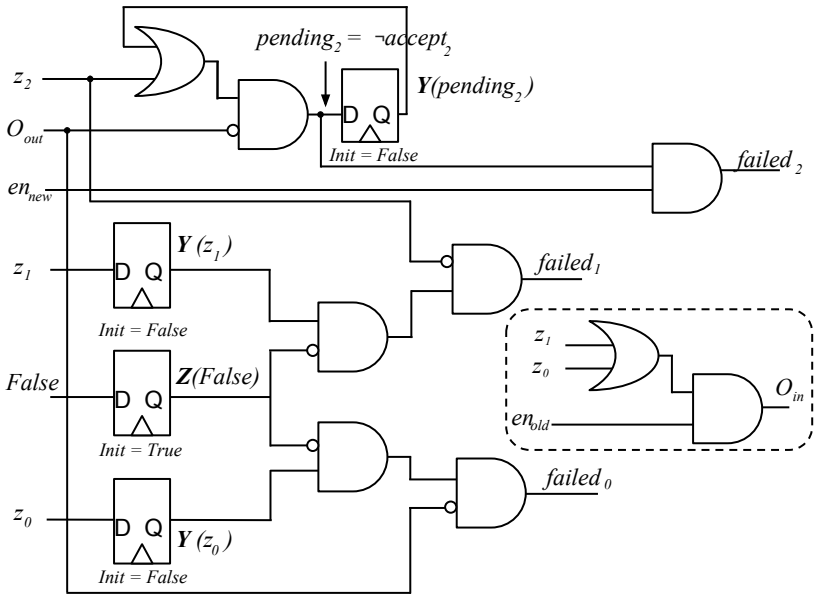


Figure 5.13: Circuits for the observable condition of a gated FF input based on its old and new enable signals and the observable condition of its output.

Z and **Y** representing FFs with initial values, the monitor hardware is built as:

$$pending_0 = z_0; \qquad failed_0 = \neg\mathbf{Z}(False) \wedge \mathbf{Y}(z_0) \wedge \neg O_{out}$$
$$pending_1 = z_1; \qquad failed_1 = \neg\mathbf{Z}(False) \wedge \mathbf{Y}(z_1) \wedge \neg z_2$$
$$pending_2 = [(z_2 \vee \mathbf{Y}(pending_2)) \wedge \neg O_{out}]; \qquad failed_2 = pending_2 \wedge en_{new};$$
$$accept_2 = \neg pending_2$$
$$O_{in} = en_{old} \wedge (z_0 \vee z_1).$$

After constructing all signals required by Formula 5.1, the algorithm proposed in [10] is used to verify the observability clock-gating condition. For further details, please refer to the reference cited.

5.7 Summary

In this chapter the clock-gating conditions on DGs are described in detail. For a target signal, both update and observability conditions are defined and formulated on DGs. Based on legal observability and satisfiability clock-gating conditions, a verification flow was proposed for checking a clock-gating conditions on a set of target FFs. A circuit-based approach to verify formulated LTL or PLTL properties was provided.

Compared to the properties formulated for the previous CG method outlined in Chapter 2, DGs utilize en_{new} and en_{old} to precisely describe the clock-gating synthesis applied on

circuits, while en_{old} is always assumed to be $True$ on CGs. Also, the legal clock-gating properties formulated on DGs are stronger than those on CGs, i.e. can turn off clocks more often. The circuit in Figure 5.14 demonstrates the differences.

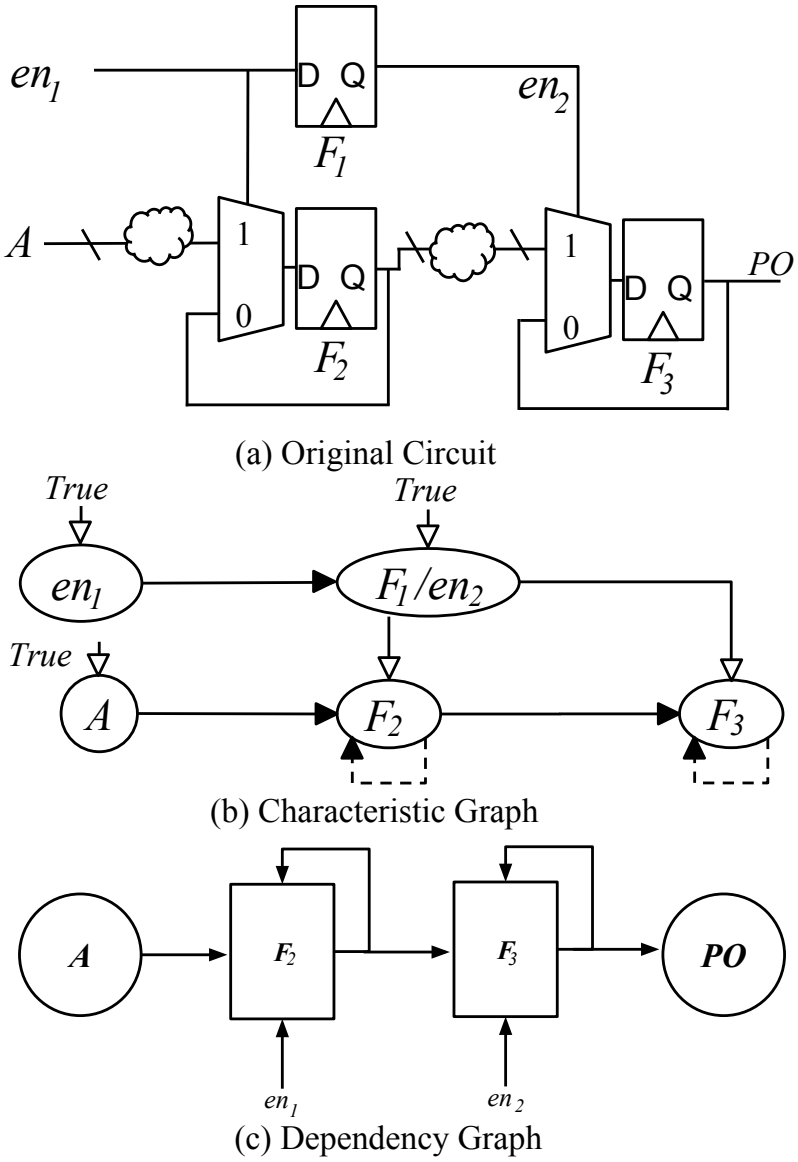


Figure 5.14: (a) A sequential circuit clock-gated using with both satisfiability and observability. (b) the corresponding characteristic graph, (c) the corresponding dependency graph

Consider the set of FFs F_2 in Figure 5.14, which is gated by en_1. To verify if en_1 on F_2 is a legal clock-gating condition (sequential redundancy), observability clock-gating is examined. The CG method proposes the sufficient condition as

$$CG : \mathbf{G}(\mathbf{X}en_2 \Rightarrow en_1).$$

The DG, assuming $en_{old} = True$, proposes a sufficient condition

$$DG : \mathbf{G}(\neg en_1 \Rightarrow \neg\mathbb{O}(A)) = \mathbf{G}(\neg en_1 \Rightarrow \neg(\mathbf{X}(\mathbb{O}(F_2)) \vee \mathbf{X}[(\neg en_1)\mathbf{U}(\mathbb{O}(F_2))]).$$

Because $\mathbb{O}(F_2) = en_2$, the CG condition implies that $[(\neg en_1)\mathbf{U}(en_2)]$ is $False$. Hence the CG condition CG limits the DG to $\mathbf{G}(\neg en_1 \Rightarrow \neg\mathbf{X}(en_2))$, which is identical to the CG condition. Thus a legal CG clock-gating condition is also valid according to the DG method. However, the "until" part of the DG condition provides additional legal conditions that are not accepted by the CG method.

Also, it is possible that F_3 is gated by a satisfiability clock-gating condition. To verify if en_2 on F_3 is legal, CG gives

$$CG : \mathbf{G}(en_1 \Rightarrow \mathbf{X}(en_2)),$$

which requires that the initial state of en_2 must be 1. DG gives

$$DG : \mathbf{G}(\neg en_2 \Rightarrow \neg\mathbb{U}(F_2)) = \mathbf{G}(\neg en_2 \Rightarrow \neg\mathbb{U}(\mathbf{Z}\{en_1 \wedge [\mathbb{U}(A) \vee \mathbf{Y}(\neg en_1\mathbf{S}[\mathbb{U}(A) \wedge \neg en_1])]\})).$$

Because $\mathbb{U}(A) = True$, then $\mathbb{U}(F_2) = \mathbf{Z}(en_1)$. Hence the whole property becomes

$$\mathbf{G}(\neg en_2 \Rightarrow \neg\mathbf{Z}(en_1)).$$

The property with the "$\mathbf{Z}$" operator considers both (1) the property with "$\mathbf{X}$" proposed by the CG method and (2) the initial state of en_2. Hence the property proposed by the DG method is more comprehensive.

Therefore, in comparison with the CG method in Chapter 2, the DG method can verify cases synthesized by more sophisticated clock-gating techniques.

Chapter 6

Sequential Equivalence Checking of Clock-Gated Circuits

For two sequential circuits, golden and revised (**G** and **R**), with mapped PIs and POs, SEC can be done similarly to the previous CG method in Chapter 2: (1) identify the additional clock-gating conditions on **R**, (2) verify if they are legal and (3) if so reduce them on **R** (because they are redundant). After removing the extra clock-gating signals, the revised design **R'** will be more similar to **G**, and hence SEC between **G** and **R'** is generally easier.

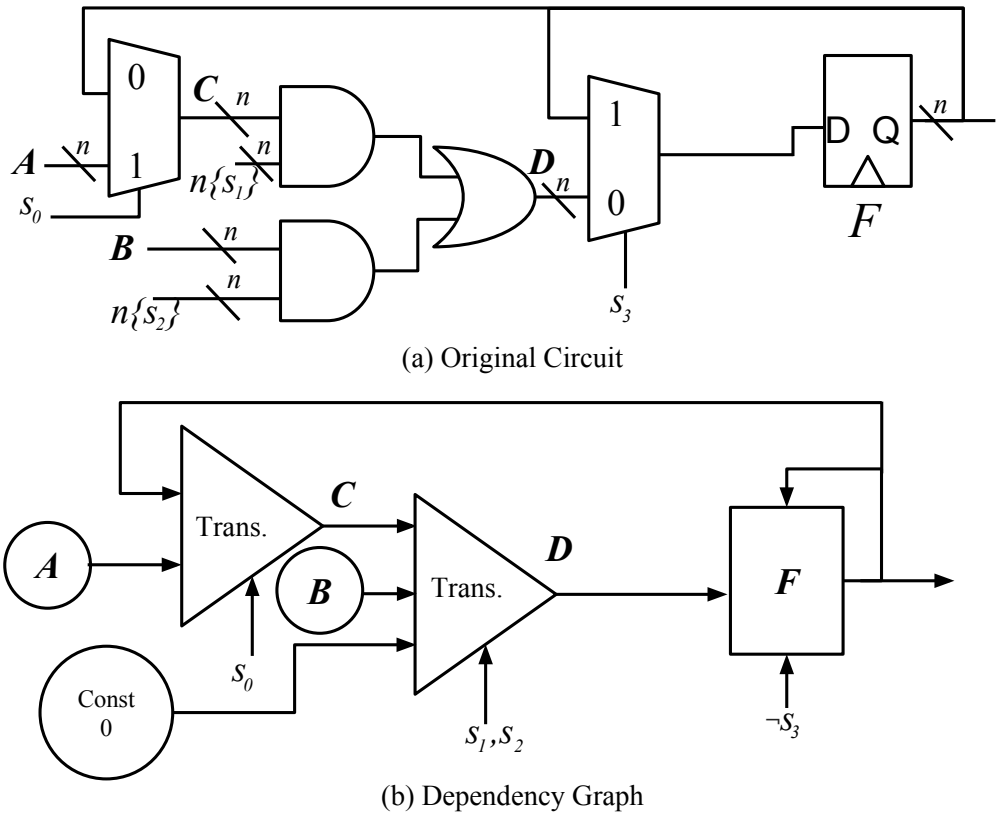
Note that **G** may be already clock-gated using possibly external satisfiability or observability conditions which we cannot know. Those added clock-gating structures may modify the sequential behavior of a version of **G** with no clock-gating, but they are assumed legal when considering external logic. Hence there is no need to verify all clock-gating structures in these designs. Here we assume the clock-gating synthesis from **G** to **R** only adds structures, so in the proposed SEC flow, **G** remains unchanged, while **R** can be simplified to **R'**.

This chapter is organized as follows: in Section 6.1, we explain how to identify clock-gating conditions for the revised circuit. The overall algorithm flow is given in Section 6.2. In Section 6.3, we discuss the issues of depths of recursion of property formulation, as well as the order of processing the candidates. Experimental results are given and discussed in Section 6.4, while Section 6.5 concludes this chapter.

6.1 Identifying Clock-Gating Conditions

To identify candidate FFs that have additional clock-gating signals in **R**, the dependency graphs for **G** and **R** are constructed as D_G and D_R. Because the correspondences of FFs between **G** and **R** are given, each standard or gated FF vertex in D_G has a corresponding vertex in D_R. According to Section 5.1, there are two types of changes in DGs that indicate clock-gating synthesis was done. These are used for comparison between each FF vertex pair to detect new clock-gating conditions on D_R.

Given a candidate gated FF vertex on D_R, the enable signal is denoted as en_{new}. If the corresponding vertex in D_G is a standard FF, en_{old} = constant-1; if gated and controlled by en_G, we need to detect the difference between en_{new} and $en_G = en_{old}$. Since clock-gating synthesis only added extra logic, i.e. MUXes with feedback loops, the signal en_{extra} in **R** can be found by checking all controls covered by the collapsed clock-gating condition $en_{new} = en_{old} \wedge en_{extra}$. Proving en_{extra} redundant, allows it to be removed, creating **R'**.



