SAT-based Automated Design Debugging: Improvements and Application to Low-Power Design

by

Bao Le

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by Bao Le
Abstract

SAT-based Automated Design Debugging:
Improvements and Application to Low-Power Design

Bao Le
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2012

With the growing complexity of modern VLSI designs, design errors become increasingly common. Design debugging today emerges as a bottleneck in the design flow, consuming up to 30% of the overall design effort. Unfortunately, design debugging is still a predominantly manual process in the industry. To tackle this problem, we enhance existing automated debugging tools and extend their applications to different design domains.

The first contribution improves the performance of automated design debugging tools by using structural circuit properties, namely dominance relationships and non-solution implications. Overall, a 42% average reduction in solving run-time demonstrates the efficacy of this approach.

The second contribution presents an automated debugging methodology for clock-gating design. Using clock-gating properties, we optimize existing debugging techniques to localizes and rectifies the design errors introduced by clock-gating implementations. Experiments show a 6% average reduction in debugging time and 80% of the power-savings retained.
Acknowledgements

First and foremost I would like to express my sincere gratitude to my supervisor, Professor Andreas Veneris, for his continuous support, guidance, encouragement, patience and dedication to my research career. You are truly an inspiration to me.

Many thanks to my Master committee members Professor Jason Anderson, Vaughn Betz, and Konstantinos N. Plataniotis for their thorough reviews and their insightful suggestions.

I would also like to thank my family for believing in me, supporting me in every way they could. My parents have always been there loving me wholeheartedly and for that I am extremely grateful. I would like to thank my fiance, Van Vu, for sticking with me through thick and thin, and for always cheering me up.

I would like to thank my friends in Toronto, Patrick, Ian, Justin, Steven, Eric and Jay, for always helping me during my time here. My time in Toronto would have been much harder without you guys.

I am also indebted to my colleagues at Vennsa and the University of Toronto for all their advice and feedback. Special thanks to Hratch Mangassarian, Brian Keng, Dipanjan Sengupta, Zissis Poulos, Sean Safarpour, Duncan Smith, Terry Yang, Evean Qin and Alan Baker.

Acknowledgments are also due to the University of Toronto for their financial support.
Contents

List of Tables vii
List of Figures viii
List of Algorithms x

1 Introduction 1
  1.1 Background and Motivation ........................................ 1
  1.1.1 Design Verification and Design Debugging ....................... 3
  1.2 Contribution .......................................................... 5
  1.2.1 Non-Solution Implications using Reverse Domination in a Modern SAT-based Debugging Environment ........................................ 6
  1.2.2 Reviving Erroneous Stability-based Clock-gating using Partial Max-SAT ........................................ 7
  1.3 Thesis Outline .......................................................... 8

2 Background 9
  2.1 Introduction ............................................................ 9
  2.2 Boolean Satisfiability .................................................... 9
  2.2.1 CNF Representation .................................................. 10
  2.2.2 SAT Algorithms ...................................................... 11
  2.2.3 All-Solutions SAT .................................................... 15
  2.2.4 Partial Max-SAT ...................................................... 17
  2.3 Verification and Debugging Overview ................................ 19
## 2.4 Automated Design Debugging

- 2.4.1 Automated Design Debugging with Error-select Variables
- 2.4.2 Automated Design Debugging with Partial Max-SAT

## 2.5 Clock-gating for Low-power Design

## 2.6 Summary

## 3 Non-Solution Implications using Reverse Domination in a Modern SAT-based Debugging Environment

- 3.1 Introduction
- 3.2 Dominance Relationships and Previous Work
- 3.3 Non-Solution Implications using Reverse Domination
- 3.4 Decision Heuristic for Early Non-Solution Learning
  - 3.4.1 Decision Heuristic
  - 3.4.2 Detecting Non-Solution Blocks
  - 3.4.3 Non-Solution Block Propagation
  - 3.4.4 Restart Heuristics
  - 3.4.5 Overall Modified SAT Algorithm
- 3.5 Experimental Results
- 3.6 Summary

## 4 Reviving Erroneous Stability-based Clock-gating using Partial Max-SAT

- 4.1 Introduction
- 4.2 Previous Work
- 4.3 Clock-gating Preprocessing
  - 4.3.1 Definitions
  - 4.3.2 Erroneous Components Identification
- 4.4 Clock-gating Debugging
- 4.5 Clock-gating Node Rectification
- 4.6 Experimental Results
- 4.7 Summary
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simple gates and their CNF representation</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Instance Information</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Design Debugging with Non-Solution Implications and Reverse Dominance Results</td>
<td>48</td>
</tr>
<tr>
<td>3.3</td>
<td>Design Debugging with Non-Solution Implications and Reverse Dominance Results</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Instance Information - Table 1</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>Instance Information - Table 2</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Debugging with Pre Processing Results</td>
<td>73</td>
</tr>
<tr>
<td>4.4</td>
<td>Debugging with Rectification Results</td>
<td>74</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Simplified VLSI flow</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Flow chart of typical manual debugging process</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>A SAT Algorithm Example</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Typical debugging flow</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Time-frame expansion example</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Design Debugging with Error-Select Variables Formulation</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Stability-based Clock-gating example</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Clock-gated Design</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Reverse Dominator Example</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Non-solution Implications Example</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Decision Tree Example</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Non-solution blocks using Decision Tree</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Non-solution blocks using our decision heuristic</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Ratio of implied non-solutions to all non-solutions using \texttt{dbg-dom+rev+RR}</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Number of solutions vs run-time for rsdecoder2</td>
<td>50</td>
</tr>
<tr>
<td>3.8</td>
<td>Number of implied non-solutions vs run-time for rsdecoder2</td>
<td>51</td>
</tr>
<tr>
<td>3.9</td>
<td>Performance Results</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>A Clock-gating Design</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Clock-gated Design</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Enhanced Circuit: $C_e$</td>
<td>60</td>
</tr>
</tbody>
</table>
4.4 Clock-gating Transformation
## List of Algorithms

1. DPLL-based SAT Procedure .................................................. 11
2. A modern DPLL-based SAT procedure ................................. 13
3. SAT Solver for Design Debugging ........................................ 44
4. clock-gating Nodes Computation ......................................... 58
5. clock-gating Debug Preprocessing .................................... 63
6. Clock-gating Fault Localization ......................................... 66
7. Clock-gating Node Rectification ....................................... 68
Chapter 1

Introduction

1.1 Background and Motivation

The relentless demand for complex and superior devices continues to drive the growth of the semiconductor industry. Electronic devices are getting smaller in physical size, superior in performance and lower in power consumption\(^3\). For instance, in 1971, the first complete CPU on chip, Intel 4004, was introduced with only 2,300 transistors\(^6\) while in 2011, an Intel i7 processor surpassed the 2 billion transistor count\(^24\). This growth requires the electronic design community to keep innovating and improving existing design flows. In fact, many interdependent steps in the design flow for Very Large Scale Integration (VLSI) chips have been fully or partially automated using Computer Aided Design (CAD) tools. These tools are essential as they automate steps which would otherwise be intractable if performed manually. Constant research and innovation in these tools have given us the ability to build the complex computer systems available today.

With the complexity of modern VLSI chips, the design flow has become an intricate process consisting of many steps. At each step, a higher level description of the circuit is synthesized into a lower level one. Figure 1.1 shows a simplified version of the design flow. First, a specification written in high level languages such as C or Matlab is converted into a Register Transfer Level (RTL) description usually written in Verilog or VHSIC hardware description language (VHDL). Next, a gate-level netlist is constructed from the RTL. Place and route tools then generate the
Figure 1.1: Simplified VLSI flow
chip's physical layout from the netlist. Finally, the physical layout is sent to the fabrication facility for manufacturing.

In the design flow, multiple verification phases are run to ensure that no errors are introduced. Functional verification guarantees that the netlist corresponds to its specification. Post-silicon testing compares the physical chip with its netlist. When a verification phase fails, diagnosis is performed to locate the cause of the failure. Specifically, if a failure is found during functional verification, design debugging is performed to locate design errors in the netlist. If a failure is found during test, a similar process, known as silicon debug, is performed to locate design errors that escaped to silicon or physical defects introduced during fabrication. It is preferable to find errors earlier rather than later in the flow as errors found late increase the overall design cost.

1.1.1 Design Verification and Design Debugging

Functional verification is the process of determining whether a design conforms to its specification. Many techniques and methodologies have been developed both in academia and in the industry to enhance and automate functional verification. The introduction of equivalence checking\cite{22}, property checking\cite{7}, automated test-bench generation\cite{1}, assertion-based verification (ABV)\cite{16}, and the use of Binary Decision Diagrams (BDDs)\cite{2, 5} and Boolean Satisfiability (SAT) solvers\cite{11, 50} has dramatically changed how functional verification is done. A verification engineer, who once heavily relied on simulation-based methods, today has many more tools at his/her disposal. In general, these new techniques and tools have eased the difficulty of verification.

Once the design exhibits a discrepancy from its specification, design debugging is performed to locate the cause of the discrepancy. Typically, design debugging starts after a verification engine produces a counter-example demonstrating that the design is erroneous. The objective of design debugging is to locate potentially erroneous components/blocks/gates in the design. Once these are found, the designer or the verification engineer performs rectification to remove the bug(s). To ensure that no bug still exists in the design after rectification, the verification process is repeated.
Although verification has been partially automated, design debugging remains a predominantly manual process. Manually locating the erroneous components is a difficult task since the engineer must search through both space (RTL modules, gates, etc.) and time (clock-cycles). As a result, this process requires a detailed understanding at both the system level and the block level. Given the complexity of modern designs, such massive knowledge is generally not available to a single designer/engineer. Figure 1.2 displays a contemporary debugging practice in the industry. Note that a bug may require the time and effort of many verification engineers and designers before it is fixed. A hard-to-find bug can delay the design cycle greatly and jeopardize the release date of the product. With today’s stringent time-to-market constraints, this can cause many undesirable financial consequences. Therefore, automating design debugging is vital for improving overall design efficacy.

Figure 1.2: Flow chart of typical manual debugging process.
Many methodologies have been proposed to automate design debugging and to alleviate its cost \cite{12, 32, 48, 49, 51, 52}. Due to recent advancements in formal engines \cite{14, 37, 39, 56}, many modern debugging techniques use SAT solvers \cite{49}. In these SAT-based techniques, the debugging problem is encoded as a SAT instance. Next, the instance is given to a SAT solver. Each solution to the SAT instance corresponds to a potentially erroneous component. These techniques have shown promising results in terms of scalability and run-time when compared to traditional methods based on simulation and BDDs \cite{55}.

Despite great advancements produced over the last decade, the adoption of automated design debugging in the industry is still limited. One of the reasons is that academic automated design debugging solutions do not cope well with the complexity of industrial problems. With designs reaching billion of transistors, their debugging propositional instances have become a challenging test even to modern SAT solving engines. Moreover, there is a need today for debugging solutions tailored specifically to problems arising from low-power design or retiming. These reasons limit the use of automated design debugging techniques and hinder their adoption in the industry. Thus, automated design debugging must be improved in terms of performance and must expand its application to different design domains.

1.2 Contribution

This thesis offers research that improves SAT-based design debugging in two novel ways. The first research contribution improves the performance of SAT-based design debugging by using structural circuitry properties, namely dominance relationships. A SAT solver utilizing dominance relationships to solve debugging instances efficiently is presented. The second research contribution optimizes SAT-based design debugging for low-power design. Specifically, a novel debugging methodology for clock-gated circuits is proposed. Enhancing automated design debugging in these two aspects will make automated design debugging more practical in an industrial environment.

In summary, this thesis makes the following contributions:

- A novel design debugging technique is proposed which uses structural circuit properties,
known as *dominance relationships*. A SAT solver is tailored to leverage reverse dominators for early on-the-fly detection of bug-free components. The early pruning of these components significantly reduces the debugging solution space.

- A SAT-based debugging methodology for clock-gating design is proposed. By utilizing clock-gating design properties, the design error introduced during a clock-gating implementation is localized and rectified in an efficient manner.

In the following two sub-sections, both of these contributions are discussed in greater detail.