(a) Original Circuit

(b) Dependency Graph

Figure 6.1: (a) Golden circuit for the one in Figure 4.4. (b) Corresponding dependency graph for the above circuit.

Example: The circuit in Figure 4.4 is revised from the one in Figure 6.1. The control conditions for the gated FF F in D_G and D_R are different, so the enable signal of F in D_R is analyzed. Since the clock-gating synthesis only inserted extra MUXes between F and the MUXes controlled by s_3, s_3 in the two circuits can be mapped. Therefore, $\neg s_3$ is used as en_{old}, while $en_{new} = \neg s_3 \wedge s_4$ (i.e. $en_{extra} = s_4$). Then the flow described in Section 5.5.4 can be followed to verify if the new gating condition en_{new} is legal. If legal, en_{extra} is sequentially redundant and can be replaced with a constant in **R** creating **R'**. When working on the next candidate FF, $D_{R'}$ and **R'** are used.

6.2 Algorithm Flow

Algorithm 6.1 outlines the proposed SEC flow based on DGs and reports if the input circuits are sequentially equivalent. The inputs are golden and revised circuits, **G** and **R**, and *depth*, which is used to limit the number of explored time frames when formulating update and observable conditions.

Algorithm 6.1 SEC Flow based on DGs

Require: **G** and **R**: two circuits with mapped PIs, POs and FFs.
Ensure: **EQ** or **NON-EQ**

1: **R'** = **R**
2: $D_G = dependGraph(\mathbf{G})$
3: $D_{R'} = dependGraph(\mathbf{R'})$
4: **controlPairs** = $findMappedControls(\mathbf{G}, \mathbf{R}, D_G, D_{R'})$
5: **while** $((targetFF, mapFF) = compare(D_G, D_{R'}))$ **do**
6: $\quad en_{new} = getEnable(targetFF)$
7: $\quad en_{old} = analyzeControl(en_{new}, mapFF, \mathbf{controlPairs})$
8: $\quad \mathbf{P} = defineProperty(targetFF, D_{R'}, en_{old}, en_{new}, depth)$
9: $\quad$ **testCir** = $buildCircuit(\mathbf{P}, \mathbf{R'})$
10: $\quad$ **proof** = $multiProve$(**testCir**)
11: $\quad$ **if** $isLegal$(**proof**) **then**
12: $\quad\quad revise(targetFF, en_{old}, D_{R'})$
13: $\quad\quad$ **R'** = $simplify(targetFF, \mathbf{R'}, en_{old}, en_{new})$ $\quad \triangleright\ D_{R'} \equiv dependGraph(\mathbf{R'})$
14: **return** $SEC(\mathbf{G}, \mathbf{R'})$

Function *dependGraph*(...) executes the algorithm of Algorithm 4.1 to construct the dependency graph for each input circuit. Line 4 $findMappedControls$(...) builds a combinational miter between **G** and **R** and performs SAT-sweeping on this to identify related signals. For each enable signal of a gated FFs in **G**, there is a corresponding control signal in **R**. The matchings between these controls are returned as **controlPairs**.

The loop between Line 5 and Line 13 verifies each candidate and revises $D_{R'}$ and **R'** based on the proved candidates one by one. At Line 5, *compare*(...) finds each pair of mapped FF vertices in D_G and $D_{R'}$ and checks if clock-gating has been applied. If the vertices, $(targetFF, mappFF)$, as described in Section 5.1, indicate clock gating has been done, then $targetFF$ is a candidate. To find en_{old} in **R'**, *analyzeControl*(...) explores the components of en_{new} and finds the control which is associated with the enable signal of $mapFF$.

Based on en_{new}, en_{old} and *depth*, $defineProperty$(...) formulates the update and observable conditions for the input of $targetFF$ separately, and builds properties as in Section 5.5.4. Then, based on R', *buildCircuit*(...) constructs a corresponding circuit with multiple outputs (sub-properties) for checking up-to-date condition, along with satisfiability and observability clock-gating conditions. Hence the model-checker $multiProve$(...) at Line 10 verifies all properties simultaneously.

The clock-gating synthesis on $targetFF$ is legal if it reaches the "Legal Clock-Gating Synthesis" terminal in Figure 5.11. If it is legal, then $revise(...)$ modifies the enable signal of $targetFF$ to en_{old}, and $simplify(...)$ replaces other extra controls with constants and simplifies **R'**.

Finally, $SEC(\mathbf{G}, \mathbf{R'})$ is invoked to check if **G** and the completely revised **R'** are sequentially equivalent. If it fails, a counter-example trace is available.

Example: Let the circuit in Figure 6.1 be **G** and the one in Figure 4.4 be **R**, and let $depth = 1$ in the proposed SEC flow. $findMappedControls(...)$ can associate s_0 to s_3 from the two circuits. $compare(...)$ identifies gated FF F as a candidate. Taking $\neg s_3 \wedge s_4$ as en_{new} and $\neg s_3$ as en_{old}, $defineProperty(...)$ formulates the update condition and the observable condition for the input of F (D).

According to D_R in Figure 4.4, since $depth = 1$, $\mathbb{U}(D) = [\mathbb{U}(C) \wedge s_1] \vee [\mathbb{U}(B) \wedge s_2] \vee [s_1 \neq \mathbf{Y}(s_1) \vee s_2 \neq \mathbf{Y}(s_2)]$. Also, $\mathbb{U}(C) = [s_0 \wedge \mathbb{U}(A)] \vee [\neg s_0 \wedge \mathbb{U}(F)] \vee ... = True = \mathbb{U}(B)$ due to $\mathbb{U}(A) = True$ and $\mathbb{U}(F) = True$, because the proposed algorithm returns $True$ for each FF vertex when $depth = 1$. Hence $\mathbb{U}(D) = [s_1] \vee [s_2] \vee [s_1 \neq \mathbf{Y}(s_1) \vee s_2 \neq \mathbf{Y}(s_2)]$, while $\mathbb{O}(D) = \neg s_3$.

Following the flow in Figure 5.11, to check if F is always up-to-date, $\mathbf{G}(\mathbb{U}(D) \Rightarrow s_3)$ is checked, which is invalid if s_3 is independent of s_1 and s_2. Then the observable constraint must be checked, $\mathbf{G}\{[\neg s_3 \wedge \neg s_4] \Rightarrow s_3\}$, which is not satisfied. This property holds only when $[\neg s_3 \wedge \neg s_4] = False$.

If s_3 and s_4 have some correlation (not shown on the circuits), it is possible that $\mathbf{G}\{[\neg s_3 \wedge \neg s_4] \Rightarrow s_3)\}$ holds. Then s_4 in **R'** is sequentially redundant and can be replaced with constant 1. Therefore **G** and **R'** are identical, and the final SEC is trivial.

6.3 Depths and Orders

The parameter *depth* affects not only the range of legal clock-gating conditions, but could affect the overall verification time. For a set of target FFs, if the optimal clock-gating condition (updating the FFs least frequently) must consider up to n time frames, any settings with $depth < n$ might block out some cases where the FFs can be gated.

Example: In Figure 4.4, the update condition for D when $depth = 1$ is formulated recursively as follows: the algorithm goes from D to C, which requires the update condition of F. Due to Line 1 of Algorithm 5.2, $\mathbb{U}(F) = True$, so $\mathbb{U}(C) = [s_0 \wedge \mathbb{U}(A)] \vee [\neg s_0 \wedge \mathbb{U}(F)] \vee [s_0 \neq \mathbf{Y}(s_0)] = [s_0 \wedge True] \vee [\neg s_0 \wedge True] \vee [s_0 \neq \mathbf{Y}(s_0)] = True$ and hence $\mathbb{U}(D) = [s_1] \vee [s_2] \vee [s_1 \neq \mathbf{Y}(s_1) \vee s_2 \neq \mathbf{Y}(s_2)]$. However, if $depth > 1$, the proposed algorithm can go across one time frame through F and formulate $\mathbb{U}(F)$ as $\mathbf{Z}\{[s_3] \wedge [\mathbb{U}(D) \vee \mathbf{Y}([\neg s_3]\mathbf{S}[\mathbb{U}(D) \wedge \neg s_3])]\}$. Because F can potentially remain the same ($\mathbb{U}(F) \neq True$), and $\mathbb{U}(C)$ and $\mathbb{U}(D)$ can be more limited, such that the legal clock-gating condition is strengthened.

When the explored depth is limited, the orders of removing proved redundancies can influence the overall performance.

Example: In Figure 5.8, assume en_1 has been created before clock-gating synthesis was done. The synthesis might add clock-gating conditions to both F_2 and F_3 as $\mathbf{Z}(en_1)$ and $\mathbf{Z}(\mathbf{Z}(en_1))$, respectively. If the proposed SEC flow is used with $depth = 2$, and F_2 is considered first for synthesis with F_3 second, the algorithm can prove and replace the clock-gating condition en_2 on F_2 with en_1. However, when working on F_3, after removing en_2, the algorithm stops exploring and returns $True$ when reaching F_1, so the legal clock-gating condition of F_3 cannot be justified and hence not removed. Therefore, the final SEC can still be challenging.

This issue can be resolved by increasing the depth, but this can degrade the overall performance because it requires exploring DGs across more time frames, making the properties more complicated. Also, the example above can be sorted out if we keep all proved control signals in both the circuit and the DG until all candidates are checked. However, when those proved signals are kept, properties may be formulated that are more complicated than needed, taking more time to prove the remaining candidates.

If the clock-gating synthesis is known to consider only update conditions and all FFs are up-to-date, candidates can be processed in a reverse topological order with a more limited depth.

Example: In Figure 5.8, if the algorithm works with $depth = 2$ on F_3 first, the satisfiability clock-gating condition on F_3 can be justified by the control condition of F_2. Then both clock-gating conditions on F_2 and F_3 can be removed.

If the clock-gating synthesis only relies on observable conditions, candidates should be processed in a topological order. For the cases synthesized with both conditions, there seems to be no best order.

In general, with a limited *depth*, the proposed algorithm cannot guarantee the removal of all extra clock-gating signals. One possibility is to start with a small *depth*, and increase it gradually until the final SEC is easy enough, i.e. repeat the algorithm in Figure 6.1 with increasing *depth* until the SEC problem is proved or disproved.

6.4 Experimental Results

We compare the DG method against the CG method introduced in Chapter 2. The DG method, including transparent logic recognition, DG construction, property formulation and proving, and simplifying **R**, is implemented in ABC. In the following experiments, the liveness properties for observability clock-gating are simplified into weaker safety properties, which will be explained in the next Chapter. The function $multiProve(...)$ used here is the *pdr* command in ABC. We also apply *super_prove* to the sequential miter between **G** and **R'** for the final SEC step in Algorithm 6.1.

6.4.1 Performance for General Clock-Gated Cases

Along with the five cases used in Chapter 2, we add three industrial cases to demonstrate that the DG method is more effective than the CG method. The statistics of those cases and the runtimes for both the methods are listed in Table 6.1.

Table 6.1: Comparisons with the CG method on three OpenCores [33] cases, two synthetic cases and three industrial cases.

Circuit	Clock-Gating Techniques	AND #	FF #	CG method(s)		DG method(s)		
				Simplify	*SEC*	*Simplify*	*SEC*	Reduced #
aes.Round	Observability	125k	645	0.67	2.95	24.344	3.606	128
Md5Core	Satisfiability	95k	40k	0.92	7.92	4.323	9.448	512
CLA_fixed	Observability	3k	97	0.66	1.97	0.228	0.553	32
Synthetic_1	Observability	4k	73	0.56	0.23	0.568	0.980	24
Synthetic_2	Both	877	74	0.65	0.43	0.216	0.430	36
Industry_1	Observability	2k	92	N/A	3.046	0.234	2.339	38
Industry_2	Observability	6k	187	N/A	28.449	1.698	6.257	64
Industry_3	Observability	20k	379	N/A	96.327	3.421	34.431	128

The last column in Table 6.1 indicates the total number of reduced FFs after the simplifying step. For the cases which have been used in Chapter 2, the reduced numbers are the same for both the DG and CG methods. For the three industrial cases, the CG method is incapable of proving and removing any clock-gating conditions. Hence the runtimes under the $CG-SEC$ label refer to the time *super_prove* spends on miters between the original **G** and **R**.

As shown in Table 6.1, for some cases, the proposed DG method needs more time for the steps before the final SEC, due to its more sophisticated pre-processing steps. After the simplification based on DGs, the final SEC problem is easier than the original SEC problem and can be solved efficiently.

Constructing DGs also provides more insights about the input circuits. For example, the second case in Table 6.1, $Md5Core$, is a pipeline circuit with 64 stages, where each stage has four 32-bit data words and one 512-bit word for control saved as FFs. Hence the total number of FFs is about $(32 \times 4 + 512) \times 64 \approx 40k$. In this case, only control words can be gated due to the data dependencies in **G**. Also, the revised circuit (**R**) used in the experiment was gated for only one stage, so the number of re-synthesized FFs (the enable conditions are reduced by the DG method) is 512. However, it is possible that control words in other stages can be gated as well.

6.4.2 Comparisons of Scalability

Here we use the same set of cases from Chapter 2 to demonstrate the scalability of the DG method.

Table 6.2: Comparisons with the CG method on *qmult*, a design from OpenCores [33], with varying bit-widths.

Circuit	AND #	FF #	CG method(s)		DG method(s)		
			Simplify	*SEC*	*Simplify*	*SEC*	Reduced #
qmult_8	487	25	0.58	0.33	0.14	0.39	16
qmult_9	632	28	0.64	0.34	0.16	0.62	18
qmult_10	791	31	0.65	0.34	0.13	0.39	20
qmult_11	964	34	0.65	0.35	0.17	0.63	22
qmult_12	1151	37	0.65	0.35	0.13	0.35	24
qmult_13	1352	40	0.66	0.36	0.19	0.34	26
qmult_14	1567	43	0.65	0.37	0.15	0.36	28
qmult_15	1796	46	0.54	0.36	0.18	0.32	30
qmult_16	2039	49	0.65	0.37	0.15	0.35	32

Table 6.2 shows that the increase of the circuit complexity has no obvious influence on the proposed DG method, because the irrelevant combinational logic is excluded.

For this set of circuits, the proposed DG flow is slightly more efficient than the CG method. It is probably because for each case, the DG flow creates the same properties as the CG method does, and instead of running *super_prove* externally, it performs *pdr* directly inside ABC. Because the properties formulated here are easy to prove, the benefit of running *super_prove* is not clear for these cases, but running *super_prove* externally might result in some overhead.

6.5 Summary

In this chapter, the concepts of DGs were used to resolve SEC problems between circuits before and after clock-gating synthesis. The general SEC flow is similar to the one proposed in Chapter 2, but with following refinements:

1. More control logic and detailed signal dependencies are considered. Hence the DG method can handle more cases.

2. The properties formulated on DGs are more sophisticated and comprehensive.

3. The SEC flow in this chapter assumes all clock-gating conditions on the golden design are legal and only reduces clock-gating conditions on the revised circuit.

Chapter 7

Clock-Gating Synthesis

The proposed DG concepts, including update and observable conditions, also can be used for clock-gating synthesis. In Section 7.1, synthesis techniques for satisfiablity clock-gating are described, while Section 7.2 states the approaches and challenges of synthesizing observability clock-gating conditions. The proposed synthesis flow is shown in Section 7.3. Section 7.4 demonstrates the experimental results. Finally, Section 7.5 concludes this chapter.

7.1 Synthesis with Update Conditions

Given a set of up-to-date target FFs with input in, satisfiability clock-gating aims at building an enable signal $en_{new} = en_{old} \wedge en_{extra}$, which satisfies

$$\mathbf{G}(en_{old} \wedge \neg en_{new} \Rightarrow \neg \mathbb{U}(in)).$$

This can be rewritten as:

$$\begin{aligned} \mathbf{G}(en_{old} \wedge \neg(en_{old} \wedge en_{extra}) &\Rightarrow \neg \mathbb{U}(in)) \\ \mathbf{G}(en_{old} \wedge \neg en_{extra} &\Rightarrow \neg \mathbb{U}(in)) \\ \mathbf{G}(\mathbb{U}(in) &\Rightarrow \neg en_{old} \vee en_{extra}). \end{aligned}$$

Because the FFs are up-to-date at this point in the flow, $\mathbb{U}(in) \Rightarrow en_{old}$.

Since $en_{new} = en_{extra} \wedge en_{old}$, the strongest enabling signal is $en_{extra} \equiv \mathbb{U}(in)$. As in Section 5.6, $\mathbb{U}(in)$ can be constructed recursively as one signal in the original circuit. Note also that it can be constructed prior to the first step of checking the up-to-date condition. Hence satisfiability clock-gating can be done easily.

As with verification, synthesis with a larger depth can result in better (stricter) clock-gating conditions. However, for synthesis with a limited depth, it is better to process targets in a topological order, so that constructed update conditions for other targets may be reused.

Example: in Figure 5.8, assume only en_1 has been created before synthesis, while F_2 and F_3 are standard FFs and considered as targets. Following a topological order, satisfiability

synthesis should be applied to F_2 before F_3. Since F_2 is always up-to-date, it can be gated by $\mathbb{U}(F_1) = \mathbf{Z}(en_1) = en_2$. Hence en_2 is synthesized in the original circuit. Then F_3 can be gated by $\mathbb{U}(F_2) = \mathbf{Z}\{(en_2) \wedge [\mathbb{U}(F_1) \vee ...]\}$. Because the satisfiability clock-gating synthesis guarantees the strongest clock-gating condition ($en_2 \equiv \mathbb{U}(F_1)$) and F_2 must be up-to-date, $\mathbb{U}(F_2)$ can be simplified as $\mathbf{Z}(en_2)$. In other words, F_3 can be analyzed and gated without exploring the fanin cone of F_2, while en_2 can be reused to create $en_3 = \mathbf{Z}(en_2)$.

7.2 Synthesis with Observable Condition

For a set of target FFs (out) with input in and en_{old}, observability clock-gating synthesis requires the construction of an enable signal en_{new} that satisfies

$$\mathbf{G}((en_{old} \wedge \neg en_{new}) \Rightarrow \neg \mathbf{O}(in)),$$

where $\mathbf{O}(in) = [en_{old}] \wedge \{\mathbf{X}(\mathbf{O}(out)) \vee \mathbf{X}[(\neg en_{new})\mathbf{U}(\mathbf{O}(out))]\}$. However, there is no naive way to construct such en_{new} with a liveness concept.

In the proposed flow, observable clock-gating synthesis is only applied to standard FFs, i.e. without clock-gating (en_{old} = constant-1). Also, to simplify the target property, $\mathbf{O}(in)$ is over-approximated with $\mathbf{X}(\mathbf{O}(out))$. Therefore the synthesis flow aims at constructing a signal en_{new} to satisfy

$$\mathbf{G}(\mathbf{X}(\mathbf{O}(out)) \Rightarrow en_{new}).$$

The strongest synthesis is therefore to take $en_{new} = \mathbf{X}(\mathbf{O}(out))$. Notice that a signal en_{new} satisfied with the simplified property satisfies the original property for sure, because $\mathbf{G}(\mathbf{X}(\mathbf{O}(out)) \Rightarrow en_{new})$ implies $(\neg en_{new})\mathbf{U}(\mathbf{O}(out))$ must be constant-0.

It is possible that the desired observable clock-gating condition cannot be synthesized because it is related to events in the future, which might not be predictable in the current time frame.

7.2.1 Synthesis with "Next" Property

To synthesize a signal en_{new} as $\mathbf{X}(\mathbb{O}(out))$, this needs to be constructed as a signal O_{out} on the original circuit. Then the target signal $\mathbf{X}(O_{out})$, depends on the fanin cone of O_{out} back one time frame.

If O_{out} is fully supported by FFs, $\mathbf{X}(O_{out})$ can be constructed by adding a one-time frame fanin cone of O_{out} to the previous time frame, i.e. skipping all FFs between the two time frames. There is an example shown in Figure 7.1(a).

In Figure 7.1, O_{out} is evaluated by a combinational block **A**, which is fully supported by FFs. Those FFs are the next state function of certain outputs from another block **B**. The value of O_{out} in the next time frame is determined by feeding the current outputs of **B** to **A**. Hence $\mathbf{X}(O_{out})$ can be built by duplicating **A** and adding it to **B**, which supports **A** across one time frame.

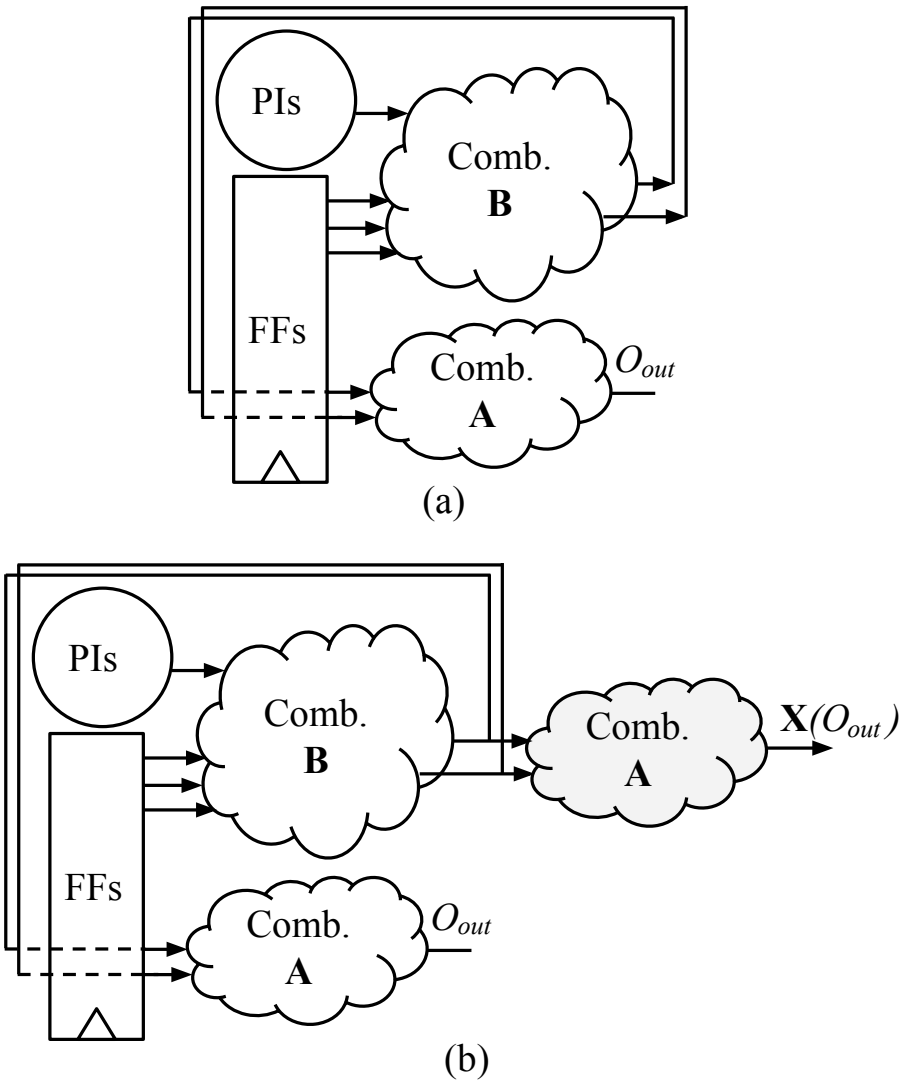


Figure 7.1: Example for synthesizing $en_{new} = \mathbf{X}(\mathbb{O}(out))$. In (a), O_{out} has been built as a new signal in the circuit. Then in (b), the combinational circuit **A** supporting O_{out} is duplicated and added to the other combinational block **B** which supports **A** across one time frame.

If the fanin cone of O_{out} (**A**) is supported by any primary inputs, it cannot be built because primary inputs are free and impossible to predict. Hence clock-gating synthesis fails.

Moreover, when constructing $\mathbf{X}(\mathbb{O}(out))$ recursively with a depth limit, it is possible to reach gated FFs, which can introduce until **U** properties to the formulation.

7.2.2 Synthesis with "Until" Property

To synthesize the observable condition, $\mathbb{O}(in)$, for the input of a set of gated FFs, because it is not the target FF, $en_{old} = en_{new} = en$. Then

$$\mathbb{O}(in) = en \wedge \mathbf{X}(\mathbb{O}(out)) \vee \mathbf{X}[(\neg en)\mathbf{U}(\mathbb{O}(out))]),$$

where $\mathbb{O}(out)$ is the observable condition of the output.

To build $\mathbb{O}(in)$, $\mathbb{O}(out)$ is constructed in the original circuit first and then the idea in the previous section is applied to create $\mathbf{X}(\mathbb{O}(out))$. If $\mathbb{O}(out) = True$, or if $\mathbf{X}(\mathbb{O}(out))$ cannot be built because $\mathbb{O}(in)$ depends on some primary inputs, $\mathbf{X}(\mathbb{O}(out))$ is set to $True$ directly. Therefore, $\mathbb{O}(in)$ can be simplified to en.

After $\mathbf{X}\mathbb{O}(out) \neq True$ has been constructed, the relationship between en and $\mathbf{X}\mathbb{O}(out) \neq True$ needs to be analyzed to resolve the "until" $\mathbf{U}$ part of the target property. If the property

$$\mathbf{G}(\mathbf{X}(\mathbb{O}(out)) \Rightarrow (en)) \tag{7.1}$$

holds, $[(\neg en)\mathbf{U}(\mathbb{O}(out))]$ must be $False$. That is, there is no out-of-date data that can be observable before the FFs updates to newer data. Then $\mathbb{O}(in)$ is built as $\mathbb{O}(in) = [en \wedge \mathbf{X}(\mathbb{O}(out))] = \mathbf{X}(\mathbb{O}(out))$.

If the property in Formula 7.1 fails, $[(\neg en)\mathbf{U}(\mathbb{O}(out))]$ can be $True$ during execution. This sub-formula can be decomposed into the union of an infinite number of formulas as follows:

$$\begin{aligned}
&[\neg en \wedge \mathbf{X}(\mathbb{O}(out))] \\
\vee&[\neg en \wedge \neg\mathbf{X}(\mathbb{O}(out)) \wedge \neg\mathbf{X}(en) \wedge \mathbf{X}^2(\mathbb{O}(out))] \\
\vee&[\neg en \wedge (\neg\mathbf{X}(\mathbb{O}(out) \wedge ... \wedge \neg\mathbf{X}^2(en) \wedge \mathbf{X}^3(\mathbb{O}(out))] \\
\vee&...,
\end{aligned}$$

which describes all traces where $[(\neg en)\mathbf{U}(\mathbb{O}(out))]$ becomes $True$. It is impractical to create all sub-formulas so $[(\neg en)\mathbf{U}(\mathbb{O}(out))]$ is approximated as $True$ and thus $\mathbb{O}(in) = en$. That is, the observable condition is formulated with the enable signal of the gated FFs only, while the observable condition of the outputs are excluded. Hence, the recursion of formulating $\mathbb{O}(out)$ is not performed.