### 1.2.1 Non-Solution Implications using Reverse Domination in a Modern SAT-based Debugging Environment

Today, bigger designs and longer counter-examples have made debugging a resource-intensive task. With the typical design block size exceeding half a million gates, the propositional formulas encoding design debugging problems can contain tens of millions of variables and clauses. Solving such large instances is challenging even for modern SAT solvers.

The first part of this thesis presents a technique that allows the SAT solver to reduce the complexity of the design debugging problem. Using *dominance relationships*, we introduce the dual concepts of *reverse domination* and *non-solution implications*. A SAT solver is tailored to leverage reverse dominators in order to identify bug-free components in the design. These components are then pruned from the solution space and this pruning in effect reduces the complexity of the problem.

The major contributions of this work can be summarized as follows:

- A SAT solver is tailored to leverage reverse domination relationships for early detection of bug-free components.

- A SAT solver is tailored to utilize bug-free components for early pruning of non-solution area in the search space. As a result, the modified SAT solver performance on design debugging instance is improved significantly.
An extensive set of experiments on real industrial designs demonstrates the practicality of this work. These experiments confirm that bug-free components can be identified early in the process. All instances show improvements when employing this new technique. In particular, a 42% average reduction in solving run-time compared to the state-of-the-art SAT-based debugger is observed.

1.2.2 Reviving Erroneous Stability-based Clock-gating using Partial Max-SAT

As low-power implementation is ubiquitous in modern VLSI chip design, designers are under immense pressure to adopt aggressive power-saving techniques. Although progress has been made in automating such implementations, significant manual effort is still required to realize savings. Due to inherent design complexity and the human factor, power-saving methods can introduce design errors. When such errors occur, debugging the design to identify the root cause of failure is an overwhelming and time-consuming process. Consequently, designers often get discouraged and fail to maximize power-savings.

Clock-gating is a circuit transformation where the clock input of a register no longer latches in a new value but instead holds its current value. This enables power-savings as unnecessary switching of gates can be avoided.

The second part of this thesis introduces a design debugging methodology for low-power designs, targeting clock-gated circuits. The presented research leverages existing design debugging techniques and optimizes them to operate under a clock-gating synthesis environment. To expand, we first introduce a fast preprocessing step that uses clock-gating properties to identify bug-free components. The discovery of these components reduces the complexity of the debugging problem. Possible fixes are suggested so that the error is corrected and power-saving is not sacrificed. Overall, the proposed method reduces the debugging time without sacrificing all power-savings. This gives the designer more confidence and time to realize all the available power-savings.

The major contributions of this research can be summarized as follows:

- A fast preprocessing step is introduced to identify bug-free components in an erroneous
clock-gating design.

- A rectification procedure is presented. The suggested fixes correct the error and partially, if not fully, retain the power-savings achieved.

Experiments on standard benchmark circuits show the effectiveness of this method. The preprocessing step does not pose any significant overhead and it has the potential to reduce the complexity of the SAT instance significantly. When compared to the state-of-the-art design debugging method, a 6% average reduction in debugging time is observed. Moreover, 80% of the power-savings can be retained through this rectifying technique.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 contains background information on design debugging and clock-gating. Chapter 3 presents the SAT-based design debugging technique that uses non-solution implications and reverse domination concepts. Chapter 4 proposes a SAT-based design debugging methodology for clock-gating designs. Finally, Chapter 5 offers the conclusion.
Chapter 2

Background

2.1 Introduction

This chapter presents background material. Particularly, Section 2.2 discusses Boolean Satisfiability and modeling techniques for sequential circuits. Section 2.3 provides an overview of functional verification and design debugging. Section 2.4 introduces two automated design debugging techniques: an all-solutions SAT-based technique and a Partial Max-SAT based technique. Finally, Section 2.5 describes clock-gating for low-power designs.

2.2 Boolean Satisfiability

This section introduces relevant concepts regarding Boolean Satisfiability. Before formally defining Boolean Satisfiability, we will examine propositional logic formulas. A propositional logic formula $\Phi$ is built over a set of Boolean variables $x_1, x_2, \ldots, x_n$, where $x_i \in \{\text{true, false}\} = \{0, 1\}$, using Boolean connectives such as negation ($\neg$), conjunction ($\land$) and disjunction ($\lor$).

**Definition 1**  *Given a propositional logic formula $\Phi$ over a set of Boolean variables $\langle x_1, x_2, \ldots, x_n \rangle$, the Boolean Satisfiability problem asks whether there exists an assignment of logic values that evaluates $\Phi$ to true (1). If such an assignment exists, the instance $\Phi$ is said to be satisfiable (SAT), otherwise it is unsatisfiable (UNSAT).*
Boolean Satisfiability plays an important role in computation and complexity theory as it is
the first proven NP-complete problem\[9\]. Boolean Satisfiability also has significant practical
applications as a wide range of decision and optimization problems are encoded as SAT instances
and can be solved in this manner. In fact, many SAT solving algorithms, called SAT solvers,
have been developed to efficiently solve instances derived from practical applications. The
development of efficient SAT solvers has increased their adoption in various CAD tools. Today,
numerous problems in synthesis, verification, debugging and test are formulated as propositional
formulas and solved by SAT engines.

2.2.1 CNF Representation

This subsection describes conjunctive normal form (CNF), a representation of propositional
logic formulas that most modern SAT solvers use.

**Definition 2** A Boolean logic formula is said to be in CNF form if it is a conjunction (\(\land\)) of
clauses where each clause is a disjunction (\(\lor\)) of literals. A literal is a representation of the
positive or the negative phase of a variable.

Clearly, an assignment that satisfies \(\Phi\) in CNF must satisfy every clause of \(\Phi\). For a clause
to be satisfied, at least one of its literals must evaluate to 1.

**Example 1** Consider the following Boolean formula in CNF representation:

\[
\Phi = (x_1 \lor \bar{y} \lor s) \land (\bar{x_1} \lor y \lor s) \land (x_2 \lor \bar{y} \lor \bar{s}) \land (\bar{x_2} \lor y \lor \bar{s}) \land (y)
\]  

(2.1)

\(\Phi\) is SAT as \(\{s = 1, x_1 = 0, x_2 = 1, y = 1\}\) is a satisfying assignment of \(\Phi\).

1. \((x_1 \lor \bar{y} \lor s)\) and \((\bar{x_1} \lor y \lor s)\) are SAT since \(s = 1\)
2. \((x_2 \lor \bar{y} \lor \bar{s})\) is SAT since \(x_2 = 1\)
3. \((\bar{x_2} \lor y \lor \bar{s})\) and \((y)\) are SAT since \(y = 1\)

Logic circuits can be encoded in CNF using the Tseitin transformation with the use of
auxiliary variables\[53\]. The Tseitin transformation first generates a CNF formula describing
the behavior of each gate in the circuit and then it conjuncts all of these formulas. Thus, any assignment that satisfies the CNF formula of the entire circuit also satisfies the formula of each and every gate in it. Table 2.1 shows CNF representations for most common logic gates.

**Example 2** The formula $\Phi$ in Example 1 is actually a CNF representation of a MUX with the output $y$, the input $x_1, x_2$ and the select line $s$. The output of the MUX is constrained to 1.

### 2.2.2 SAT Algorithms

Modern SAT solvers such as miniSAT and zChaff are based on the Davis-Putnam-Logemann-Loveland (DPLL) search algorithm. The DPLL algorithm can be seen as a depth-first-search on a binary tree representing the search space of a SAT instance. In this tree, each level is a variable, each node is a partial assignment and each edge is an assignment (0 or 1) to a variable. A DPLL-based SAT algorithm finds a node where an assignment evaluates the CNF formula to 1 or confirms that no such node exists.

Algorithm 1 shows a basic DPLL procedure, `DPLL SAT()`. `DPLL SAT()` returns true if and only if the CNF formula is SAT. Initially, the procedure is passed an empty assignment $A = \emptyset$. At each iteration, the formula becomes UNSAT if there exists an unsatisfied clause (line 2), or SAT if all clauses are satisfied (line 3). Otherwise, `DPLL SAT()` selects the next unassigned variable $v$ to add to the partial assignment $A$ (line 4). `DPLL SAT()` is called recursively with the new partial assignment $A \cup \{(v = 0)\}$. If $(v = 0)$ results in an unsatisfied clause, $(v = 1)$ is tried (line 5). The procedure evaluates all possible assignments until finding a satisfying assignment or confirming the formula is UNSAT.

**Algorithm 1: DPLL-based SAT Procedure**

1. `DPLL SAT (Partial Assignment A)`
2. `if (contains_unsatisfied_clause()) then return false;`
3. `if (all_clauses_satisfied()) then return true;`
4. `v = next_unassigned_variable();`
5. `return DPLL SAT (A \cup \{v = 0\}) \lor DPLL SAT (A \cup \{v = 1\})`
<table>
<thead>
<tr>
<th>Gates</th>
<th>CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = \text{AND}(x_1, x_2, \ldots, x_n) )</td>
<td>((x_1 \lor \overline{y}) \land (x_2 \lor \overline{y}) \ldots \land (x_n \lor \overline{y}))</td>
</tr>
<tr>
<td></td>
<td>((\overline{x_1} \lor \overline{x_2} \lor \ldots \lor \overline{x_n} \lor y))</td>
</tr>
<tr>
<td>( y = \text{NAND}(x_1, x_2, \ldots, x_n) )</td>
<td>((x_1 \lor y) \land (x_2 \lor y) \ldots \land (x_n \lor y))</td>
</tr>
<tr>
<td></td>
<td>((\overline{x_1} \lor \overline{x_2} \lor \ldots \lor \overline{x_n} \lor y))</td>
</tr>
<tr>
<td>( y = \text{OR}(x_1, x_2, \ldots, x_n) )</td>
<td>((x_1 \lor x_2 \lor \ldots \lor x_n \lor \overline{y}))</td>
</tr>
<tr>
<td>( y = \text{NOR}(x_1, x_2, \ldots, x_n) )</td>
<td>((x_1 \lor x_2 \lor \ldots \lor x_n \lor \overline{y}))</td>
</tr>
<tr>
<td></td>
<td>((x_1 \lor x_2 \lor \ldots \lor x_n \lor \overline{y}))</td>
</tr>
<tr>
<td>( y = \text{XOR}(x_1, x_2) )</td>
<td>((\overline{x_1} \lor x_2 \lor y)(x_1 \lor \overline{x_2} \lor y))</td>
</tr>
<tr>
<td></td>
<td>((x_1 \lor x_2 \lor \overline{y})(\overline{x_1} \lor x_2 \lor y))</td>
</tr>
<tr>
<td>( y = \text{XNOR}(x_1, x_2) )</td>
<td>((\overline{x_1} \lor x_2 \lor y)(x_1 \lor \overline{x_2} \lor y))</td>
</tr>
<tr>
<td></td>
<td>((x_1 \lor x_2 \lor \overline{y})(\overline{x_1} \lor x_2 \lor y))</td>
</tr>
<tr>
<td>( y = \text{NOT}(i) )</td>
<td>((i \lor y) \land (\overline{i} \lor \overline{y}))</td>
</tr>
<tr>
<td>( y = \text{BUFFER}(i) )</td>
<td>((i \lor \overline{y}) \land (\overline{i} \lor y))</td>
</tr>
<tr>
<td>( y = \text{MUX}(s, x_1, x_2) )</td>
<td>((x_1 \lor \overline{y} \lor s) \land (\overline{x_1} \lor y \lor s))</td>
</tr>
<tr>
<td></td>
<td>((x_2 \lor \overline{y} \lor s) \land (\overline{x_2} \lor y \lor s))</td>
</tr>
</tbody>
</table>

Table 2.1: Simple gates and their CNF representation
Modern SAT solvers optimize the basic DPLL procedure shown in Algorithm 1 for industrial instances. They use an iterative procedure, `modern_DPLL_SAT()`, shown in Algorithm 2. Major advancements in `modern_DPLL_SAT()` are grouped into three areas: decision heuristics (`decide()`), Boolean constraint propagation (`bcp()`), and conflict analysis (`analyze_conflict()`). Each of these is visited in the following subsections.