Example: Figure 7.2 shows a circuit and its corresponding DG. Only FFs F_3 have been gated by en, which is computed by combinational block $\mathbf{A}$ based on FFs F_5. All signals in F_5 are determined by combinational block $\mathbf{B}$ based on FFs F_4, while F_4 is supported by block $\mathbf{C}$. In other words, $en \equiv \mathbf{A}(F_5)$, $\mathbf{X}(F_5) \equiv \mathbf{B}(F_4)$, and $\mathbf{X}(F_4) \equiv \mathbf{C}(PIs)$

F_1 and F_2 are candidates for observability clock-gating. If the process starts with F_1, the observable condition of its inputs is :

$$\begin{aligned}
\mathbb{O}(in) = \mathbf{X}(\mathbb{O}(F_1)) =& \mathbf{X}(\mathbf{X}(\mathbb{O}(F_2))) \\
&= \mathbf{X}(\mathbf{X}(en) \wedge [\mathbb{O}(F_3) \vee ...]).
\end{aligned}$$

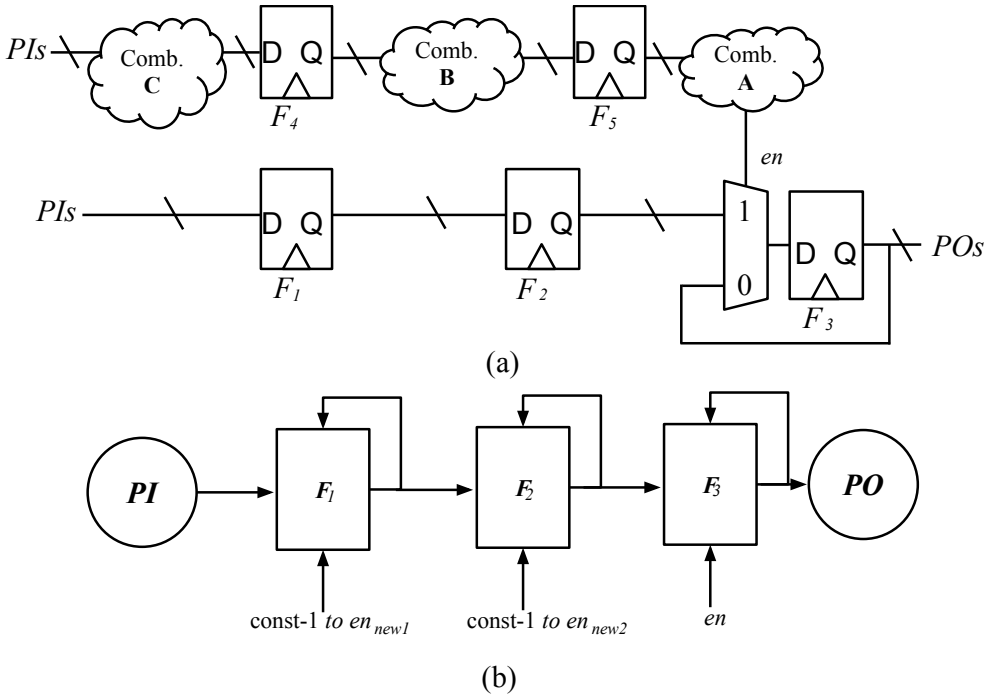


Figure 7.2: An example for observability clock-gating synthesis. (a) A sequential circuit with two sets of target FFs, F_1 and F_2. (b) The corresponding DG.

where $\mathbb{O}(F_3) = True$. Hence the target enable signal for F_1 is $en_{new1} = \mathbf{X}(\mathbf{X}(en))$. According to Section 7.2.1, $\mathbf{X}(en)$ can be built as $\mathbf{A}(\mathbf{B}(F_4))$, and $\mathbf{X}(\mathbf{X}(en))$ is identical to $\mathbf{A}(\mathbf{B}(\mathbf{C}(PIs)))$. Thus, the three combinational blocks are concatenated together to build en_{new1}. Similarly, the observability clock-gating synthesis for F_2 and proposes the enable signal en_{new2} as $\mathbf{X}(en)$, which can be built as $\mathbf{A}(\mathbf{B}(F_4))$. Note that en_{new1} and en_{new2} can be simplified by combinational synthesis.

However, if F_2 is processed first, en_{new2} is constructed and simplified as $simp[\mathbf{A}(\mathbf{B}(F_4))]$ with the same steps. Then when working on F_1, the observable condition is

$$\mathbb{O}(in) = \mathbf{X}(\mathbb{O}(F_1)) = \mathbf{X}(en_{new2}) \wedge [\mathbf{X}(\mathbb{O}(F_2)) \vee ...]),$$

where $\mathbf{X}(\mathbb{O}(F_2)) = \mathbf{X}(en)$, which has been analyzed and constructed for F_2. The property in Formula 7.1 must hold because $en_{new2} = \mathbf{X}(en)$. Hence $\mathbb{O}(in)$ is simplified as $\mathbf{X}(\mathbf{X}(\mathbb{O}(F_2)))$, where $\mathbf{X}(\mathbb{O}(F_2)) \equiv \mathbf{X}(en) \equiv \mathbf{A}(\mathbf{B}(F_4)) = en_{new2}$. The simplified $simp[\mathbf{A}(\mathbf{B}(F_4))]$ can be copied and built $en_{new1} = simp[\mathbf{A}(\mathbf{B}(\mathbf{C}(PIs)))]$. Therefore, when formulating the observability condition for a set of gated FFs, where observability clock-gating synthesis has been applied, the property in Formula 7.1 must hold, and the constructed enable condition can be used directly. That is, there is no need to explore the fanout cone of the gated FFs again.

Therefore, when performing observability clock-gating synthesis for multiple targets, it is more efficient to follow a reverse topological order.

7.3 Synthesis Flow

Given a sequential circuit and a set of target FFs, which has been gated by en_{old} (en_{old} = constant-1 refers to standard FFs), the flow in Figure 7.3 is proposed for clock-gating synthesis, in which a new enable condition en_{extra} is synthesized and the target FFs are gated by $en_{new} = en_{extra} \wedge en_{old}$.

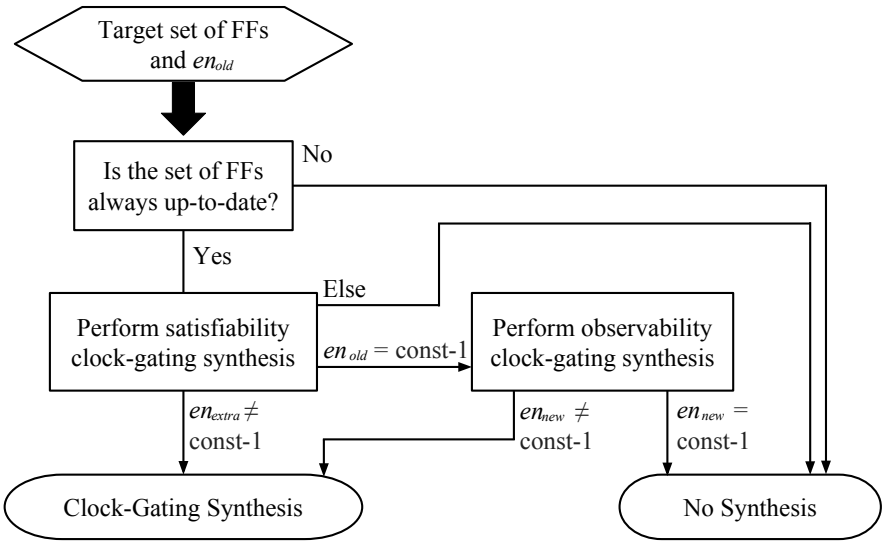


Figure 7.3: Synthesis flow for a target set of FFs.

In Figure 7.3, the first step checks if the target FFs are up-to-date under en_{old}. This can be done by formulating the update condition for the FFs input and proving Formula 5.2 with a hardware model checker. If the property holds, the target FFs can be clock-gated by using update conditions (satisfiability clock-gating synthesis) to create a legal enabling signal en_{extra} without considering the observable conditions. If $en_{extra} \neq$ constant-1, then clock-gating synthesis is achieved.

If (1) the target FFs can be out-of-date, or (2) $en_{old} \neq$ constant-1 and there is no $en_{extra} \neq$ constant-1 for satisfiability clock-gating, the proposed flow terminates (indicated by the arrow labeled with *Else* from the satisfiability clock-gating block in Figure 7.3.) That is, only when en_{old} = constant-1 and no en_{extra} can be added by considering update conditions, the observable conditions are used to derive $en_{extra} = en_{new} \neq$ constant-1 as observability clock-gating synthesis. The proposed en_{new} must satisfy the property in Formula 5.1. If no such en_{new} exists, the flow stops without modifying the circuit.

Finally, if an additional enable condition en_{extra} is proposed, on the corresponding DG, the enable signal of the target FFs is modified to $en_{old} \wedge en_{extra}$. On the original circuit, it can be represented as inserting a set of MUXes controlled by en_{extra} between the target FFs and their corresponding input signals, where there are feedback loops from the FFs to

MUX inputs. In practice, as in Figure 1.2, it can be achieved by ANDing the clock with $en_{old} \wedge en_{extra}$.

Based on the flow for each target FF shown in Figure 7.3, for input circuit, **Cir**, and input parameter *depth*, a proposed synthesis algorithm is outlined in Figure 7.1. Functions $synthesisUpdate(...)$ and $synthesisObservable(...)$ are used to (1) formulate update or observable conditions on the DG based on the algorithms in Algorithm 5.1 and 5.2, and (2) construct corresponding signals on **revCir** as described in the previous sections. Note that the property formulation part can be terminated earlier when reaching already analyzed FFs, and enable signals constructed by clock-gating synthesis can be used to build other clock-gating conditions.

Algorithm 7.1 Clock-Gating Synthesis

Require: **Cir**: a sequential circuit; *depth*: parameter for the functions $synthesisUpdate(...)$ and $synthesisObservable(...)$.
Ensure: **revCir**: the circuit after clock-gating synthesis.

```
1:  DG = dependGraph(Cir)
2:  revCir = Cir
3:  candidates = topologicalSort(FFVertices(DG))
4:  for all target in candidates do
5:      U_in = synthesisUpdate(input(target), depth, revCir, DG)
6:      if isUpToDate(target, U_in) then
7:          updateEnable(revCir, DG, target, U_in)
8:  candidates2 = reverseTopologicalSort(FreeFFs(DG))
9:  for all target in candidates2 do
10:     O_in = synthesisObservable(input(target), depth, revCir, DG)
11:     updateEnable(revCir, DG, target, O_in)
12: return revCir
```

After constructing the DG for **Cir**, all standard and gated FF vertices are sorted in a topological order. From Line 4 to 7, each FF vertex is examined and satisfiability clock-gating is performed one by one. At Line 5, the update condition for the vertex input is formulated on the DG, and then the corresponding signal U_{in} is constructed on the circuit. Based on U_{in} and the old enable signal, $isUpToDate(...)$ uses a hardware model checker to verify if the target is always up-to-date. If so, the clock-gating condition of $target$ is revised with U_{in}, both on the DG and the circuit.

Then only standard FFs (free FFs) are considered for observability clock-gating. As discussed, observability clock-gating synthesis should be done in a reverse topological order. For each FF vertex, the observable condition for its input is formulated, and the algorithm also builds the corresponding signal O_{in} on the circuit. Note that when recursively constructing O_{in}, to resolve "until" **U** formulas, $synthesisObservable(...)$ may formulate some properties as Formula 7.1 and prove them by a hardware model checker. If the observable condition

cannot be constructed as a signal, O_{in} is set to $True$, while $updateEnable(...)$ returns without doing anything. If O_{in} can be constructed, both the DG and **revCir** are revised.

Finally, **revCir** is returned as a circuit after clock-gating synthesis, which is sequentially equivalent to **Cir**.

7.4 Experimental Results

The proposed synthesis flow was implemented in ABC and was applied to the golden circuits in the benchmark pairs we used for verification. We do not include the industrial cases here because the number of gated FFs are unknown. In Table 7.1, the fourth column indicates the number of gated FFs proposed by a reference manual analysis (which models what a designer might do), while the fifth column shows the number of gated FFs proposed by the DG method.

Table 7.1: Experimental results of the proposed clock-gating synthesis flow.

Target Circuit	AND #	FF #	Reference Gated FF #	Synthesis with DG	
				Gated #	Runtime(s)
aes.Round	125k	645	128	128	1.479
Md5Core	95k	40k	512	32256	23.313
CLA_fixed	3k	97	32	32	0.192
Synthetic_1	4k	73	24	24	0.241
Synthetic_2	877	74	36	50	0.201

As shown in Table 7.1, the proposed method can synthesize clock-gating conditions efficiently. For some cases, like the pipeline circuit $Md5Core$, the DG method can propose more clock-gating conditions than the reference synthesis does. Hence the proposed method provides improved possibilities for low-power circuit designs. The additional advantage is that this can be done automatically by our proposed synthesis algorithm.

7.5 Summary

In this chapter, we use DGs and legal clock-gating conditions on DGs to perform clock-gating synthesis for sequential circuits. Experimental results show that the proposed method is effective and efficient.

In the modern VLSI design flow, once a legal clock-gating condition is proposed, it is a must to analyze the overall improvements in power consumption to determine whether this clock-gating condition should be inserted to the circuit. Proposing clock-gating conditions and examining real power consumption should be done iteratively to achieve the best performance. The automatic synthesis framework proposed in this chapter makes the whole flow more independent of manual efforts.

Chapter 8

Circuit Recognition with Convolutional Neural Networks

In addition to transparent logic, recognizing arithmetic operators, including adders, multipliers, dividers, etc., has been a focus of reverse engineering. For hardware security inspection, reverse engineering, which extracts high-level components from bit-level designs, is desired.

Reverse engineering methodologies [28, 45, 44] typically consist of two parts: (1) decomposing a gate-level circuit into suspected blocks and (2) mapping sub-circuits of a block to high-level components from a library. Thus, reverse engineering is mainly about detecting and locating functional components from gate-level circuits. This process can then isolate any unknown components for further inspection. Generally, existing frameworks for this process start with finding a set of candidate words and operators using structural and functional approaches, and then applying formal methods to justify each candidate. Because proving with formal methods is time-consuming, an efficient method of finding small candidate subsets is desired. Once found, they can be used to further decompose the circuit into smaller parts.

To check if a target component in a library can be a candidate for a match, certain features or properties are derived to detect the target component. For example, based on behavioral pattern mining on simulation traces, Li et. al. [28] constructed pattern graphs for library components and target gate-level netlists. Also, Soeken et. al. [44] used simulation graphs of a functional block to detect arithmetic operators. The characteristics and varieties of selected features dramatically influence the accuracy of candidate search, and thus the performance of the overall algorithm.

In this chapter, convolutional neural networks are proposed to find suitable features for identifying candidates effectively. This is possibly the first time CNNs have been applied to circuit recognition. The relative success of the proposed methodologies offers the possibility of application to other computer-aided design problems.

The remainder of this chapter is organized as follows: relevant background knowledge about modern machine learning techniques is introduced in Section 8.1. In Section 8.2, essential requirements for representing circuits for CNN processing are discussed. Section 8.3

describes the proposed convolution operation for circuits, and a proposed dynamic pooling is given in Section 8.4. The framework for recognizing circuits is shown in Section 8.5. Section 8.6 compares the performance of different methods on operator classification and detection. Section 8.7 summarizes the results and some future directions.

8.1 Preliminary: Modern Machine Learning Algorithms

Machine learning is a type of artificial intelligence which enables computers to identify special patterns from input data and construct models without being explicitly programmed. The ability of adapting to new data is the main focus when developing machine learning algorithms. Such algorithms are often categorized as *supervised* or *unsupervised*. In this chapter, only supervised learning is considered.

Supervised learning relies on a known dataset (training set), which includes pairs of input observations and expected outputs. A supervised learning algorithm uses the training set to build a model that can predict the outputs for other sets of observations (testing set). Many such algorithms have been developed for classifying collections of observations into specific classes, e.g. decision tree learning [36], support vector machines (SVM) [11] and artificial neural networks [13].

8.1.1 Support Vector Machines

Support vector machines (SVM) [11] are a set of supervised learning methods used for classification and regression. Roughly speaking, given a set of labeled training data, a support vector machine constructs a hyperplane or set of hyperplanes in a high-dimensional space as a model to classify new data points into different regions (classes).

Consider a training data set for SVM: each data point is represented as a p-dimensional vector, and the training procedure of SVM aims at finding a $(p-1)$-dimensional hyperplane to separate those points based on their labels. The p-dimensional vector for each point refers the collection of p *features*. For example, for a digital image, the value that represents the brightness of one color for each pixel is one feature.

A critical drawback of SVM methods is that the selection of features significantly influences the accuracy rate of SVM. Hence manually selecting or extracting features might be required. Consider the problem of handwritten digit recognition as an example: when applying SVM, using orientation histograms as features can achieve a much better accuracy rate than using raw pixels [30]. In contrast, due to the ability of extracting proper features automatically, convolutional neural networks can achieve better performance with raw pixels [25].

8.1.2 CNN for Images

A convolutional neural network (CNN) [26] is a type of artificial neural network inspired by the mechanism of the animal visual cortex [19]. It can make use of the internal structure of data through convolution layers containing computation units, each of which processes only a small region of input data. CNNs have been used widely in computer vision and image processing applications. Recent research also apply CNNs to text categorization [22] and graph classification [32].

A major advantage of CNNs is their lack of dependence on manual efforts in designing and selecting features. Most machine learning techniques rely on manual feature selection to achieve high performance, while CNNs require relatively little pre-processing of data. That is, a CNN is responsible for learning essential filters while these need to be manually defined in other algorithms.

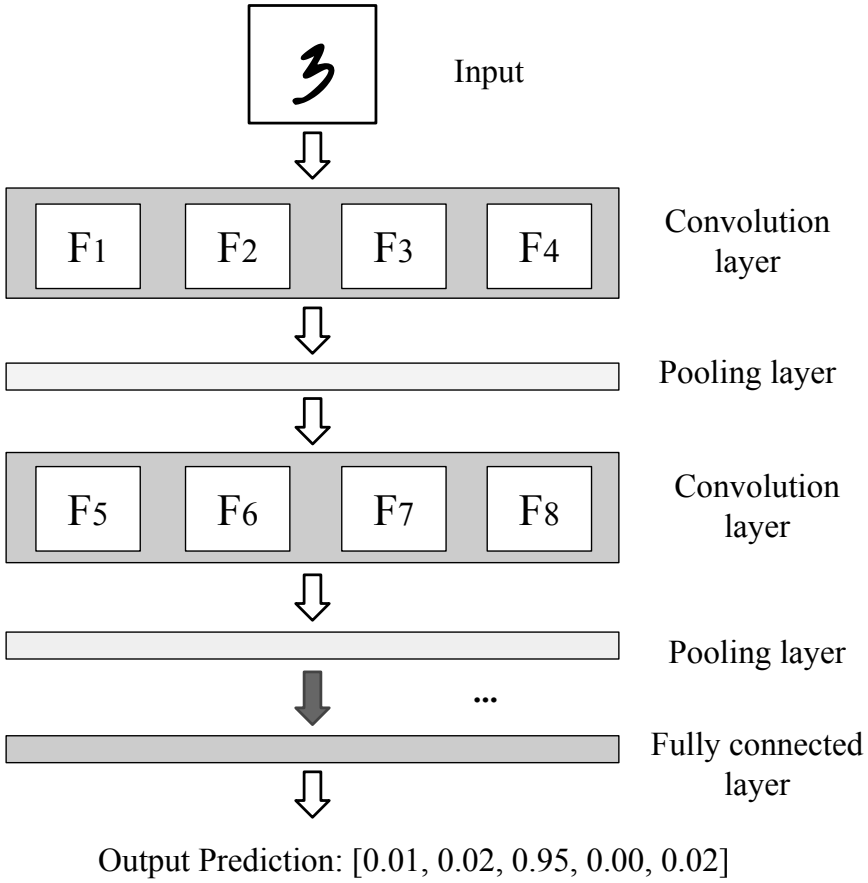


Figure 8.1: A typical convolution network for image processing. There are five expected outputs (classes), and the input object is most likely to belong to the third class in this case.

A typical CNN is a feed-forward network mainly composed of convolution layers, pooling

layers and fully connected layers. There are some types of layers, like activation or dropout layers, that provide non-linear properties or avoid overfitting. Moreover, convolution and pooling layers are critical parts of CNNs. Figure 8.1 illustrates a CNN with convolution layers interleaved with pooling layers, and a final fully connected layer which performs predictions based on the features generated by the preceeding layers. The network in Figure 8.1 has five expected outputs. For the input object, the model predicts the probability (bottom line) of belonging to each class. The final classification is that the input belongs to the third class because it has the highest probability (.95).

The various types of CNN layers used in this chapter are explained as follows:

1. A *convolution layer* consists of a set of trainable filters. Each filter is a small computation unit extracting one local feature across the whole input data. During the forward pass, dot products are performed between the entries of each filter and each input image centered at any position to produce one feature map. Consider Figure 8.2 as an example: Filter 1 detects vertical edges by computing horizontal gradients, while Filter 2 reveals horizontal edges by calculating vertical gradients. These filters are applied in sliding 2×2 windows to the array of input data. Those local features are combined to derive higher order features in the succeeding layers. An activation layer might be appended to a convolution layer to result in non-linear combinations of the weighted inputs.

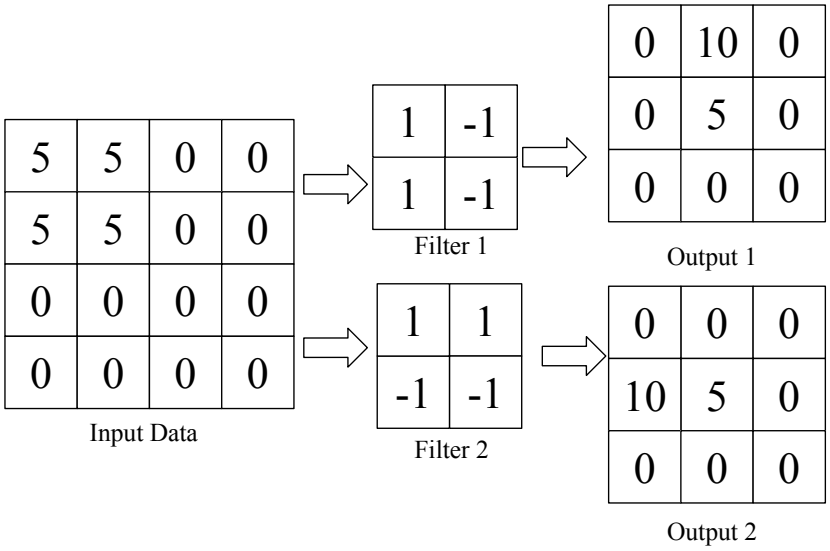


Figure 8.2: A convolution layer with two 2×2 filters for image processing.

2. To reduce the amount of parameters and computation in CNNs, usually a *pooling layer* is inserted after a convolution layer. In a pooling layer, an input feature map is partitioned into small regions and shrunk by a certain operator. Figure 8.3 demonstrates

max-pooling, in which the MAX operation is performed in each small region. Pooling layers also lessen the over-fitting issue of CNNs by ignoring small disturbances. Notice that the pooling operator is pre-defined and cannot be modified during training.

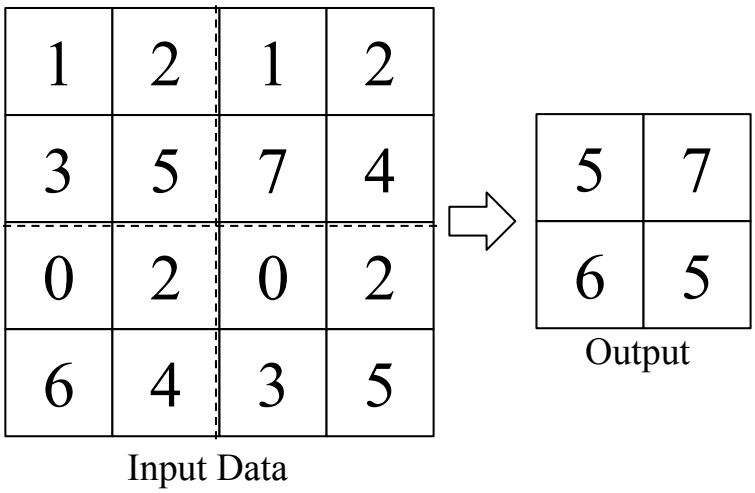


Figure 8.3: Max-pooling layer for image processing.

3. A *fully connected layer* takes all output features computed by the previous layer to determine each of its output values. For each output, it performs the dot products between all input features and a set of trainable parameters. In most CNNs, there is more than one fully connected layer, where the number of outputs of the last layer is the number of expected classes.

4. An activation layer processes values from its previous layer with an *activation function* f, and outputs to the next layer. Typically the activation functions used in neural networks are non-linear. There are two types of activation functions used in this chapter: *rectified linear unit (ReLU)* and *Softmax* function. An ReLU activation layer is applied after each convolution layer, where each input value x is transferred to $f(x) = max(0, x)$ individually. A Softmax activation layer usually works as the final layer of a CNN to present the classification results, because the output of the softmax function can demonstrate a categorical distribution, i.e. showing the probability distribution over n different possible classes [6]. The softmax function takes all values from the previous layer to compute an n-dimensional vector of real values in the range [0, 1] that add up to 1.

5. A dropout layer is added into a CNN to avoid overfitting. Each dropout layer comes with a parameter, *dropout fraction* p ($0 \leq p \leq 1$), which means each value from the previous later is either "dropped out" with probability p or kept with probability $1 - p$ at each training stage. Hence given the same training set, the actual data processed after a dropout layer can be different.

8.2 CNN-Adaptive Circuit Representation

The most critical issue in using CNNs for circuits is how to convert a circuit into a readable format for CNNs, which expect matrices filled with real numbers. For each CNN, the sizes of all input matrices representing all circuits must be the same for a given problem. Images with different sizes can be easily re-scaled to fit the input requirement, but there is no obviously one best way to re-size circuits.

A naive approach is to use an adjacency matrix to describe connections among gates. For instance, an And-Inverter Graph (AIG) with N signals (primary inputs or AND gates) can be described as an $N \times N$ matrix A. The element A_{ij} is 1 when there exists an edge from signal i to signal j, -1 when this edge comes with negation (in the AIG), and 0 when there is no edge.

However, this format has a scalability issue. All circuits for training or testing are expected to be in the same format, which is the adjacency matrix of the largest circuit. Then other smaller circuits would require a proper scaling or zero-padding to fit into this format.