In Algorithm 2, `decide()` selects the next unassigned variable on which to branch (line 3). If all variables are assigned, `decide()` returns `false` and the CNF formula is `SAT`. Otherwise, `decide()` gives the unassigned variable a value, also called `decision`. Next, `bcp()` finds all the implied assignments (line 7), also known as `implications`. If `bcp()` results in an unsatisfied clause, it returns `false` indicating that a conflict has occurs. The `analyze_conflict()` function is called (line 9) to resolve the conflict. `analyze_conflict()` finds the earliest `decision` that causes the formula to be unsatisfiable and undoes it by flipping the value of the decision assignment. If the conflict cannot be resolved, `analyze_conflict()` returns `false` and the CNF formula is `UNSAT`. 

---

**Algorithm 2: A modern DPLL-based SAT procedure**

1. `modern_DPLL_SAT()`
2. while (true) do
   3. if (!`decide()`) then
      // All clauses are satisfied
      4. return SAT;
   6. end
    7. while (!`bcp()`) do
       // Conflict encountered in `bcp()`
       9. if (!`analyze_conflict()`) then return UNSAT;
    11. end

---
2.2.2.1 Decision Heuristics

In Algorithm 2, the `decide()` function selects an unassigned variable and assigns a value to it. The order of these assignments can have a major impact on the performance of a SAT solver. Decision heuristics are techniques used to optimize `decide()`. Many modern SAT solvers deploy the variable state independent decaying sum (VSIDS) decision heuristic introduced in the zChaff solver. VSIDS is a dynamic decision strategy which gives preference to the most recently discovered information.

2.2.2.2 Boolean Constraint Propagation

Boolean constraint propagation (bcp()) computes the implied assignments by the current partial assignment. It is based on the “unit clause rule”. A unit clause is a clause in which one literal is unassigned and all other literals evaluate to 0 under the current partial assignment. In order to satisfy the unit clause, the unassigned literal has to evaluate to 1. In the event that two unit clauses force a variable to both 0 and 1, a conflict occurs. A conflict indicates that the current partial assignment cannot be extended to a satisfying assignment.

The bcp() function, which identifies unit clauses, accounts for 90% of a SAT solver’s runtime. Hence, an efficient bcp() engine is essential for any SAT solver. A simple but inefficient way of identifying unit clauses is to use a counter for each clause in order to keep track of the number of literals set to 0. However, these counters must be incremented or decremented whenever a variable is assigned or unassigned respectively. To alleviate this problem, the pioneering zChaff SAT solver introduces the two watched literals mechanic for the bcp() function. In zChaff, each clause contains two watched literals which are not assigned to 0. This guarantees that until one of the watched literals is set to 0, the clause cannot be a unit clause. When one watched literal is assigned to 0, the function attempts to watch a new literal that is not assigned to 0 in the clause. If all other literals are assigned to 0, the clause is a unit clause, and the other watched literal value is implied. Clearly, this technique reduces the number of visits to a clause as only assignments on watched literals require visits. Moreover, it allows constant backtracking (i.e., any decision assignment can be switched in a constant time). This significantly improves performance.
2.2.2.3 Conflict Analysis and Other Advancements in SAT Algorithms

When \( bcp() \) results in a conflict, the \texttt{analyze\_conflict()} function is called to resolve it. The \texttt{analyze\_conflict()} function finds the earliest \textit{decision} that is the cause of the conflict and flips the assignment value for that decision. This allows the algorithm to backtrack in a non-chronological order as the decision that is switched may not be the most recently made decision. Modern SAT solvers with effective conflict analysis and non-chronological backtracking techniques have the ability to prune out large parts of the search space, resulting in significantly improved performance.

There have been many other advancements in SAT solving. Techniques such as restarts\textsuperscript{[21, 39]}, parallel processing\textsuperscript{[44]} and pre-processing\textsuperscript{[13]} have further improved the performance of modern SAT solvers.

**Example 3** Consider the following Boolean formula in CNF:

\[
\Phi = (x_1) \land (x_1 \lor \overline{x}_2) \land (x_3 \lor x_7) \land (x_2 \lor \overline{x}_1 \lor \overline{x}_5 \lor x_6) \land (x_4 \lor x_5 \lor x_6) \land (x_5 \lor x_6) \land (\overline{x}_6 \lor \overline{x}_4)
\]  

Figure 2.1 shows an example of a solving process for \( \Phi \). In this example, the SAT solver first performs \( bcp() \). The \( bcp() \) function determines that \( x_1 = 0 \) and \( x_2 = 0 \) to satisfy \( \Phi \). Next, the SAT solver selects \( x_4 \) for a decision and assigns it to 1. This decision requires \( x_6 = 0 \) and \( x_5 = 1 \) assignments in order to satisfy clause \( \overline{7} \) and \( \overline{6} \) of \( \Phi \), respectively. However, the partial assignment \( \{x_2 = 0, x_4 = 1, x_5 = 1, x_6 = 0\} \) cannot satisfy clause \( \overline{4} \). In other words, \( x_4 = 1 \) results in a conflict. As a result, the SAT solver undoes the decision by assigning \( x_4 \) to 0. Under the partial assignment \( \{x_1 = 0, x_2 = 0, x_4 = 0\} \), the SAT solver assigns \( x_5 \) to 1 and \( x_3 \) to 1 to satisfy the remaining clauses.

2.2.3 All-Solutions SAT

Many real-world problems require a SAT solver to return all possible satisfying assignments. All-solutions SAT, a variation of SAT, aims to find all satisfying assignments for a CNF formula.

A SAT solver can be extended to become an all-solutions SAT solver using \textit{blocking clauses}. In this technique, a SAT solver is incrementally called to find all satisfying assignments. At
Figure 2.1: A SAT Algorithm Example
each iteration, a clause which is the negation of the previous solution is added as a blocking
clause in order to prevent the SAT solver from finding the same solution again. The reason
for this is because the blocking clause forces the solver to backtrack once the same solution is
reached again during the search process. The next step is to call the SAT solver again to find
the next solution. When no more satisfying assignments are left, the SAT solver returns \texttt{UNSAT}
and the solving process is completed.

\textbf{Example 4} \textit{Given the Boolean formula} $\Phi$

\[ \Phi = (x_1 \lor y \lor s) \land (x_2 \lor y \lor \overline{s}) \land (x_2 \lor y \lor \overline{s}) \land (y) \quad (2.3) \]

\textit{All satisfying assignments of} $\Phi$ \textit{and their blocking clauses are given below:}

- \{ $s = 1, x_1 = 0, x_2 = 1, y = 1$ \} \textit{with the blocking clause} $(x_1 \lor x_2 \lor \overline{y} \lor \overline{s})$

- \{ $s = 1, x_1 = 1, x_2 = 1, y = 1$ \} \textit{with the blocking clause} $(x_1 \lor x_2 \lor \overline{y} \lor \overline{s})$

- \{ $s = 0, x_1 = 1, x_2 = 0, y = 1$ \} \textit{with the blocking clause} $(\overline{x}_1 \lor x_2 \lor \overline{y} \lor s)$

- \{ $s = 0, x_1 = 1, x_2 = 1, y = 1$ \} \textit{with the blocking clause} $(\overline{x}_1 \lor \overline{x}_2 \lor \overline{y} \lor s)$

\textbf{2.2.4 Partial Max-SAT}

Another variation of the SAT problem is the \textit{Maximum Satisfiability} (Max-SAT). Given an unsatisfiable CNF formula $\Phi$, the goal of a Max-SAT solver is to return an assignment that max-
imizes the number of satisfied clauses in $\Phi$. Max-SAT is a well-known NP-hard problem\footnote{29}.

In \textit{Partial Max-SAT}, the CNF formula $\Phi$ is partitioned as a set of \textit{hard} clauses which must be satisfied and a set of \textit{soft} clauses which may or may not be satisfied. The set of hard clauses and soft clauses are denoted as $\Phi_H$ and $\Phi_S$, respectively, (i.e., $\Phi = \Phi_H \land \Phi_S$). The goal of Partial Max-SAT is to find an assignment that satisfies all hard clauses while maximizing the number of satisfied soft clauses.

\textbf{Example 5} \textit{Consider the following Boolean formula} $\Phi$

\[ \Phi = [\overline{x}_1 \lor \overline{x}_2] \land [x_3] \land (x_1) \land (x_2) \land (\overline{x}_3 \lor \overline{x}_1) \quad (2.4) \]
Note that in $\Phi$, hard clauses are denoted in the $[ ]$ brackets and soft clauses are in $( )$ brackets. $\Phi$ is UNSAT since there is no assignment which satisfies $[[x_1 \lor x_2] \land (x_1) \land (x_2)]$. The first two clauses are hard clauses which must be satisfied, whereas the last three clauses are soft clauses which may or may not be satisfied. The assignment $\{x_1 = 0, x_2 = 1, x_3 = 1\}$ satisfies all hard clauses and two soft clauses which is the maximum number of satisfiable soft clauses.

Recent advances in Maximum Satisfiability solvers have demonstrated promising results in many industrial applications \cite{15}. Today, the most effective Max-SAT algorithm is based on solving the Max-SAT instance with unsatisfiable sub-formula identification and relaxation \cite{19,34,35}. Given an unsatisfiable CNF formula $\Phi$, the algorithm first identifies the unsatisfiable subformula. $\Phi$ is then enhanced by adding an extra variable to each clause in the UNSAT subformula. The clause is relaxed if it is satisfied due to the extra variable being assigned to 1. The extra variable is thus called a relaxation variable. If enough clauses are relaxed, $\Phi$ becomes satisfiable. Consequently, a solution to the Max-SAT instance $\Phi$ is a satisfying assignment that minimizes the number of relaxed clauses. To find such a satisfying assignment, a Max-SAT solver sets the number of relaxed clauses to 0 and increases it until the instance becomes satisfiable.

**Example 6** The following example describes the solving process of a Max-SAT solver on the UNSAT formula $\Phi$ presented in Example \cite{3}:

\[
\Phi = [\overline{x_1} \lor x_2] \land [x_3] \land (x_1) \land (x_2) \land (\overline{x_3} \lor \overline{x_1}) \tag{2.5}
\]

First, the solver identifies an unsatisfiable sub-formula in $\Phi$:

\[
[\overline{x_1} \lor x_2] \land (x_1) \land (x_2) \tag{2.6}
\]

Next, it adds relaxation variables $(c_1, c_2)$ to soft clauses in the sub-formula and keeps record of the number of relaxation variables assigned to 1. The relaxed instance of $\Phi$, $\Phi_{\text{relaxed}}$, has the following formula:

\[
\Phi_{\text{relaxed}} = [\overline{x_1} \lor x_2] \land [x_3] \land (x_1 \lor c_1) \land (x_2 \lor c_2) \land (\overline{x_3} \lor \overline{x_1}) \tag{2.7}
\]
As mentioned, the Max-SAT solver sets the number of relaxed clauses initially to 0 and increases it until $\Phi_{\text{relaxed}}$ is satisfiable.

When the number of relaxed clauses is equal to 0, $c_1$ and $c_2$ have to be equal to 0. In this case, the instance $\Phi_{\text{relaxed}}$ is exactly the same as $\Phi$ and is hence UNSAT. This means that the number of relaxed clauses has to be increased to make $\Phi_{\text{relaxed}}$ satisfiable.

When the number of relaxed clauses is increased to 1, \{x_1 = 0, x_2 = 1, x_3 = 1, c_1 = 1, c_2 = 0\} is a satisfying assignment of $\Phi_{\text{relaxed}}$. Note that the presented satisfying assignment only relaxes one clause since $c_1 = 1, c_2 = 0$. From the satisfying assignment of $\Phi_{\text{relaxed}}$, the solver can extract the solution \{x_1 = 0, x_2 = 1, x_3 = 1\} for the Max-SAT instance $\Phi$.

### 2.3 Verification and Debugging Overview

This section gives an overview of functional verification and design debugging. Functional verification is the task of ensuring that a logic netlist conforms to its specification. Once functional verification fails, it returns a counter-example indicating that the design is erroneous. A counter-example contains the initial state, the input and the expected output values that are necessary to reproduce the error. Although there are many types of design errors, this thesis focuses on functional errors; timing errors are not considered.

Design debugging starts after a functional verification engine produces a counter-example. The goal of design debugging is to locate all potentially erroneous components in the netlist. Potentially erroneous components, also referred to as suspects, are components that can be modified to fix the error demonstrated by the counter-example. Based on the description of the netlist (gate-level, RTL files, hierarchy, etc.), suspects can be gates, design modules/blocks, etc.

Figure 2.2 depicts a functional verification and design debugging cycle. Starting with a design and its specification, a verification engine determines whether the design corresponds to its specifications. Once functional verification fails, it produces a counter-example and passes it to design debugging. Next, design debugging finds all suspects with respect to the counter-example. Using these suspects, a designer or a verification engineer can devise rectifications to
correct the error. To complete the cycle, functional verification is performed again to ensure that no errors are left in the design.

Design debugging is still a predominantly manual task in the industry. This coupled with the growing complexity of modern chips and counter-examples has made design debugging a resource-intensive task consuming as much as 60% of the total verification effort. Thus, many methodologies have been proposed to automate design debugging and mitigate its cost. With great improvements in the last decade, SAT solvers have been deployed in many modern automated design debugging techniques.

### 2.4 Automated Design Debugging

The following notation is necessary for the discussion of SAT-based debugging techniques later in this section and throughout this thesis. Given an erroneous sequential circuit $C$, the symbols $X$, $Y$, and $S$ respectively represent the sets of primary inputs, primary outputs and state elements (flip flops). Let $L$ denote the set of all nodes (including nodes in $X, Y, S$). For each $Z \in \{L, X, Y, S\}$, the Boolean variable $z_i$ denotes the $i$th element of set $Z$.

When debugging with a counter-example spanning multiple clock-cycles, it is necessary to model the behavior of a sequential circuit. This can be accomplished using time-frame expansion, also known as Iterative Logic Array (ILA). In time-frame-expansion, the combinational component of $C$ is replicated (i.e., unrolled) $k$ times such that the next state of each time-frame is connected to the current state of the next time-frame. For any variable $z_i$ (or set $Z$), $z_i^t$ (or $Z^t$) denotes the corresponding variable (or set) in time-frame $t$. The behavior of $C$ during the $t$th clock-cycle is dictated by the transition relation predicate $T(S^t, S^{t+1}, X^t, Y^t)$ which can be extracted from $C$ and encoded in CNF using the Tseitin transformation with the auxiliary variables in $L^t$ (i.e., the logic gates).

In a modern design, the nodes are usually grouped into blocks. Each block consists of the synthesized gates corresponding to an RTL “block” such as an ‘if’ statement or an ‘always’ block in Verilog. Let $B = \{b_1, b_2, \ldots, b_{|B|}\}$ denote the set of all blocks, where each $b_i \subseteq L$. Note that the same node $l_i$ could belong to more than one block because of the hierarchical nature of
Figure 2.2: Typical debugging flow
RTL. The symbol $\text{out}(b_i)$ denotes the outputs of block $b_i$. In the unrolled circuit, $b_i^t$ denotes the set of nodes belonging to block $b_i$ in time-frame $t$. Consequently, $\text{out}(b_i^t)$ is the set of outputs of block $b_i$ at time-frame $t$.

**Example 7** Figure 2.3 shows an example of time-frame-expansion. The sequential circuit and its combinational component are displayed in Figure 2.3(a) and Figure 2.3(b), respectively. The corresponding time-frame expansion for two time-frames is illustrated in Figure 2.3(c).

Using the presented notation, we can now describe automated design debuggers and their underlying techniques. Given an erroneous sequential circuit $C$, a counter-example $\text{Trace}$, and an error cardinality $N$, an automated debugger aims to find all sets of $N$ blocks that can potentially be responsible for the faulty behavior associated with the counter-example. Each such set is referred as a solution of cardinality $N$. It is essential to note that an automated debugger uses an error cardinality $N$ to limit the number of blocks to be found in each solution. This is due to the fact that the complexity of a design debugging problem grows exponentially with regard to the number of errors [54].

### 2.4.1 Automated Design Debugging with Error-select Variables

Automated design debugging with error-select variables was introduced in [49] and since then has been improved and enhanced greatly [26, 27, 46, 47]. The debugging process consists of the following steps.

First, a set of error-select variables $E = \{e_1, \ldots, e_{|B|}\}$ is added to the circuit, where each $e_i$ is associated with a block $b_i$. The circuit is modified such that setting $e_i = 1$ disconnects the nodes in $\text{out}(b_i)$ from their fanins, making them free variables, while setting $e_i = 0$ does not modify the circuit behavior. Next, time-frame expansion is performed on this enhanced circuit such that $\text{out}(b_i^t)$ are controlled by the same error-select variable $e_i$ for all time-frames $t$. This allows the SAT solver to modify the outputs of block $b_i$ across all time-frames by setting $e_i = 1$ to “fix” any potential errors in $b_i$.

Then, constraints are applied to the initial state, primary inputs and primary outputs. These constraints ensure that given the initial state values $\Phi_{IS}$ and primary inputs values...
(a) Sequential circuit

(b) Combinational component

(c) Time-frame expansion for two time-frames

Figure 2.3: Time-frame expansion example
\( \Phi_X(X^1, \ldots, X^k) \) from the counter-example, the enhanced circuit produces the expected outputs \( \Phi_Y(Y^1, \ldots, Y^k) \), which are the outputs produced by the design specification. Finally, an error cardinality constraint \( \Phi_N(E) \) is added to enforce \( \Sigma_{i=1}^{\lvert B \rvert} e_i = N \). Overall, the design debugging problem is encoded as:

\[
\text{Debug} = \bigwedge_{t=1}^{k} T_{en}(S^t, S^{t+1}, X^t, Y^t, E) \land \Phi_{IS} \land \\
\Phi_X(X^1, \ldots, X^k) \land \Phi_Y(Y^1, \ldots, Y^k) \land \Phi_N(E) \tag{2.8}
\]

where \( T_{en}(S^t, S^{t+1}, X^t, Y^t, E) \) denotes the transition relation predicate of the enhanced circuit at time-frame \( t \).

Each assignment to \( E = \{e_1, \ldots, e_{\lvert B \rvert} \} \) satisfying \( \text{Debug}^\circ \) corresponds to a debugging solution and the SAT solver must find all such satisfying assignments to \( E \). This is normally done by iteratively blocking each satisfying assignment using a blocking clause and re-solving \( \text{Debug} \) until the problem becomes \text{UNSAT} (i.e., all-solutions SAT).

**Example 8** Consider the sequential circuit \( C \) presented in Figure 2.3(a). We are also given a two-cycle counter-example with initial state \( s_0^1 = 1 \), inputs \( \langle x_1, x_2, x_3, x_4 \rangle = \langle \langle 1, 1, 0, 1 \rangle, \langle 0, 0, 0, 1 \rangle \rangle \) and expected outputs \( \langle y_1, y_2 \rangle = \langle \langle 1, 1 \rangle, \langle 1, 1 \rangle \rangle \), demonstrating a mismatch in the second time-frame at the output \( y_2 \).

The corresponding design debugging formulation is illustrated in Figure 2.4. As shown, each block \( b_i \) is associated with an error-select variable \( e_i \). The initial-state/input/output constraints are shown in boxes. The constraint \( \Phi_N \) is omitted for brevity. For \( N = 1 \), \( \{b_4\} \) is returned by the automated design debugger as the only solution. \( b_4 \) is indeed the buggy block and could be corrected by turning gate \( g_4 \) into an \text{OR} gate.

### 2.4.2 Automated Design Debugging with Partial Max-SAT

This section discusses the Partial Max-SAT design debugging formulation proposed in [8]. This research is based on the unsatisfiable nature of design debugging problems and the great advancements of Max-SAT solvers in the recent past.

First, the circuit is unrolled \( k \) times using time-frame expansion. Constraints are then applied to the initial state, primary inputs and primary outputs to ensure that given the initial
state values $\Phi_{IS}$ and primary inputs values $\Phi_X(X^1, \ldots, X^k)$ from the counter-example, the
primary outputs yield their expected values $\Phi_Y(Y^1, \ldots, Y^k)$ given by the specifications. The
resulting Partial Max-SAT formulation to debug a sequential circuit $C$ is given as:

$$\text{Debug}_{\text{Max-SAT}} = \left( \bigwedge_{t=1}^{k} T(S^t, S^{t+1}, X^t, Y^t) \right) \wedge [\Phi_{IS}]$$

$$\wedge [\Phi_X(X^1, \ldots, X^k)] \wedge [\Phi_Y(Y^1, \ldots, Y^k)] \quad (2.9)$$

where $T(S^t, S^{t+1}, X^t, Y^t)$ denotes the transition relation predicate at time-frame $t$.

Since $\Phi_X$, $\Phi_Y$ and $\Phi_{IS}$ represent the constraints that must be respected, they are modeled as
hard clauses ($[ ]$ brackets). Meanwhile, $T_{en}$ is modeled as soft clauses ($()$ brackets). Because the
circuit is erroneous, it cannot produce the expected output values from the counter-example;
therefore, $\text{Debug}_{\text{Max-SAT}}$ is UNSAT$^{49}$.

For the unsatisfiable instance $\text{Debug}_{\text{Max-SAT}}$, a Partial Max-SAT solver returns a maximal
set of satisfied clauses. The complement of such a solution presents a set of clauses whose
removal satisfies $\text{Debug}_{\text{Max-SAT}}$. As such, these sets of clauses may correspond to the error.
In other words, the blocks that are associated with these clauses may be responsible for the
erroneous behavior exhibited by the counter-example.

As we are only interested in the complement of the solution of the Partial Max-SAT instance,
the phrase “Max-SAT solution” in the remainder of this thesis refers to the set complement of the Partial Max-SAT solution. The error cardinality $N$ is enforced by constraining the number of clauses in each Max-SAT solution. Like error-select techniques, blocking clauses are used to block previously found Max-SAT solutions and $\text{Debug}_\text{Max-SAT}$ is resolved until all solutions of cardinality $N$ are found.

2.5 Clock-gating for Low-power Design

The conflicting yet increasing demand for high performance and low-power in multi-functional chips has pushed techniques for power reduction to the forefront of VLSI design. Clock-gating is a circuit transformation where the clock input of a register no longer latches in a new value but instead holds its current value. This enables power-savings as unnecessary switching of gates can be avoided. The enable/control signal to stop the clock is computed in two phases. First, coarse-grained clock-gating is implemented by identifying the idle condition for major functional unit(s) of the design. Next, fine-grained clock-gating is inserted in register(s) by identifying idle conditions that are not visible at the architectural level.

For any register in the circuit, fine-grained clock-gating takes place in the following two situations: 1) the output of the register is not observed at the primary outputs, also known as an observability don't-care (ODC) condition, or 2) the output of the register retains the same logic value for two or more consecutive clock-cycles, a situation widely known as stability condition (STC). ODC is a well-known combinational technique, studied extensively and automatically extracted from steering logic such as multiplexers, tri-state buffers and enable states. Many techniques also have been developed in the industry to extract ODC conditions and achieve great power-savings. In contrast, STC is a sequential technique that must be extracted from the finite state machine of the circuit and thus is much more challenging. Moreover, computing the exact STC is a resource intensive process which is susceptible to design errors. For this reason, in this thesis we focus on debugging clock-gating implementations under STC conditions. For simplicity, we only consider designs with single clock domain. Below is the formal definition for the stability condition.
**Definition 3** A register is said to be stable when its output does not change for two consecutive clock-cycles (i.e., \( q^t = q^{t-1} \)).

**Example 9** Figure 2.5 shows an example of a stability-based clock-gating. Consider the register in Figure 2.5(a). In this case, register \( R \) receives a new data whenever \( en = 1 \). Otherwise, when \( en = 0 \), the output \( q \) of \( R \) remains unchanged. Because \( R \) retains its value from the previous time-frame, it does not perform any switching. As a result of the stability of \( R \), the designer can safely turn off its clock signal to reduce dynamic power dissipation on the clock network.