Another approach is to express a circuit in a format similar to AIGER [3]: an AIG with N signals is described as an $N \times 3$ matrix. Each signal in the AIG is associated with a variable indexed by a natural number. Each row of the $N \times 3$ matrix represents one AND gate with its input and output variable indexes. To represent negated edges on the AIG, each variable index is multiplied by two for its positive phase, while its negation is expressed by adding one to the multiplied number. For example, a variable indexed with 5 is expressed as 10 and 11 for its positive and negative phases, respectively. Therefore, for each AND gate, it is represented by a row with three elements: (1) positive phase of the gate output, (2) one input with its phase (negation or not) on the AIG, and (3) the other input with the phase.

However, this format contradicts fundamental principles of CNNs because the range of natural numbers (indexes) used in the input varies among different circuits. Also, normalization or standardization of entries is prohibited, because it can destroy a circuit's properties completely.

Moreover, the convolution operation on the above format cannot catch properties precisely, because a linear combination of literals does not act as a guide for finding features. For example, the inner product between a filter (0, 1, -1) and (8, 3, 5) is identical to the product between the filter and (110, 103, 105), but their interpretations are distinct in circuits. Hence expressing circuits with variable indexes is not suitable for CNNs.

With the above examples in mind, some desirable properties proposed for describing circuits for CNN processing are:

1. The input data has a fixed size, regardless of the original circuit size.
2. The input data describes both functional and structural information.
3. Two circuits with distinct properties should have different descriptions.
4. Numerical entries are within a fixed range, and no normalization or standardization is needed.

To address the above properties, a convolution preprocessing operation for circuits and a corresponding pooling operation to represent circuits for CNNs are proposed.

8.3 Convolution on Circuits

For circuit synthesis, *technology mapping* is a process of implementing a target circuit by choosing gates from a technology library. It has some features of a convolution layer used for images: both extract local properties from the original subjects and both maintain the globally relative positions of the extracted components, e.g. topological orders.

8.3.1 Technology Mapping

As a running example, Figure 8.4 includes a subject circuit and a library of gates. A *cover* (labeled with dashed rounded rectangles) is a mapping from library cells to the subject circuit, where every node is contained by at least one cell. The cost of a cover is determined by a set of costs of objective functions. The problem is to find a cover with minimum cost for the subject circuit. For our example, the cost of a cover might be the total number of cells used in the cover.

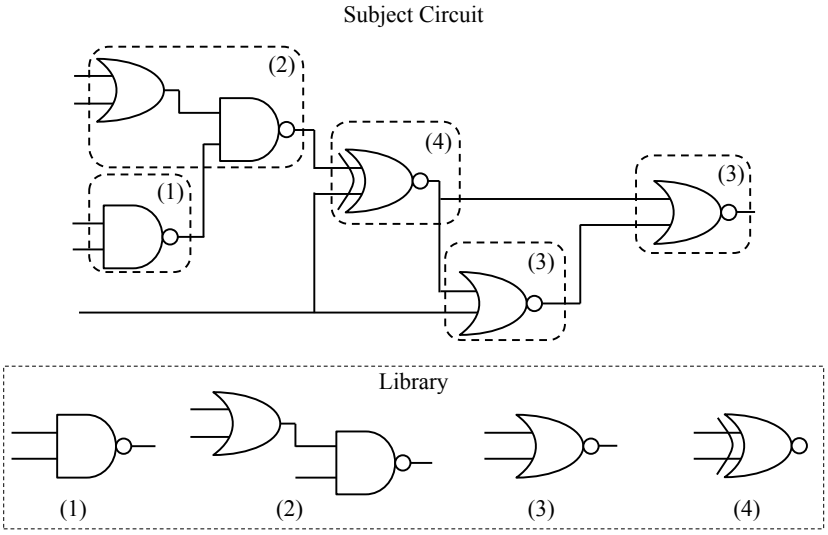


Figure 8.4: A running example of circuit convolution.

Many technology mapping methods have been developed to efficiently minimize costs of mapped circuits [31, 38]. As shown in Figure 8.4, the subject circuit is converted to another directed acyclic graph (DAG), where vertices are indexed with cells in the library.

The concept of technology mapping can be extended by allowing one cell in the library to represent more than one type of sub-circuits. Thus, two different sub-circuits can be

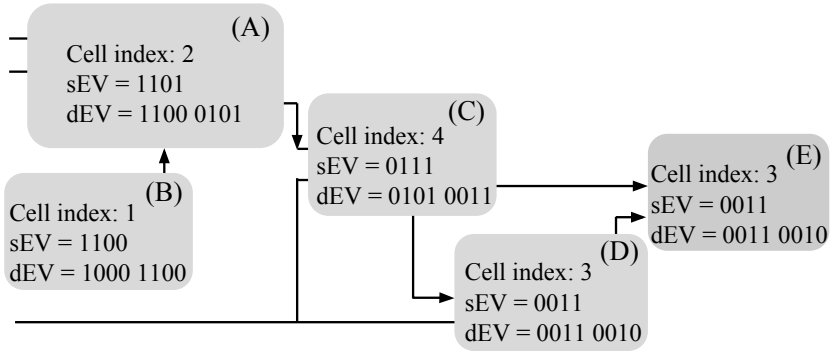


Figure 8.5: The subject circuit after technology mapping. The cell index for each node refers to the corresponding library cell in Figure 8.4.

identified as the same cell if they share some properties. For example, XOR and XNOR gates can be classified as the same cell, because both are heavily used in certain operations. This extension can decrease the size of the library and produce smaller data files that reduce the computation efforts of CNNs.

To process circuits written in a high-level language, like word-level Verilog, a cell library can be a set of arithmetic and other operators for bit-vectors. Then an input circuit is a DAG with indexed vertices. This representation of high-level information can be used for recognizing more high-level properties, such as "this circuit contains a linear sum or a filter of some kind".

The library used in this paper is the set of all npn isomorphism classes of 4-input lookup-tables (4-LUTs). A 4-LUT can implement any Boolean function with 4 inputs and one output. The library size (total number of npn isomorphism classes) is 222. For experiments in Section 2.4, all circuits are mapped into 4-LUTs, and then converted into DAGs, where vertices are indexed by their npn isomorphism classes of 4-LUTs. This brings in an element of local functionality rather that relying on the structure of the underlying circuit.

8.3.2 Existence Vector

Technology mapping converts a circuit into a DAG with indexed vertices, which illustrate local functionalities. However, CNNs require vector representation of data that preserve neighborhood information (directed edges in this case) as input.

Existence vectors (EVs) are used to represent not only functionality but connectivity for each vertex in a DAG. For each vertex, the corresponding EV indicates the existence of each cell type in its neighborhood using a binary encoding: 1 means presence a cell type while 0 stands for absence. Each element of an EV refers to a cell type in the library. For example, if the 135th entry has a 1 in it, it means that there is a cell of type 135 in the immediate neighborhood of the vertex associated with the EV.

For a DAG synthesized with a library of L cells, a vertex can be associated with a *double existence vector* (dEV) with length $2L$. The first L entries of the dEV refer to the neighborhood including the corresponding vertex and its distance-1 fanins, while the other L entries stand for the neighborhood of the vertex and its distance-1 fanouts.

A node can be expressed as a *single existence vector* (sEV) with length L. To express a vertex as an sEV, the neighborhood of the target vertex includes its distance-1 fanins and fanouts and itself. For the same vertex, the sEV is identical to the bitwise OR between the first and the second half of its dEV.

Example: Figure 8.5 has 4 cells in the library, so each sEV contains 4 entries and each dEV has 8. Vertex C (indexed 4) is driven by vertex A (indexed 2) and fanouts to vertices D (indexed 3) and E (indexed 3). The first half of its dEV is 0101 because cell types 2 and 4 are present, while the other half is 0011 based on the indexes of C, D and E. The sEV for vertex C is 0111 because the neighborhood includes cell types 2, 3 and 4.

Generally dEVs can express the connectivity of vertices more precisely, but using sEVs can reduce the network size and lessen the training effort. Comparisons between using dEV and sEV are given in Section 2.4.

8.4 Pooling for Circuits

After this circuit-based pre-processing, all vertices in a mapped circuit are represented by a set of EVs. The total number of EVs are different for different circuits. There is one EV for each cell that is in the cover of library cells created by technology mapping. Each sEV has a length equal to the number of cells in the library. Cell types are classified by the npn isormorphism type of the Boolean function it implements.

8.4.1 Dynamic Grouping

For a circuit mapped into a DAG with m vertices, there are m EVs to represent the whole circuit. To produce a fixed-sized result, all vertices are partitioned into p pooling groups and an output is generated for each group in a standard format according to the set of EVs inside that group. Then the overall representation is a collection of data entries in the order of a chosen group ordering. Thus for a fixed number of groups p, all input circuits have identical data formats.

A naive approach for group ordering is to sort the vertices of a circuit in a topological order, and then uniformly partition the vertices into p groups, where each of the first (m mod p) groups contains $\lfloor \frac{m}{p} \rfloor + 1$ vertices, while others contain $\lfloor \frac{m}{p} \rfloor$ vertices. Thus, the first $\lfloor \frac{m}{p} \rfloor + 1$ vertices in the topological order are in the first group, the second $\lfloor \frac{m}{p} \rfloor + 1$ vertices are in the second group, etc.

However, there are many topological orders for the same DAG, and uniform distribution of vertices based on a topological order does not guarantee vertices in the same group are

connected or close. In Section 2.4, experiments with different topological orders are compared and discussed.

8.4.2 Most Representative Pooling

After partitioning vertices into groups, we need to determine certain representative features over groups for feeding successive layers.

A $k-$*most representative pooling* is proposed, which takes k "most representative" EVs for each group. The final matrix for each circuit has kp rows and |EV| (length of an EV) columns. To determine the most representative EVs for each group, its EVs are sorted first by occurrence counts, and then by the number of 1's in each EV in descending order. The first k EVs for each group are selected as rows of the matrix.

Example: In Figure 8.5 the four sEVs are sorted as $0011(2) \Rightarrow 0111(1), 1101(1) \Rightarrow 1100(1)$, where numbers in parentheses stand for occurrence counts. if $k = 2$, 0011 and one of 0111 and 1101 are chosen to represent this group.

8.5 Classification Framework

Figure 8.6 shows a framework devised for circuit classification based on the operations described in Section 8.3 and 8.4.

Given a set of circuits and a cell library, all circuits are mapped into DAGs, where vertices are indexed by library cells. Each vertex is assigned an EV by exploring its neighborhood. The output of the circuit-based convolution is a set of EVs. Then the circuit pooling layer divides vertices into a fixed number of groups and selects the most representative EVs for each group. After that, all circuits are represented as Boolean matrices with the same size.

Finally, a standard CNN (demonstrated in Figure 8.1) takes those matrices as input data and processes them with layers of operations, where parameters are trained to classify circuits.

8.6 Experimental Results

The proposed framework for circuit classification was implemented with Keras [9] and ABC [8] . All circuits were generated randomly in word-level Verilog and synthesized into gate-level circuits by Yosys [46]. Then ABC synthesized these into AIGs, performed technology mapping with the command *&if -K 4*, and executed operations described in Section 8.3 and 8.4 to write out Boolean matrices. Finally, CNNs are built and the models were trained with Keras packages. All procedures were run on a 16-core 2.60GHz Intel(R) Xeon(R) CPU.

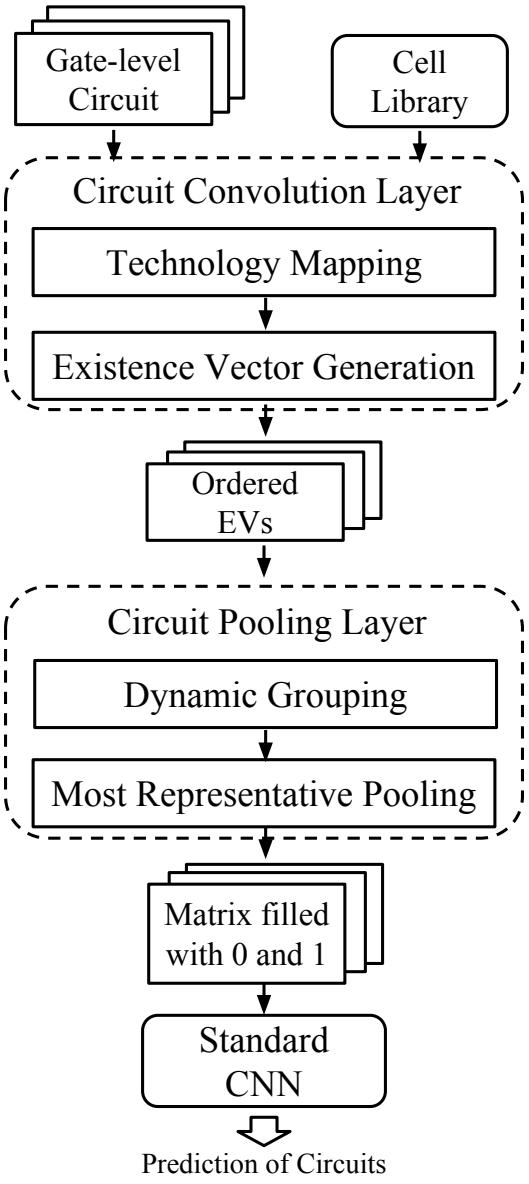


Figure 8.6: The proposed framework for circuit classification.

8.6.1 Experiment Setup and Runtimes

Data preparation. The library used was the set of npn isomorphism classes for all 4-LUTs (library size = 222). The number of pooling groups p was 40. For each group, the top three most representative EVs ($k = 3$) were taken. Thus for sEVs, the dimension of each input matrix is 120×222 and for dEVs, 120×444. The runtime of converting AIGs into Boolean matrices is sampled and shown in Table 8.1. The pre-processing time is relatively small and

grows linearly with the circuit size.

In addition to comparing dEVs and sEVs, we compared different topological orders: (1) the input order provided by synthesis (*Orig*), (2) ordering LUTs by depth-first search (*DFS*) and (3) breadth-first search (*BFS*).

Table 8.1: Sample runtime of converting AIGs into Boolean matrices.

AND #	168	1063	5505	10645	21485	28930
LUT #	43	246	1182	2299	4716	6118
Runtime (s)	0.102	0.102	0.135	0.173	0.285	0.335

Two reference methods. For a comparison against the CNN approach, SVM methods were used. As an input to such, we make a single vector of features out of all entries in an input matrix (generated with sEVs and the input topological orders.) The SVM methods implemented in scikit-learn [35] were used to classify the circuits.

To experiment with a faster method, a set of naive features were created to use with the SVM methods: circuits are mapped into 3-LUTs with the command *&if -K 3* in ABC, where the number of npn isomorphism classes is 14. Hence there are $14 \times 14 = 196$ types of LUT pairs connected by directed edges (fanin→LUT). The percentage of each connection type over the whole circuit is one feature having a range from 0 to 1. Each circuit is represented as a vector of 196 floating point numbers and then classified by the SVM method.

Using CNNs. In the following experiments, CNNs were built to distinguish two classes (to **detect** if a multiplier is present or not) or three classes (to **classify** the type of operator a circuit is - multiplier, divider or modulo operator.) All circuits in each class are distinct in operator types (for detection), or how they are synthesized (for classification), or their size of words (for both classification and detection). Circuits in each class are partitioned into a training set, a validation set and a testing set, where the size of each validation or testing set is fixed at 50 cases, while the size of the training set varies.

For all experiments, CNNs with the following layers in series were built:

1. a convolution layer with sixty-four 8×8 filters,
2. an ReLU activation layer,
3. a max-pooling layer with filters of size 2×2, like the example in Figure 8.3,
4. a dropout layer with dropout fraction 0.25,
5. a fully connected layer with 32 outputs,
6. an ReLU activation layer,
7. a dropout layer with dropout fraction 0.5,
8. a fully connected layer whose number of outputs is equal to the number of expected classes, and

Table 8.2: Statistics of running CNNs for different data formats.

Data Type	Parameter #	Training Time(s)	Testing Time(s)
sEV	13643938	254.241	7.465
dEV	27283618	501.476	13.497

9. a softmax activation layer for predictions (outputs).

The optimizer used for training was Adadelta [48], while the loss function was *categorical_crossentropy*.

In each individual experiment, a CNN was trained and optimized with the same training set for 30 *epochs*. One epoch means one full training cycle for the given training set. At the end of each epoch, the resulting CNN model is applied to the validation set. After each epoch, the best model and its validation result are saved. Training a CNN applies gradient descent methods and due to the existence of dropout layers, the actual training data after dropout layers differs in each epoch. After each epoch, if the accuracy rate is better than the best validation seen so far, the model and result are stored as "best". The final prediction accuracy is determined by applying the final reference model to the test set. The accuracy rates shown in Table 8.3 and Table 8.4 show the average and standard deviation of 10 separate runs, where each run had its data sets re-partitioned and re-shuffled.

Because the sizes of the matrices using dEVs and sEVs differ, the numbers of parameters in the corresponding CNNs differ. The sizes of the CNNs influence the training and prediction times. For both cases, Table 8.2 lists (1) the number of parameters, (2) the total runtime for 20 training rounds, where the sizes of training and validation sets are 100, and (3) the total runtime of testing on 100 circuits. These runtimes include reading data and model construction.

8.6.2 Operator classification.

Three classes of operators are considered for identification, multipliers, dividers and modulo operators with varying bit-widths. Training is done for CNNs with different training set sizes (the number of cases in each class). The resulting CNNs are used to predict the accuracy. The average accuracy and standard deviation for each experimental setting is listed in Table 8.3.

Comparing CNN with SVM. For CNNs and for all settings, the accuracy of prediction increases notably at the beginning and then becomes stable as the training sets get larger. The saturated accuracy of CNNs is above 97%. In comparison, when applying SVM to the data generated for CNNs, the accuracy rate can achieve 86% only. Thus, for the proposed data format, CNNs outperform SVM methods, seemingly by extracting more essential properties to recognize circuits.

The accuracy rate of the naive method is fixed at 66%, no matter the size of the training set, because it cannot distinguish dividers from modulo operators, while the proposed representation can do this.

Table 8.3: The average and standard deviation of accuracy rates for each setting of operator classification. The numbers under *Training Number* indicate the number of training cases in each class; the total training size is triple of the number.

Method Combination	Training Number								
	50	100	150	200	250	300	350	400	450
CNN sEV+Orig (%)	76.3 ±4.7	83.5±4.0	89.7±3.9	95.1±1.9	96.3±1.4	98.2±1.1	98.8±1.1	98.7±0.9	99.1±0.6
CNN dEV+Orig (%)	79.9±3.3	84.7±4.8	92.8±3.5	95.4±1.8	96.3±2.4	96.9±1.1	98.2±1.2	98.1±1.1	98.0±1.2
CNN sEV+DFS (%)	78.7±5.3	88.9±3.2	93.4±3.0	96.9±2.0	98.6±1.3	98.7±0.9	99.0±1.4	98.9±0.5	98.9±1.1
CNN dEV+DFS (%)	79.3±5.2	82.5±4.7	89.2±3.0	94.0±2.4	95.4±4.3	96.5±1.5	97.5±1.7	97.9±0.9	98.4±0.8
CNN sEV+BFS (%)	77.3±4.0	84.1±4.1	88.1±5.7	93.1±1.5	94.3±2.7	95.8±1.8	96.5±2.2	96.9±1.8	97.0±0.8
CNN dEV+BFS (%)	80.8±3.6	90.1±3.7	95.0±1.6	96.6±0.7	96.7±0.9	97.4±1.2	98.1±1.4	97.7±1.7	98.2±0.9
SVM sEV+Orig (%)	74.7±3.9	75.9±4.1	75.7±4.2	77.6±3.2	81.3±5.1	82.5±3.5	84.1±5.3	86.3±5.1	86.2±3.6
SVM Naive (%)	65.3±4.8	66.3±4.7	64.9±3.9	65.8±4.7	65.1±3.7	65.7±4.7	66.0±4.6	66.5±4.5	66.1±5.3

Table 8.4: The average and standard deviation of accuracy rates for each setting of operator detection. The value of n indicates the total number of arithmetic operators for each case, where at most one is a multiplier.

Method Combination	Total Number of operators (n)							
	2	3	4	5	6	7	8	9
CNN sEV+Orig (%)	99.1±0.7	95.7±1.9	91.4±2.1	89.7±2.1	87.5±2.6	82.4±3.2	82.9±5.0	75.8±5.7
CNN dEV+Orig (%)	99.3±0.5	95.2±2.4	92.4±3.7	89.5±3.3	88.6±3.0	86.0±4.3	84.8±4.1	75.3±5.0
CNN sEV+DFS (%)	98.9±1.0	96.0±2.1	92.3±3.1	89.7±2.2	87.5±1.8	84.4±3.3	80.8±5.8	76.5±3.7
CNN dEV+DFS(%)	99.0±1.9	96.6±1.1	93.0±2.4	89.3±3.0	87.4±4.7	84.7±2.7	78.3±9.0	73.6±14.8
CNN sEV+BFS (%)	99.1±1.0	96.4±2.2	93.0±2.3	91.6±2.2	92.9±2.9	89.8±2.7	83.9±5.9	83.4±4.1
CNN dEV+BFS(%)	99.6±0.5	97.5±1.5	91.8±1.5	92.2±3.1	91.0±4.4	87.8± 1.7	87.4±4.2	84.8±2.8
SVM sEV+Orig (%)	79.5±3.2	74.0±3.1	71.9±4.5	71.5±6.9	74.8±5.3	75.7±3.8	74.1±3.4	68.2±4.6
SVM Naive (%)	73.8±14.4	68.1±14.1	59.3±4.0	58.8±6.4	51.3±1.1	51.0±0.1	50.9±1.1	51.8±2.6

Additionally, for classifying just between multipliers and dividers, all methods can reach 100% accuracy even with the smallest training set. The experimental results show that the proposed method can distinguish not only distinct operators (divider and multiplier) but also when applied to quite similar operators (divider and modulo operator) at once.

Comparing different settings. dEVs can describe the connectivity of cells more precisely than sEVs, but CNNs using dEVs require double the runtime to train and test models (shown in Table 8.2). When using DFS orders, CNNs using dEVs require larger training sets than with sEVs to reach the same performance, because it takes extra efforts to train a model with more parameters. In contrast, when using BFS orders or the orders provided by synthesis (*Original*) with dEVs, the performance stabilizes with smaller training sets, versus sEVs which take more training data to stabilize.

The saturated accuracy rates of all settings for sEVs and dEVs are quite close. Hence for operator classification, it is better to use sEVs.

8.6.3 Operator detection

This experiment detects the existence of a multiplier embedded in a larger circuit. All cases tested were randomly generated with n arithmetic operators of varying bit-widths, where at most one is a multiplier (of varying size). These circuits are classified as to the absence or presence of a multiplier. We examine how the total number of operators (n) influences the accuracy of the prediction, considering different settings. The number of training cases for each class (with and without a multiplier) is fixed at 350, so the training set contains 700 cases.

Comparing CNN with SVM. Independent of the relative sizes of the embedded multipliers, Table 8.4 shows that the proposed CNN method for any of the settings outperforms the reference SVM methods. For circuits with only two operators, CNNs with all settings can reach 99% accuracy, while the SVM method on the same data can only achieve 79%.

The SVM approach with the naive features described in Section 8.6.1 fails to detect multipliers because its feature set omits local properties. As the relative sizes of the multiplier decrease, the accuracy rate goes down to 50%, which means basically no classification.

Comparing different settings. According to Table 8.4, CNNs for any setting can detect multipliers successfully when the total number of operators (n) is small. As n increases, data matrices generated with the BFS order are more suitable for operator detection than with both the DFS or input orders. When ordered in BFS, LUTs of the same operator tend to be grouped together. Hence the dynamic pooling can catch more representative features for each operator. There is no obvious benefit of using dEVs, so sEVs are better.

As the size of the multiplier decreases relative to the total circuit size, prediction accuracy decreases. For all the above experiments, the number of pooling groups p was fixed at 40. When n increases, each group covers a larger region, so some representative features of the target operator can be over-shadowed by sub-circuits in the same group. Increasing p can resolve this issue somewhat, but then the corresponding data matrices and the sizes of the

CNNs increase, which consumes more resources and runtime. To address this issue, we experimented with the use of sliding windows to sub-divide the circuit.

8.6.4 Operator Detection on Sub-circuits

Here a set of sub-circuits was extracted from a circuit using a sliding window, and then an already trained model was used to detect the existence of a multiplier for each window position.

Given a target circuit, the circuit convolution layer in Figure 8.6 was used to map the circuit into LUTs and associated with EVs. A set of sub-circuits is obtained by exploring the neighborhood (fanin LUTs, fanout LUTs and extended neighbors) of each LUT to a certain depth. Thus, each sub-circuit is a set of connected LUTs and each sub-circuit goes through the circuit pooling layer to generate a matrix for prediction. The set of LUTs used as the centers of windows for generating sub-circuits can be all LUTs or a set of LUTs randomly chosen across the whole circuit, because for many cases, using nearby LUTs as centers can result the same sub-circuit.

For detection, the presence of a multiplier can be determined by the existence of a sub-circuit with a high probability for containing a multiplier or a part of it. The overall accuracy rate is influenced by the following factors:

1. The already trained model was trained for detecting targets with a whole multiplier, not parts of it. Thus, a good model for detecting a whole multiplier may not be very good for finding parts of it.

2. The size of the window relative to the multiplier is crucial for the whole task. If the window is too small, each sub-circuit can only capture small parts of the multiplier; if the window is too large, the target multiplier may not be easy to detect.

3. The center LUTs influence the positions of the sub-circuits, which determine how much of the multiplier can be covered by each sub-circuit.

The above issues can be mitigated by increasing the number of sub-circuits that are generated with varying window sizes and centers. However, then a large number of sub-circuits may cause performance issues.

When the number of sub-circuits is large enough, this flow can be used also to locate the target operator, i.e., to find a set of LUTs (a sub-circuit), which comprises a multiplier. Hence we use the following process to find the *likelihood* of each LUT belonging to a multiplier.

Given a target circuit, numerous sub-circuits can be found with a moving window. For each sub-circuit, a trained multiplier detector predicts the probability (a score) of containing a multiplier. If a nLUT is in a window then it is given a score equal to the window probability. Thus an LUT is given a set of scores. After predicting all sub-circuits, each LUT keeps a set of scores coming from all sub-circuits containing it. Finally, for each LUT, the average of its set of scores is computed as its likelihood of being part of a multiplier.

Example of locating a multiplier. To show how the proposed flow works, consider a circuit with one multiplier surrounded by six other word-level operators (operators are connected together.) This circuit is mapped into 3151 LUTs, including 1 constant and 71 primary inputs. For each LUT, a neighborhood containing between 320 to 640 LUTs is saved as one sub-circuit. Based on the settings in Section 8.6.3, a multiplier detector, which is trained with $n = 2$ and sEVs using the input order, is applied for prediction on each sub-circuit. Figure 8.7 demonstrates the average score versus each LUT indexed using the topological order provided by a synthesis tool. All LUTs of one word-level operator are indexed continuously.