The clock-gated register using the STC condition from Figure 2.5(a) is shown in Figure 2.5(b). In order to turn off the clock signal of \( R \), the designer connects \( en \) and \( clk \) with an AND gate. When \( en = 0 \), the output of the AND gate stays 0 regardless of the \( clk \) value. Hence, no switching activity happens. Note that the output \( q \) of \( R \) of Figure 2.5(b) is equal with its counterpart in Figure 2.5(a) even when the clock signal is turned off.

Typically, there are three components in a clock-gated circuit: the existing state elements,
combinational circuitry from the original implementation before clock-gating is applied and the additional clock-gating circuitry that computes the new clock signals.

Example 10 Figure 2.6 displays a typical clock-gating design. The combinational circuitry is shown at the top and it computes next states \( \{d_1, \ldots, d_z\} \) and primary outputs \( \{y_1, \ldots, y_m\} \) from the set of primary inputs \( \{x_1, \ldots, x_n\} \) and current states \( \{q_1, \ldots, q_p\} \). The clock-gating circuitry is shown underneath and it computes clock signals of all registers from existing nodes of the combinational circuitry. Last, state elements include registers \( R_1 \) to \( R_p \) where \( p = |S| \). For a clock-gated register, a new enable signal is computed and connected with the clock input \( \text{clk} \) to create the new clock signal for the register.

2.6 Summary

This chapter provides relevant concepts that are necessary to understand this thesis. First, Boolean Satisfiability (SAT) was introduced along with its extensions, all-solutions SAT and Partial Max-SAT. Next, a review on verification and debugging was given. This was followed by the description of popular debugging techniques utilizing all-solutions SAT and Partial Max-
SAT. Finally, we discussed the design fundamentals behind clock-gating.
Chapter 3

Non-Solution Implications using Reverse Domination in a Modern SAT-based Debugging Environment

3.1 Introduction

This research uses the concepts of reverse domination and non-solution implications to improve all-solutions SAT-based design debugging. Block \( b \) is said to be a reverse dominator of block \( a \) if \( a \) dominates \( b \). It is shown that if block \( a \) is not part of any solution, then all its reverse dominators can also be ruled out as non-solutions, that are blocks that cannot be modified in any way to correct the counter-example. Based on this idea, we tailor a SAT solver to leverage reverse dominators for performing non-solution implications. We present a new decision heuristic where error-select \([3,49]\) variables are decided upon first. This allows us to learn blocks that are not part of any solution early in the solving process. The existing restart mechanism in the SAT solver is also enhanced for non-solution learning. Therefore, non-solution implications can be used much more effectively. After learning non-solution blocks, we prune them from the debugging search space to reduce the complexity of the problem. Experimental results demonstrate the practicality of the proposed framework.
3.2 Dominance Relationships and Previous Work

This section describes dominance relationships in a sequential circuit and explains previous work that utilizes dominance relationships in design debugging. In a sequential circuit $C$, a node $u$ is said to dominate a set of nodes $V$ if every path from a node $v \in V$ to a primary output contains $u$. If $V$ only contains a single node, $u$ is called a single-vertex dominator; otherwise, it is called a multiple-vertex dominator. From the dominance relationships between nodes, we can describe block dominance as follows.

Block $b_j$ is said to dominate block $b_i$ if every path from a node in $\text{out}(b_i)$ to a primary output contains a node in $b_j$. The notation $b_j \triangleright_D b_i$ indicates that $b_j$ dominates $b_i$, where $\triangleright_D$ is referred to as the block dominance relation. Furthermore, the set $D(b_i) = \{b_j | b_j \triangleright_D b_i\}$ consists of blocks that dominate $b_i$.

**Example 11** Consider the sequential circuit in Figure 3.1. Block $b_3$ dominates block $b_1$ while no other blocks dominate any other blocks. This is because every path from $\text{out}(b_1)$ has to pass through gate $g_3$ of $b_3$ to reach the primary outputs $y_1, y_2$. However, block $b_4$ does not dominate any blocks because there exist paths from $b_1, b_2$ and $b_3$ to primary output $y_2$ that do not pass through $b_4$.

![Figure 3.1: Reverse Dominator Example](image)

The authors of [31] discuss why existing methods for computing single and multiple-vertex dominators are not applicable in a design debugging setting, and present a fix-point algorithm.
for computing the block dominance relation $D$. The run-time of their algorithm is $O(c \cdot |B| \cdot |Edges|)$, where $|B|$ is the number of blocks, $|Edges|$ is the number of edges in $C$ and $c$ is the loop-connectedness of $C$.

Furthermore, \[31\] proves that given a solution $\{b_{i_1}, ..., b_{i_N}\}$ of $\text{Debug}$ (see Equation \[2.9\]), if $\bigwedge_{n=1}^{N}(b_{j_n} \, D \, b_{i_n})$, then $\{b_{j_1}, ..., b_{j_N}\}$ is also a solution. This allows leveraging the block dominance relation $D$ to perform solution implications, which significantly reduces the number of SAT calls and speeds up the debugging process.

### 3.3 Non-Solution Implications using Reverse Domination

In this section, we first define reverse dominators and non-solution blocks. Next, we prove that reverse dominators can be leveraged to perform non-solution implications, given an original non-solution block. In the following section, we present a branching heuristic which enables the SAT solver to find original non-solutions much faster, leading to earlier non-solution implications.

**Definition 4** A block $b_i$ is a reverse dominator of block $b_j$, denoted as $b_iD^{-1}b_j$, if and only if $b_jD b_i$.

Clearly, the reverse block dominance relation $D^{-1}$ is completely determined by $D$, which can be computed using the algorithm in \[31\]. The set $D^{-1}(b_j) = \{b_i | b_iD^{-1}b_j\}$ consists of reverse dominators of $b_j$ (i.e., the blocks that $b_j$ dominates).

**Definition 5** Given an erroneous design $C$, a counter-example of length $k$ along with the corresponding expected outputs and an error cardinality $N$, $b_i$ is a non-solution block if and only if $\text{Debug} \wedge e_i$ is UNSAT.

In other terms, a non-solution block cannot be part of any solution of cardinality $N$. If $N = 1$, then a non-solution block cannot be modified in any way to correct the erroneous behavior in the counter-example trace. We will prove that reverse dominators of non-solution blocks are also non-solution blocks. In order to do so, we need to prove the following lemma.
Lemma 1 Given an erroneous design $C$, a counter-example of length $k$ along with the corresponding expected outputs and an error cardinality $N$, we have:

$$(\text{Debug} \land e_i \text{ is SAT}) \land b_j \text{Db}_i \Rightarrow (\text{Debug} \land e_j \text{ is SAT})$$

Proof: Let $\pi$ denote the satisfying assignment of $(\text{Debug} \land e_i)$. Assuming that $b_j \text{Db}_i$, we will construct an assignment $\pi'$ satisfying $(\text{Debug} \land e_j)$.

We first construct $\pi'(e)$. Let the set of error-select variables assigned to 1 in $\pi(e)$ be

$$\{e_i, e_{\sigma_1}, \ldots, e_{\sigma_{N-1}}\},$$

where $\{\sigma_1, \ldots, \sigma_{N-1}\} \subseteq [1, |B|] - \{i\}$.

1. If $j \not\in \{\sigma_1, \ldots, \sigma_{N-1}\}$, we let the set of error-select variables assigned to 1 in $\pi'(e)$ be

$$\{e_j, e_{\sigma_1}, \ldots, e_{\sigma_{N-1}}\}.$$

2. If $j \in \{\sigma_1, \ldots, \sigma_{N-1}\}$, we let the set of error-select variables assigned to 1 in $\pi'(e)$ be

$$\{e_i, e_{\sigma_1}, \ldots, e_{\sigma_{N-1}}\}.$$

In both cases, the number of error-select variables assigned to 1 in $\pi'(e)$ is $N$, satisfying $\Phi_N$.

Since $b_j \text{Db}_i$, any path from out$(b_i)$ to a primary output must pass through out$(b_j)$. This makes it possible to partition the unrolled enhanced circuit described in Subsection 2.4.2 into two parts: Let $I$ refer to the sub-circuit in the fan-out cone of out$(b_i)$ (that fans out to out$(b_j)$) and let $J$ refer to the rest of the circuit (excluding error-select variables). In Debug $\land e_j$, clearly $\pi'(e_j) = 1$ in both cases shown above, effectively disconnecting out$(b_j)$ from its fanins. As such, out$(b_i)$ is disconnected from the primary outputs. A node is said dangling if there is no path from such node to the primary outputs. Hence, out$(b_i)$ becomes dangling logic. This means that $I$ is dangling (although $J$ can fan-out to $I$). Since there are no external constraints on $I$, $\pi'(I)$ can be computed by simply “propagating” whatever $\pi'(out(b_i))$ and $\pi'(J)$ are into $I$ (using gate propagation, which is effectively Boolean constraint propagation in CNF). Hence, what remains is to construct $\pi'(J)$.

Note that every error-select variable $e_k$ other than $e_i$ or $e_j$ is assigned to the same value in $\pi$ and $\pi'$, as shown in both cases above. Furthermore, since $\pi'(e_j) = 1$, we are free to set $\pi'(out(b_j)) = \pi(out(b_j))$. In addition, recall that out$(b_i)$ has no effect on $J$ since $I$ is dangling. As such, since $\pi'(e_k) = \pi(e_k)$ for all other $e_k$, for all nodes $v \in out(b_k) \cap J$, we can simply set
\( \pi'(v) = \pi(v) \). As a result, \( \pi'(J) = \pi(J) \). Since \( \pi(J) \) satisfies all the constraints in \text{Debug}, so does \( \pi'(J) \). Finally, since \( \pi'(e_j) = 1 \), \( \pi' \) satisfies \text{Debug} \( \land e_j \).

The following theorem proves that reverse dominators can be used to perform non-solution implications.

**Theorem 1** Given an erroneous design \( C \), a counter-example of length \( k \) along with the corresponding expected outputs and an error cardinality \( N \), if \( b_j \) is a non-solution block of \text{Debug} and \( b_i D^{-1} b_j \), then \( b_i \) is also a non-solution block of \text{Debug}.

**Proof:** To clarify the presentation, let us define the predicates \( \Phi_i \) and \( \Phi_j \), as follows:

\[
\Phi_i = \text{Debug} \land e_i \text{ is SAT} \quad \Phi_j = \text{Debug} \land e_j \text{ is SAT}
\]

Using Lemma \( \Box \) we have:

\[
(\Phi_i \land b_j D b_i) \Rightarrow \Phi_j \\
\Leftrightarrow \neg \Phi_i \lor \neg (b_j D b_i) \lor \Phi_j \\
\Leftrightarrow \neg \Phi_i \Leftrightarrow (b_j D b_i \land \neg \Phi_j) \\
\Leftrightarrow (b_i D^{-1} b_j \land \neg \Phi_j) \Rightarrow \neg \Phi_i
\]

\( \Box \)

**Example 12** Consider the sequential circuit in Figure 3.1 and the debugging problem presented in Figure 3.2. We know that block \( b_3 \) is a dominator of block \( b_1 \) from Example 11. If \( b_3 \) is known to be a non-solution, using Theorem 1, we know that \( b_1 \) is also a non-solution. We can therefore automatically add the clause \( (\overline{e_1}) \) to prune the search space of \text{Debug}.

It is necessary to learn some non-solution blocks before Theorem 1 can be used. An all-solutions SAT solver returns when a solution is found. This makes it possible to imply its dominating solutions\( ^{-31} \) without modifying the SAT solver. However, unlike solution blocks, information on non-solution blocks cannot be extracted from the satisfying assignment returned by the all-solutions SAT solver.
One way to identify non-solution blocks is by using learned clauses. Learned clauses are clauses generated by a SAT solver during its conflict analysis phase. These clauses keep a record of partial assignments that result in conflicts. Adding these clauses guarantees that if the SAT solver ever tries to make the same partial assignments in the future, these clauses will immediately become unsatisfied and will force the SAT solver to backtrack. The following example shows how a modern SAT solver constructs a learned clause.

**Example 13** Consider the solving process presented in Example 3 on page 16. A modern SAT solver, after learning that $x_4 = 1$ as the first decision resulting in an unsatisfied clause, will add a clause $(\overline{x_4})$ to prevent $x_4$ from being assigned to 1 again.

Non-solution blocks can be identified by watching learned clauses of the form $(\overline{e_i})$. However, watching these clauses is not practical because the SAT solver rarely learns such unit clauses. Instead, learned clauses are much more complex and usually involve many other variables along with error-select variables.

Another way to realize that a block is a non-solution is to examine the implied assignments of Boolean constraint propagation ($\text{bcp}()$) after each solver restart. A restart clears all partial assignments including all the decisions. With no decision, the implied assignments of ($\text{bcp}()$) after each solver restart are implications of the instance. In the following example, we show...
Chapter 3. Non-Solution Implications using Reverse Domination

how to extract these implications.

**Example 14** Consider the scenario where the SAT solver, given the Boolean formula $\Phi$ in Example 3, restarts after it adds the learned clause ($\pi_1$). In this case, performing $\texttt{bcp()}$ after such a restart learns that $(x_1 = 0)$, $(x_2 = 0)$ and $(x_4 = 0)$ are implications of $\Phi$.

Using this method, if some $e_i = 0$ by Boolean constraint propagation given no decision has been made, then the solver has “learned” that $b_i$ is a non-solution block. However, from our experience, using a generic SAT solver, virtually all non-solution blocks are learned during the last solver restart in the last call of the procedure (after all solutions have been found) leaving little room for improvements using non-solution implications.

The following section shows how to modify the decision heuristic of the SAT solver to overcome this problem.

### 3.4 Decision Heuristic for Early Non-Solution Learning

In this section, we describe a new decision heuristic for design debugging, where error-select variables are decided upon first. This allows the early learning (and simple detection) of non-solutions, making non-solution implications using reverse dominators useful.

#### 3.4.1 Decision Heuristic

A decision tree in a SAT solver gives the order in which variables are decided upon. Implied assignments due to $\texttt{bcp()}$ are not part of a decision tree. In a decision tree, leaf nodes represent the result ($\texttt{SAT}$/$\texttt{UNSAT}$) from the partial assignment extended from the decision made, whereas other nodes represent unassigned variables selected for decisions.

**Example 15** Consider the Boolean formula $\Phi$ in Example 3. Figure 3.3 shows an example of a decision tree for $\Phi$. Note that implied assignments due to $\texttt{bcp()}$ are not part of a decision tree.

The first motivation for assigning the error-select variables early in the decision tree relates to their importance in a design debugging SAT instance. Error-select variables are the careset...
variables of a design debugging problem. Careset variables are variables that must be assigned to witness a satisfying assignment. To elaborate, a variable $v$ is a careset variable if and only if there is no partial assignment $A$ such that $A \cup \{(v = 0)\}$ and $A \cup \{(v = 1)\}$ are both satisfying assignments. For a SAT instance, assigning careset variables first is preferable because a complete assignment on careset variables is a “gateway” to a solution. Example 16 shows careset variables in a SAT instance.

**Example 16** Consider the Boolean formula $\Phi$ in Example 3. For this instance, $x_1, x_2, x_4$ are the careset variables because $x_1, x_2, x_4$ have to be assigned to 0 to witness a satisfying assignment. For each other variable $x_i$, there exists a partial assignment $A$ such that $A \cup \{(x_i = 0)\}$ and $A \cup \{(x_i = 1)\}$ are both satisfying assignments. The following list gives the partial assignment for each of those variables.

- $x_3$: $(x_1 = 0, x_2 = 0, x_4 = 0, x_5 = 1, x_7 = 1)$
Chapter 3. Non-Solution Implications using Reverse Domination

- \(x_5\): \(\{x_1 = 0, x_2 = 0, x_4 = 0, x_6 = 1, x_3 = 1\}\)
- \(x_6\): \(\{x_1 = 0, x_2 = 0, x_4 = 0, x_5 = 1, x_3 = 1\}\)
- \(x_7\): \(\{x_1 = 0, x_2 = 0, x_4 = 0, x_5 = 1, x_3 = 1\}\)

Error-selects are careset variables in a design debugging problem because there is no partial assignment \(A\) such that \(A \cup \{(e_i = 0)\}\) and \(A \cup \{(e_i = 1)\}\) are both satisfying assignments. The two assignments \(A \cup \{(e_i = 0)\}\) and \(A \cup \{(e_i = 1)\}\) cannot both satisfy the error cardinality constraint \(N\) because in that case we would have two debugging solutions that have different numbers of solutions. As error-selects variables belong to the careset, they should be assigned first in a decision tree.

The second motivation for assigning the error-select variables early in the decision tree is their impact on other variables in the SAT solving process. For instance, when \(e_i = 1\), the internal nodes of block \(b_i\) become dangling, and therefore they become don’t-cares. As such, assigning the nodes in \(b_i\), as well as their fan-outs, is useless if \(e_i\) is later assigned to 1. On the other hand, when \(e_i\) is assigned to 0, block \(b_i\) is connected with its fan-out blocks. This allows the SAT solver to compute values of nodes in these fan-out blocks using Boolean constraint propagation.

The third, and most important, reason for assigning the error-select variables early is that it allows the solver to learn non-solution blocks much faster. This in turn enables non-solution implications due to reverse dominance in order to prune the SAT search space earlier and therefore more effectively. Subsection \[3.4.2\] discusses how to detect learned non-solutions using our decision heuristic.

As a result, we force the SAT solver to first decide on all error-select variables \((\langle e_1, e_2, \ldots, e_{|E|} \rangle)\). Furthermore, we force the solver to always assign error-select variables that are decided \((i.e.,\) not forced due to \(\text{bcp}()\)) to 1 before trying to set them to 0. The reason for doing this is to learn non-solutions, and is explained in detail in Subsection \[3.4.2\]. Once all the error-select variables are assigned, the solver uses the standard decision heuristics \((\text{e.g., VSIDS}^{−39})\) for the remaining variables.
3.4.2 Detecting Non-Solution Blocks

To simplify the presentation of this subsection, let us assume without loss of generality that the variable at the root of the decision tree is $e_1$. According to our decision heuristic explained in the previous section, the SAT solver first assigns $e_1 = 1$. If the solver later switches to $e_1 = 0$ without finding a satisfying assignment under $e_1 = 1$, this means that $e_1 = 1$ cannot be extended to a satisfying assignment. Hence, $e_1 = 0$ is true for all satisfying assignments (if any exist). In other words, $(\overline{e_1})$ has been learned and $b_1$ is a non-solution block.

This observation is not applicable to all non-root variables in the decision tree. Consider the decision tree in Figure 3.4. Although $b_1$ can be learned as a non-solution block from the switching of $e_1$, the switching from $e_2 = 1$ to $e_2 = 0$ without finding a satisfying assignment does not imply that $b_2$ is a non-solution block. In this case, $e_2 = 1$ cannot be extended to a satisfying assignment under the condition that $v = 1$. The partial assignment $\langle v = 0, e_2 = 1 \rangle$ is still unexplored and does not necessarily result in UNSAT. However, it is possible to learn about non-root variables in some circumstances, as shown by Lemma 2.

**Lemma 2** Using the decision heuristic given in Subsection 3.4.1, until a satisfying assignment is found, all the error-select variables set to 0 along the right-most path of the decision tree correspond to non-solution blocks.

**Proof:** Assume that the error-select variables are decided in the order of $\langle e_1, \ldots, e_{|B|} \rangle$. Recall that our decision heuristic forces the solver to first set each error-select variable to 1 before trying to set it to 0. Also assume that $e_1 = 0, \ldots, e_i = 0$ have been set along the right-most path of the decision tree and no satisfying assignment has been found yet. Then by construction, all other assignments to $e_1, \ldots, e_i$ have been examined and setting any of them to 1 cannot be extended to a satisfying assignment. This scenario is shown in Figure 3.5.

As $e_1 = 1$ cannot be extended to a satisfying assignment, $\text{Debug} \land (e_1)$ is UNSAT. This indicates that $(\overline{e_1})$ is an implication of $\text{Debug}$.

Since $e_1 = 0 \cup e_2 = 1$ cannot be extended to a satisfying assignment, $\text{Debug} \land (\overline{e_1}) \land (e_2)$ is UNSAT. However because $(\overline{e_1})$ is an implication of $\text{Debug}$, $\text{Debug} \land (\overline{e_1}) \land (e_2)$ is equivalent with $\text{Debug} \land (e_2)$. This indicates that $\text{Debug} \land (e_2)$ is UNSAT and $(\overline{e_2})$ is an implication of $\text{Debug}$.
Using the same argument, we can prove that $\text{Debug} \land (e_3), \ldots, \text{Debug} \land (e_i)$ are UNSAT and $(\overline{e_1}), \ldots, (\overline{e_i})$ are implications of $\text{Debug}$.

By Definition 5 each of $\text{Debug} \land (e_1), \ldots, \text{Debug} \land (e_i)$ is UNSAT implies that each of $b_1, \ldots, b_i$ is a non-solution block.

Using Lemma 2 as soon as the SAT solver switches from $e_i = 1$ to $e_i = 0$, as long as all its ancestors in the decision tree are assigned to 0 and no satisfying assignment has been found yet, we can be sure that $b_i$ is a non-solution block. Using this, we can imply that every block $b_j \in D^{-1}(b_i)$ is also a non-solution, by Theorem 11.

The following subsection shows how we utilize non-solution block information to prune the search space of the problem.
3.4.3 Non-Solution Block Propagation

This section demonstrates how we make use of non-solution block \( b_i \) learned using our decision heuristic or non-solution implications in Theorem 11. Recall that our debugging technique makes several calls to the SAT solver. Hence, indicating that \( b_i \) is a non-solution block prevents the SAT solver from unnecessarily assigning \( e_i \) to 1 in later calls. One simple way of achieving this is to remove \( e_i \) from the instance. However, this method requires finding all appearances of \( e_i \) in \text{Debug}. With non-solution implications allowing many blocks to be identified as non-solutions at once, this will add significant overhead.

To overcome this issue, we add the unit clause \((\overline{e_i})\) to \text{Debug} instead of removing \( e_i \) from
it. This clause prevents the SAT solver from assigning \( e_i \) to 1 as it will immediately become unsatisfied. This unit clause is added right after \( b_i \) is learned as non-solution during the solving process. By adding these clauses on-the-fly, we do prevent the SAT solver from assigning \( e_i \) to 1 not only in later calls but also after each solver restart during the current call. Finally, adding \((\overline{e_i})\) only requires constant time and thus does not create noticeable overhead.

### 3.4.4 Restart Heuristics

Modern SAT solvers have periodic **restarts**, usually occurring after a certain number of conflicts. A restart clears assignments of all variables (including all decisions) while keeping the learned clauses. These learned clauses combined with VSIDS guarantee that the solver will not repeat the previous analysis following a restart. As our decision heuristic alternates VSIDS, the existing decision heuristic in modern SAT solvers, there is no guarantee that previous analysis will not be repeated after a restart.

In this research, we modify the existing restart mechanism to enhance the decision heuristic for non-solution detection. If no non-solutions have been learned during a solver restart, we generate a random number \( r \) \((1 \leq r \leq |B|)\), and move all the error-select variables above level \( r \) in the decision tree under those that are below level \( r \). This ensures that the SAT solver will branch on different error-select variables after a restart. Additionally, this prevents the SAT solver from again engaging in parts of the search space where it is hard to learn new non-solutions.

### 3.4.5 Overall Modified SAT Algorithm

Algorithm 3 presents the pseudo-code of our modified SAT solver. All unassigned variables are already assumed to have been assigned **priority** values which set their order in the decision tree. Our algorithm assigns error-select variables with very large priority values in order to guarantee that they will be at the top of the decision tree (line 10). The algorithm first runs \texttt{bcp()}\ to detect instances that are trivially **SAT** or **UNSAT**.

Line 4 shows that each restart begins a new iteration of the SAT engine. At each iteration, all unassigned variables are stored in a Maxheap \[10\] \texttt{heap} in which the root is the next decision
variable (line 5). As mentioned in Subsection 3.4.4, if the solver has not learned any new non-solution blocks during a solver restart, then it has engaged in a search space where it is hard to learn new non-solutions. In order to direct the solver towards parts of the search space where it is easier to learn non-solutions, we use the variable learned. If learned = false, the SAT engine has spent a full iteration under $e_i = 1$ without learning any new non-solutions or finding a solution (line 6). In this case, the heuristic is applied to reorder the heap such that the SAT engine will not pick $e_i$ first in the next iteration. A random number $r$ ($1 \leq r \leq |B|$) is generated, and all the error-select variables above level $r$ in the heap are moved below the ones under level $r$ (line 7-line 8). At the beginning of each iteration, learned is set to false and the variable numConf, storing the number of conflicts, is set to 0.