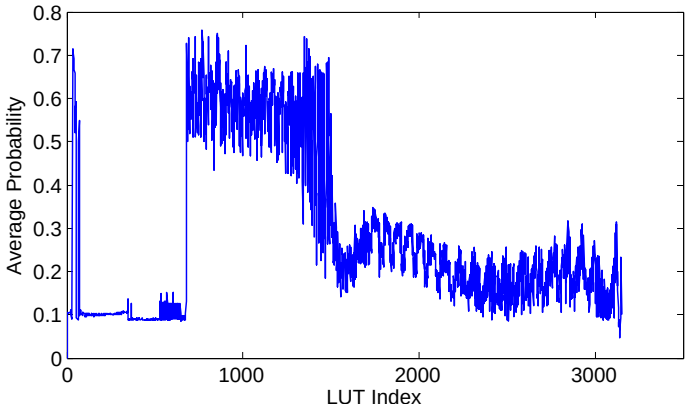


Figure 8.7: The average probability (likelihood) versus each LUT for an example circuit, which contains one multiplier.

There are two peak ranges in Figure 8.7: one is located at the primary inputs, the other covers LUTs 700 to 1400. The multiplier has one primary word and one internal word as inputs. Hence when considering neighborhoods of these primary inputs, the sub-circuits cover part of the multiplier and receive higher scores.

In the example, the multiplier feeds into another word-level operator. When mapped to LUTs, the boundaries between operators are unclear. Also, the neighborhoods of LUTs supported by multiplier outputs cover only parts of the multiplier. Therefore, for LUTs close to multiplier outputs, the average probabilities are quite variable.

The average scores contributed by sub-circuits can be used to locate the multiplier. Once an approximate location is found, more complex algorithms can be applied to the smaller region to identify the multiplier.

8.7 Summary

This chapter used one of the better-known deep learning methods, convolutional neural networks, for recognizing circuits that might be part of addressing computer-aided design and security problems.

Contributions are summarized below:

1. We enumerate essential requirements for representing circuits to work with CNNs. Most of these apply more generally to situations where an object is converted to an array of vectors of real numbers.

2. We propose a convolution operation which can represent both functional and structural properties of a circuit.

3. We propose dynamic grouping to partition a circuit into a fixed number of groups.

4. We devise *most representative pooling* to select features for each group.

5. We build a framework to a) classify or b) detect arithmetic operators from gate-level circuits.

6. We compared using CNNs against a supervised learning algorithm (SVM) where two different feature sets were tried. Experiments show that for the same training size, CNNs outperform SVMs .

Based on the spirit of using CNNs-complicated data preprocessing or feature extraction is not highly required, the proposed representation extracts essential local properties by the convolution operation for circuits, while global characteristics are caught by the circuit-based pooling. The representation works relatively well for the reverse engineering problems addressed in this chapter, but it can be enhanced by including more features. For example, based on adder tree structures found in a circuit, more columns can be added to indicate which types of adder trees exist in each group. This is a higher level feature which could be destroyed in the current representation. Extra features can assist on recognizing more sophisticated circuit properties, but it takes more manual efforts to extract and present them properly.

Future work might include: (1) revise the windowing techniques to improve operator detection, (2) apply the trained models to real industrial circuits to assist in reverse engineering, (3) use the proposed framework to recognize other combinational circuit properties, and (4) generalize the proposed framework to recognize designs implemented in different abstraction levels or libraries, including sequential circuits, word-level Verilog designs and SMT problems. With proper training data, the proposed framework could be used to train a model to help detect and locate malware in hardware designs.

Chapter 9

Conclusions and Future Directions

In this thesis, to address both verification and synthesis of clock-gated circuits, an SEC flow and a synthesis flow were proposed for work on sequential circuits, based on the fact that sequential clock-gating synthesis only inserts sequential redundancies to target circuits. In the spirit of reverse engineering, transparent logic blocks are recognized and used to construct dependency graphs to capture essential properties for clock-gating analysis. Legal clock-gating conditions are formulated with LTL and PLTL operators on DGs. Those properties can be proved on circuits with both safety and liveness model checkers. Experimental results showed that the proposed methodologies can effectively and efficiently verify proposed clock-gating conditions or to synthesize legal clock-gating conditions to reduce the frequencies of updating FFs.

The thesis is a prime example of the synergy between synthesis and verification. For verification Dependency Graphs (DG) were used to derive maximal properties that must be satisfied to be a legal clock-gating. These properties can be used to verify that a given circuit has been legally clock-gated. Even if the condition used in the given circuit is sub-optimal for clock-gating, the derived property can still be used to verify legality. This illustrates how the effort to derive a maximal condition is worthwhile because a maximal condition covers most cases done in practice. The conditions derived are described using LTL and PLTL temporal languages. Recent work on this described how to directly synthesize hardware monitors on the original sequential circuit. These provide additional outputs, which are targets of hardware model checkers and represent properties to be proved on the circuit. If proved, then the circuit was legally clock-gated. Dually, this same construction from the PLTL/LTL properties can be used to synthesize enabling signals which control new legal clock-gating for when a set of FFs needs to be updated. The maximality of the properties provide the conditions under which the most power is saved by not updating FFs.

To explore more possibilities for reverse engineering, we used convolutional neural networks to classify circuits and to detect special functions. We invented a representation of circuits to be used with modern machine learning techniques and built a circuit recognition framework based on CNNs. Experimental results showed that the proposed framework could achieve high rates of accuracy with proper training sets.

The major contributions of this thesis are listed below, with some discussion of how they might be extended to further applications:

- Transparent logic identification: we proposed functional approaches to identify transparent logic, which can be used to identify control and data flow and isolate arithmetic functional blocks. This method can be applied to general reverse engineering or assist in verification or synthesis.

- Dependency graphs: we defined a dependency graph, which is an abstraction model for a sequential circuit, in which the control and data dependencies are represented explicitly. By constructing the DG for an unknown circuit, more insights about its characteristics can be obtained allowing more suitable methods to be applied to it.

- Legal clock-gating conditions on abstraction models: LTL and PLTL properties are used to represent legal clock-gating conditions derived from abstraction models, which are the core techniques of the proposed verification and synthesis flow. The proposed method can be extended by recognizing other special functional blocks to provide new node types for DGs, such that the legal clock-gating conditions can be stronger.

- Circuit recognition with machine learning techniques: the proposed representation of circuits can be extended to include more features, such as the existence of adder trees. The proposed framework can be used to recognize different properties of circuit and to help in solving other computer-aided design problems, e.g. (1) prioritizing multiple properties for model checking, (2) choosing lazy or eager approaches for satisfiability modulo theories (SMT) problems, and (3) choosing the best synthesis techniques for circuits with special features.

Bibliography

[1] Steve Awodey. *Category theory*. Oxford Oxford New York: Clarendon Press Oxford University Press, 2006, pp. 11–12. ISBN: 9780198568612.

[2] Jason Baumgartner et al. "Scalable Sequential Equivalence Checking across Arbitrary Design Transformations". In: *International Conference on Computer Design*. IEEE. 2006, pp. 259–266.

[3] Armin Biere. *AIGER*. http://fmv.jku.at/aiger/.

[4] Armin Biere et al. "Linear encodings of bounded LTL model checking". In: *arXiv preprint cs/0611029* (2006).

[5] Armin Biere et al. "Symbolic model checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems* (1999), pp. 193–207.

[6] Christopher M Bishop. "Pattern recognition". In: *Machine Learning* 128 (2006), pp. 1–58.

[7] Robert K. Brayton, Niklas Een, and Alan Mishchenko. "Using Speculation for Sequential Equivalence Checking". In: *International Workshop on Logic and Synthesis*. 2012, pp. 139–145.

[8] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification*. Springer. 2010, pp. 24–40.

[9] Francois Chollet. *Keras*. 2015. URL: `https://github.com/fchollet/keras`.

[10] Koen Claessen, Niklas Een, and Baruch Sterin. "A circuit approach to LTL model checking". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE. 2013, pp. 53–60.

[11] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine Learning* 20.3 (1995), pp. 273–297. ISSN: 1573-0565. DOI: `10.1007/BF00994018`. URL: `http://dx.doi.org/10.1007/BF00994018`.

[12] Yu-Yun Dai, Kei-Yong Khoo, and Robert Brayton. "Sequential Equivalence Checking of Clock-Gated Circuits". In: *Design Automation Conference*. San Francisco, California: ACM, 2015.

[13] Howard B. Demuth et al. *Neural Network Design*. 2nd. USA: Martin Hagan, 2014. ISBN: 9780971732117.

[14] Niklas Een, Alan Mishchenko, and Robert K. Brayton. "Efficient Implementation of Property Directed Reachability". In: *Formal Methods in Computer-Aided Design*. 2011, pp. 125–134.

[15] Mark C Hansen, Hakan Yalcin, and John P Hayes. "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering". In: *IEEE Design & Test of Computers* 3 (1999), pp. 72–80.

[16] *Hardware Model Checking Competition 2013*. http://fmv.jku.at/hwmcc13/.

[17] *Hardware Model Checking Competition 2014*. http://fmv.jku.at/hwmcc14cav/.

[18] Yen-Sheng Ho, Alan Mishchenko, and Robert Brayton. "Uninterpreted Function Abstraction and Refinement for Word-level Model Checking". In: *International Workshop on Logic and Synthesis (IWLS)*. 2016.

[19] David H Hubel and Torsten N Wiesel. "Receptive fields and functional architecture of monkey striate cortex". In: *The Journal of physiology* 195.1 (1968), pp. 215–243.

[20] Aaron P. Hurst. "Automatic Synthesis of Clock Gating Logic with Controlled Netlist Perturbation". In: *Design Automation Conference*. Anaheim, California: ACM, 2008, pp. 654–657. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391637. URL: http://doi.acm.org/10.1145/1391469.1391637.

[21] Jie-Hong R Jiang and Wei-Lun Hung. "Inductive equivalence checking under retiming and resynthesis". In: *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*. IEEE Press. 2007, pp. 326–333.

[22] Rie Johnson and Tong Zhang. "Effective use of word order for text categorization with convolutional neural networks". In: *arXiv preprint arXiv:1412.1058* (2014).

[23] Victor Kravets et al. *Advanced Techniques in Logic Synthesis, Optimizations and Applications*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 9781441975171.

[24] Andreas Kuehlmann and Cornelis AJ Kuehlmann. "Combinational and sequential equivalence checking". In: *Logic synthesis and Verification* (2002), pp. 343–372.

[25] Yann LeCun et al. *LeNet-5, convolutional neural networks*. 2015. URL: http://yann.lecun.com/exdb/lenet.

[26] Y. LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324.

[27] Wenchao Li. "Formal Methods for Reverse Engineering Gate-Level Netlists". MA thesis. EECS Department, University of California, Berkeley, Dec. 2013.

[28] Wenchao Li, Zach Wasson, and Sanjit A Seshia. "Reverse engineering circuits using behavioral pattern mining". In: *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 83–88.

[29] Wenchao Li et al. "WordRev: Finding Word-Level Structures in a Sea of Bit-Level Gates". In: *Proceedings of the IEEE Conference on Hardware-Oriented Security and Trust (HOST)*. June 2013.

[30] Subhransu Maji and Jitendra Malik. *Fast and accurate digit classification*. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-159., 2009.

[31] Alan Mishchenko et al. "Technology mapping into general programmable cells". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, pp. 70–73.

[32] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. "Learning Convolutional Neural Networks for Graphs". In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2016.

[33] *OpenCores*. http://opencores.org/.

[34] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*. Vol. 1. Prentice Hall Professional, 2003.

[35] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[36] J. R. Quinlan. "Induction of Decision Trees". In: *Mach. Learn.* 1.1 (Mar. 1986), pp. 81–106. ISSN: 0885-6125. DOI: 10.1023/A:1022643204877. URL: http://dx.doi.org/10.1023/A:1022643204877.

[37] R. Ranjan, V. M. Purri, and F. Braga. "Hardware verification of security aspects: scalable approaches in formal system-level security verifications". In: *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE. 2016.

[38] Sayak Ray et al. "Mapping into LUT structures". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2012, pp. 1579–1584.

[39] A. Saldanha et al. "Multi-level Logic Simplification Using Don'T Cares and Filters". In: *Design Automation Conference*. Las Vegas, Nevada, USA: ACM, 1989, pp. 277–282. URL: http://doi.acm.org/10.1145/74382.74429.

[40] Hamid Savoj, Alan Mishchenko, and Robert Brayton. "Sequential Equivalence Checking for Clock-Gated Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.2 (2014), pp. 305–317.

[41] Hamid Savoj et al. "Combinational Techniques for Sequential Equivalence Checking". In: *Formal Methods in Computer-Aided Design*. FMCAD Inc. 2010, pp. 145–150.

[42] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Formal Methods in Computer-Aided Design*. London, UK, UK: Springer-Verlag, 2000, pp. 108–125. ISBN: 3-540-41219-0. URL: http://dl.acm.org/citation.cfm?id=646186.683237.

[43] Mathias Soeken et al. "Heuristic NPN classification for large functions using AIGs and LEXSAT". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2016.

[44] Mathias Soeken et al. "Reverse Engineering with Simulation Graphs". In: *Formal Methods in Computer-Aided Design*. 2015.

[45] Pramod Subramanyan et al. "Reverse Engineering Digital Circuits Using Structural and Functional Analyses". In: *Emerging Topics in Computing, IEEE Transactions on* 2.1 (2014), pp. 63–80.

[46] Clifford Wolf. *Yosys Open SYnthesis Suite*. URL: `http://www.clifford.at/yosys/`.

[47] Cunxi Yu et al. "DAG-aware logic synthesis of datapaths". In: *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE. 2016, pp. 1–6.

[48] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). URL: `http://arxiv.org/abs/1212.5701`.