In this algorithm, the variable $e_i$ represents the unassigned error-select variable with the highest priority, whereas next represents the next decision variable. On line 19, next is popped from the heap. If next is an error-select line, then it must be first assigned to 1 (line 20), otherwise the function polarity() decides the polarity of next using heuristics such as VSIDS (line 21). After each decision, bcp() is called to perform Boolean constraint propagation (line 27). Note that next may not be $e_i$ as the SAT engine can make many decisions after the decision $e_i = 1$ to learn that it cannot be extended to a satisfying assignment. Later, if $e_i$ is assigned to 0, block $b_i$ is learned as a non-solution block as presented in Lemma 2. As a result, each $b_j$ that is dominated by $b_i$ is also learned as a non-solution block and the unit clause ($\overline{e_j}$) is added (line 25). After $b_i$ is learned as a non-solution, $e_i$ is updated so that new non-solutions can be learned (line 24). Other functions of the SAT engine such as bcp() and analyze_conflict() are not modified.

### 3.5 Experimental Results

This section presents the experimental results for the proposed framework on industrial design debugging problems. The presented techniques are implemented on top of a state-of-the-art SAT-based debugger with a Verilog front-end to allow for RTL diagnosis. As all
Algorithm 3: SAT Solver for Design Debugging

```
input: CNF Debug, Dominator relation D, set E

1 foreach $e_i \in E$ do  Priority($e_i$) ← ∞;
2 learned ← true;
3 result ← bcp();
4 while result ≠ (SAT/UNSAT) do
5    heap ← buildHeap (Priority);  
6    if learned = false then
7       r ← genRandom (1, |B|);  
8       heap ← reorderHeap (r);  
9    end
10 learned ← false;
11 numConf ← 0;
12 $e_i$ ← heap.firstErrorSelect();
13 while numConf < maxConf do
14    if result = (SAT/UNSAT) then return;
15    if result = Conflict then
16       numConf ++;
17       resolveConflict();
18    end
19    next ← heap.pop();
20    if next ∈ E then  next.assign (1);
21    else next.assign (polarity ());
22    if $e_i$.value () = 0 then
23       foreach $e_j \in D(b_i)$ do  Debug ← Debug ∧ ($\overline{e_j}$);
24       $e_i$ ← nextErrorSelect ();
25       learned ← true;
26    end
27    result ← bcp();
28 end
29 end
```
dominance and reverse dominance relationships are computed in this debugging framework \cite{31}, we do not modify the front-end RTL diagnosis. In other words, this work does not add overhead to the existing framework. Instead, we only tailor the debugger’s back-end SAT solver, MiniSat 2.2.0 \cite{14}, to leverage reverse dominators for performing non-solution implications.

Eight industrial Verilog designs from OpenCores \cite{41} and three commercial designs provided by our industrial partners are used in our experiments. For each design, several debugging instances are generated by injecting different designer mistakes such as wrong state transitions, incorrect operators or incorrect module instantiations. The erroneous designs are then verified using industrial verification tools. A failure is detected and a counter-example is recorded and passed to the debugger. Experiments are conducted with three different versions of the SAT solver, the original MiniSat (\texttt{dbg-dom}), our enhanced version without the randomization heuristic after restarts (\texttt{dbg-dom+rev}), and our enhanced version with error-select variable order randomization at restarts (\texttt{dbg-dom+rev+RR}). Note that solution implications are applied in all experiments.

Table 3.1 shows the information of all debugging problems in our experiments. The first column gives the instance name. The next four columns respectively show the length of the counter example $k$, the number of nodes $|I|$ in $C$, the number of blocks $|B|$, and the total number of potentially erroneous blocks (i.e., solutions), $\# \text{sols}$.

Table 3.2 and Table 3.3 show the results of our experiments on OpenCore and industrial instances, respectively. The first column gives the instance name. Column \texttt{dbg-dom} gives the total run-time of the original MiniSat 2.2.0. Columns three (\textit{time}), four (\# impl non-sols) and five (\textit{imprv}) under \texttt{dbg-dom+rev} respectively give the total run-time of \texttt{dbg-dom+rev}, the number of implied non-solutions and the speed-up compared with \texttt{dbg-dom}. The following three columns show the same numbers for \texttt{dbg-dom+rev+RR}.

The average speed-up in total SAT run-time compared to \texttt{dbg-dom} is 1.55x for \texttt{dbg-dom+rev} and 1.72x for \texttt{dbg-dom+rev+RR}, showing significant improvement. This is equivalent with 42% reduction in solving run-time. The difference between \texttt{dbg-dom+rev} and \texttt{dbg-dom+rev+RR} is small; however, the latter is more consistent and shows improvement over \texttt{dbg-dom} in all cases. Particularly, for vga2 and memctrl1 instances, \texttt{dbg-dom+rev}
does not outperform `dbg-dom`, `dbg-dom+rev+RR` is able to achieve speedup of 1.6x and 1.8x, respectively. In some instances, such as for rsdecoder1, both versions of our solver terminate, while the original solver times out. In rare cases, such as ucrc_par and mem_ctrl2, no non-solutions are implied. However, our solvers still show significant speed-ups over `dbg-dom` due to our branching scheme which decides error-select variables first. Nevertheless, the performance gain from rearranging error-select variables is small and inconsistent. Finally, Figure 3.9 plots the SAT run-times of our solvers `dbg-dom+rev` and `dbg-dom+rev+RR` versus those of `dbg-dom` on a logarithmic scale, demonstrating the effectiveness of our method.

Figure 3.6 plots the ratio of implied non-solutions to all non-solutions using `dbg-dom+rev+RR` for each instance, sorted in increasing order. In this figure, the y-axis depicts the ratio of the implied non-solutions to all non-solutions while the x-axis shows the instances in increasing order of the ratios. It can be seen that up to 75% of all non-solutions blocks are implied and 25% of all non-solutions blocked are implied on average. As a result, the search-space of the SAT solver is pruned, resulting in significant speed-ups.

Figure 3.7 plots the number of solutions versus run-time for `dbg-dom` and `dbg-dom+rev+RR`
### Table 3.1: Instance Information

<table>
<thead>
<tr>
<th>Name</th>
<th>Trace length</th>
<th># nodes</th>
<th># blocks</th>
<th># sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsdecoder1</td>
<td>112</td>
<td>13543</td>
<td>2044</td>
<td>430</td>
</tr>
<tr>
<td>rsdecoder2</td>
<td>112</td>
<td>13564</td>
<td>2044</td>
<td>396</td>
</tr>
<tr>
<td>usb_func1</td>
<td>32</td>
<td>35158</td>
<td>3425</td>
<td>422</td>
</tr>
<tr>
<td>usb_func2</td>
<td>53</td>
<td>35350</td>
<td>4201</td>
<td>576</td>
</tr>
<tr>
<td>wb_dma1</td>
<td>35</td>
<td>191386</td>
<td>7896</td>
<td>468</td>
</tr>
<tr>
<td>wb_dma2</td>
<td>7</td>
<td>299838</td>
<td>8460</td>
<td>205</td>
</tr>
<tr>
<td>wb_dma3</td>
<td>28</td>
<td>299862</td>
<td>8836</td>
<td>526</td>
</tr>
<tr>
<td>vga1</td>
<td>423</td>
<td>89412</td>
<td>1593</td>
<td>128</td>
</tr>
<tr>
<td>vga2</td>
<td>423</td>
<td>89402</td>
<td>1741</td>
<td>84</td>
</tr>
<tr>
<td>ucrc_par</td>
<td>155</td>
<td>1056</td>
<td>63</td>
<td>20</td>
</tr>
<tr>
<td>mem_ctrl1</td>
<td>581</td>
<td>48006</td>
<td>3355</td>
<td>23</td>
</tr>
<tr>
<td>mem_ctrl2</td>
<td>1180</td>
<td>48006</td>
<td>3355</td>
<td>9</td>
</tr>
<tr>
<td>mips7891</td>
<td>153</td>
<td>30711</td>
<td>953</td>
<td>49</td>
</tr>
<tr>
<td>opensparc_ddr21</td>
<td>29</td>
<td>58399</td>
<td>2792</td>
<td>373</td>
</tr>
<tr>
<td>opensparc_ddr22</td>
<td>27</td>
<td>64915</td>
<td>2791</td>
<td>509</td>
</tr>
<tr>
<td>design1-1</td>
<td>71</td>
<td>499325</td>
<td>20204</td>
<td>69</td>
</tr>
<tr>
<td>design1-2</td>
<td>26329</td>
<td>499705</td>
<td>20211</td>
<td>117</td>
</tr>
<tr>
<td>design1-3</td>
<td>5343</td>
<td>499696</td>
<td>20209</td>
<td>120</td>
</tr>
<tr>
<td>design1-4</td>
<td>467</td>
<td>499705</td>
<td>20211</td>
<td>150</td>
</tr>
<tr>
<td>design1-5</td>
<td>177</td>
<td>499705</td>
<td>20211</td>
<td>98</td>
</tr>
<tr>
<td>design2-1</td>
<td>26</td>
<td>45632</td>
<td>5507</td>
<td>61</td>
</tr>
<tr>
<td>design2-2</td>
<td>5</td>
<td>203706</td>
<td>7416</td>
<td>50</td>
</tr>
<tr>
<td>design2-3</td>
<td>20</td>
<td>2082</td>
<td>185</td>
<td>62</td>
</tr>
<tr>
<td>design3-1</td>
<td>56</td>
<td>5454</td>
<td>495</td>
<td>129</td>
</tr>
<tr>
<td>design3-2</td>
<td>144</td>
<td>2333</td>
<td>144</td>
<td>28</td>
</tr>
</tbody>
</table>
Table 3.2: Design Debugging with Non-Solution Implications and Reverse Dominance Results

<table>
<thead>
<tr>
<th>Instance</th>
<th>dbg-dom</th>
<th></th>
<th></th>
<th></th>
<th>dbg-dom+rev</th>
<th></th>
<th></th>
<th>dbg-dom+rev+RR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>time</td>
<td># impl</td>
<td>imprv</td>
<td>time</td>
<td># impl</td>
<td>imprv</td>
<td>time</td>
<td># impl</td>
</tr>
<tr>
<td></td>
<td>(s)</td>
<td>(s)</td>
<td>non-sols</td>
<td>(x)</td>
<td>(s)</td>
<td>non-sols</td>
<td>(x)</td>
<td>(s)</td>
<td>non-sols</td>
</tr>
<tr>
<td>rsdecoder1</td>
<td>T/O</td>
<td>6955.90</td>
<td>1192</td>
<td>∞</td>
<td>5502.49</td>
<td>1211</td>
<td>∞</td>
<td>19.11</td>
<td>798</td>
</tr>
<tr>
<td>rsdecoder2</td>
<td>33.35</td>
<td>20.46</td>
<td>941</td>
<td>1.6x</td>
<td>19.11</td>
<td>798</td>
<td>1.7x</td>
<td>110.65</td>
<td>1487</td>
</tr>
<tr>
<td>usb_funct1</td>
<td>53.17</td>
<td>45.46</td>
<td>631</td>
<td>1.2x</td>
<td>48.15</td>
<td>941</td>
<td>1.1x</td>
<td>48.15</td>
<td>941</td>
</tr>
<tr>
<td>usb_funct2</td>
<td>134.46</td>
<td>117.83</td>
<td>1167</td>
<td>1.1x</td>
<td>110.65</td>
<td>1487</td>
<td>1.2x</td>
<td>110.65</td>
<td>1487</td>
</tr>
<tr>
<td>wb_dma1</td>
<td>123.89</td>
<td>97.26</td>
<td>2100</td>
<td>1.3x</td>
<td>81.34</td>
<td>1823</td>
<td>1.5x</td>
<td>81.34</td>
<td>1823</td>
</tr>
<tr>
<td>wb_dma2</td>
<td>49.14</td>
<td>36.90</td>
<td>3384</td>
<td>1.3x</td>
<td>34.92</td>
<td>2357</td>
<td>1.4x</td>
<td>34.92</td>
<td>2357</td>
</tr>
<tr>
<td>wb_dma3</td>
<td>304.18</td>
<td>182.09</td>
<td>5135</td>
<td>1.7x</td>
<td>179.51</td>
<td>3241</td>
<td>1.7x</td>
<td>179.51</td>
<td>3241</td>
</tr>
<tr>
<td>vga1</td>
<td>434.81</td>
<td>172.51</td>
<td>145</td>
<td>2.5x</td>
<td>200.75</td>
<td>192</td>
<td>2.2x</td>
<td>200.75</td>
<td>192</td>
</tr>
<tr>
<td>vga2</td>
<td>106.98</td>
<td>147.95</td>
<td>277</td>
<td>0.7x</td>
<td>66.33</td>
<td>135</td>
<td>1.6x</td>
<td>66.33</td>
<td>135</td>
</tr>
<tr>
<td>ucrc_par</td>
<td>7.97</td>
<td>3.94</td>
<td>0</td>
<td>2.0x</td>
<td>6.38</td>
<td>0</td>
<td>1.3x</td>
<td>6.38</td>
<td>0</td>
</tr>
<tr>
<td>mem_ctrl1</td>
<td>12.53</td>
<td>24.67</td>
<td>567</td>
<td>0.5x</td>
<td>7.13</td>
<td>63</td>
<td>1.8x</td>
<td>7.13</td>
<td>63</td>
</tr>
<tr>
<td>mem_ctrl2</td>
<td>11.76</td>
<td>4.78</td>
<td>0</td>
<td>2.5x</td>
<td>4.76</td>
<td>0</td>
<td>2.5x</td>
<td>4.76</td>
<td>0</td>
</tr>
<tr>
<td>mips7891</td>
<td>22.08</td>
<td>13.51</td>
<td>53</td>
<td>1.6x</td>
<td>15.80</td>
<td>34</td>
<td>1.5x</td>
<td>15.80</td>
<td>34</td>
</tr>
<tr>
<td>opensparc_ddr21</td>
<td>48.45</td>
<td>33.42</td>
<td>1072</td>
<td>1.4x</td>
<td>29.81</td>
<td>1190</td>
<td>1.6x</td>
<td>29.81</td>
<td>1190</td>
</tr>
<tr>
<td>opensparc_ddr22</td>
<td>44.11</td>
<td>39.39</td>
<td>1138</td>
<td>1.1x</td>
<td>36.57</td>
<td>1394</td>
<td>1.2x</td>
<td>36.57</td>
<td>1394</td>
</tr>
</tbody>
</table>
### Table 3.3: Design Debugging with Non-Solution Implications and Reverse Dominance Results

<table>
<thead>
<tr>
<th>Instance</th>
<th>dbg-dom</th>
<th>dbg-dom+rev</th>
<th>dbg-dom+rev+RR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>time (s)</td>
<td>time (s)</td>
</tr>
<tr>
<td></td>
<td># impl</td>
<td># impl</td>
<td># impl</td>
</tr>
<tr>
<td></td>
<td>non-sols</td>
<td>non-sols</td>
<td>non-sols</td>
</tr>
<tr>
<td></td>
<td>imprv</td>
<td>imprv</td>
<td>imprv</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>design1-1</td>
<td>53.40</td>
<td>25.08</td>
<td>20.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>556</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.1x</td>
<td>2.7x</td>
</tr>
<tr>
<td>design1-2</td>
<td>72.54</td>
<td>38.27</td>
<td>38.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5073</td>
<td>5886</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.9x</td>
<td>1.9x</td>
</tr>
<tr>
<td>design1-3</td>
<td>39.63</td>
<td>31.69</td>
<td>29.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>210</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3x</td>
<td>1.4x</td>
</tr>
<tr>
<td>design1-4</td>
<td>100.89</td>
<td>45.69</td>
<td>42.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5854</td>
<td>5882</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.2x</td>
<td>2.4x</td>
</tr>
<tr>
<td>design1-5</td>
<td>73.72</td>
<td>27.04</td>
<td>29.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5760</td>
<td>5665</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.7x</td>
<td>2.6x</td>
</tr>
<tr>
<td>design2-1</td>
<td>18.47</td>
<td>14.59</td>
<td>9.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>543</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3x</td>
<td>1.9x</td>
</tr>
<tr>
<td>design2-2</td>
<td>7.38</td>
<td>4.23</td>
<td>4.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>53</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7x</td>
<td>1.6x</td>
</tr>
<tr>
<td>design2-3</td>
<td>0.13</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6x</td>
<td>1.4x</td>
</tr>
<tr>
<td>design3-1</td>
<td>3.03</td>
<td>2.07</td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>187</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6x</td>
<td>1.6x</td>
</tr>
<tr>
<td>design3-2</td>
<td>0.083</td>
<td>0.07</td>
<td>0.066</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.2x</td>
<td>1.3x</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>73.17</td>
<td>47.03</td>
<td>42.49</td>
</tr>
</tbody>
</table>
for rsdecoder2. In this figure, the y-axis displays the number of solutions and the x-axis shows the SAT run-time. Clearly, \texttt{dbg-dom+rev+RR} outperforms \texttt{dbg-dom} by discovering solutions at a significantly faster rate. In addition to this faster rate, \texttt{dbg-dom+rev+RR} returns earlier solutions faster than its average rate (i.e., its solutions plot is concave). This is beneficial because it allows the designer to examine those solutions earlier while the debugger continues to run.

Figure 3.8 plots the number of implied non-solutions versus run-time for \texttt{dbg-dom} and \texttt{dbg-dom+rev+RR} for rsdecoder2 with the y-axis displays the number of learned non-solutions and the x-axis displays the time(s). In this figure, each implied non-solution found during the search is recorded at the time the SAT solver returns the next solution. Although \texttt{dbg-dom+rev} implies more non-solutions overall compared to \texttt{dbg-dom+rev+RR}, the latter learns non-solution blocks earlier. This is true in general, with slightly more non-solutions being implied on average in \texttt{dbg-dom+rev} (28\%) than in \texttt{dbg-dom+rev+RR} (25\%). However, non-solution implications in \texttt{dbg-dom+rev+RR} are usually found earlier than in \texttt{dbg-dom+rev} because the former tries to actively go into parts of the search-space where it is
easier to learn non-solutions. Returning non-solutions early is favorable because it helps the SAT solver prune the search-space faster.

3.6 Summary

This research shows how to leverage reverse dominators in a circuit to speed up all-solutions SAT-based automated design debugging. New SAT decision and restart heuristics are proposed for design debugging, which expedites the learning of non-solution blocks. Non-solution implications are performed to further prune non-solution areas of the problem search space. Finally, an extensive set of experiments on real industrial designs demonstrates the robustness and practicality of the presented framework.
Figure 3.9: Performance Results
Chapter 4

Reviving Erroneous Stability-based Clock-gating using Partial Max-SAT

4.1 Introduction

In this chapter, we propose a debugging methodology for low-power designs, specifically for stability condition clock-gated circuits. The proposed framework consists of three steps: preprocessing, debugging and rectification. The preprocessing step identifies erroneous components in the design. The debugging step then uses the Partial Max-SAT debugging formulation to locate gates in the erroneous components that may be defective. Finally, rectification is performed to suggest fixes for errors introduced by clock-gating implementations. Extensive experiments on benchmark circuits show the efficacy of this approach.

4.2 Previous Work

In modern chips, the dynamic switching of the clock network accounts for 30-40% of the total power consumption. Together with the large capacitance of the clock distribution network and high switching activity, the clock network is expected to remain a significant contributor to the total dynamic power consumption. Thus, minimizing the dynamic switching of the clock network is imperative for power-savings. Clock-gating is the most common way to reduce...
Chapter 4. Reviving Erroneous Stability-based Clock-gating

54

power consumption that stems from dynamic switching. Clock-gating turns off the clock when it is not required. As mentioned above, this research focuses on stability-based clock-gating. Stability-based clock-gating turns off the clock of a register when the output of the register does not change for two or more consecutive clock-cycles (i.e., \( q^t = q^{t-1} \)). This is called stability condition (STC). For the rest of this thesis, the term clock-gating is defined as clock-gating under STC. Moreover, we only consider clock-gating for power-saving purposes. Once a gating condition has been computed, it must be synthesized and this usually requires extra circuitry. The additional area and power overhead from this circuitry diminishes the power-savings that the circuitry seeks to provide. Therefore, determining the balance between coverage and cost of the clock-gating circuitry is a difficult synthesis problem.

The authors of \cite{23} present a SAT-based methodology to compute stability conditions. This method computes the stability conditions of a register out of a set of existing signals in the design. As this technique only constructs conditions from a limited set of signals, all the clock-gating conditions may not be achieved. However, the employment of existing signals allows the condition to be implemented using only a small number of gates. As a result, gating conditions computed using this method have a favorable balance between their coverage and their cost. This framework is extended and improved using interpolation in \cite{30}. The authors in \cite{30} also propose several heuristics to select signals for computing clock-gating conditions. These techniques demonstrate that Boolean Satisfiability can be used to construct efficient clock-gating conditions.

Despite these advancements in identifying clock-gating conditions, manual effort is still required to realize all available power-savings. It is much easier for designers who have a detailed understanding of the circuit to extract the STC condition and implement it manually for a particular register. Manual clock-gating insertion is usually done at gate-level netlist. The reason is twofold. First, at the gate-level, designers can estimate the cost of the additional clock-gating circuitry which is hard to estimate if implemented at a higher level. Second, implementing clock-gating at the gate-level prevents designers from using signals that are optimized out by the synthesis tool. Due to manual intervention, clock-gating insertion is susceptible to errors. With increasing design complexity, debugging these errors is an arduous task. As such, it is
common for verification engineers to discard clock-gating implementations to remove the errors. This leads to suboptimal power-savings.

Previous work [4, 28] fixes problematic/erroneous clock-gating conditions by adding circuitry. In [28], the error effects are canceled out before being observed at the primary output by injecting new clock-gating conditions. In [4], problematic clock-gating conditions are merged by approximation and clustering techniques. As discussed earlier, finding additional clock-gating conditions is an extremely challenging task and merging clock-gating is a complex process requiring a profound understanding of the design. In contrast, our technique refrains from introducing additional clock-gating conditions or merging different clock-gating logic by fixing the root cause of the design error. The motivation behind the proposed approach is to retain most or all of the manual modifications in the design while being able to rectify the circuit.

4.3 Clock-gating Preprocessing

In this section, we present our preprocessing technique to identify erroneous components in a stability-based clock-gated circuit. This preprocessing is fast and can greatly reduce the complexity of the debugging problem. Section 4.4 presents the partial Max-SAT debugging formulation for clock-gated circuits and Section 4.5 proposes the rectification methodology.

The preprocessing step identifies the erroneous component(s) where the erroneous gate may be located. Hence, instead of analyzing all components, the SAT solver only needs to analyze a subset. This can reduce the complexity of the problem considerably. Overall, our preprocessing technique does not add significant run-time overhead and can greatly simplify the problem. Before presenting our preprocessing method, we will first formally define different components in a clock-gated circuit. This partition directly affects actions of our debugging methodology.

4.3.1 Definitions

Consider an erroneous clock-gated circuit $C$, a counter-example of length $k$ and an error cardinality $N$. To simplify our presentation, we assume that $N = 1$. Let $clk$ denote the primary
clock input for $C$. For a single register, the term *clock signal* is used to indicate its dedicated clock input. The Figure 4.2 shows a clock-gated circuit $C$.

![Figure 4.1: A Clock-gating Design](image)

In $C$, all current and next state nodes $d_1, ..., d_z$ and $q_1, ..., q_z$ are grouped as one component, called *state elements*. All other nodes in $C$ are grouped into two components, *clock-gating* and *combinational*, as defined below.

**Definition 6** A node $u$ is said to be a clock-gating node if every path from $u$ to a primary output pass through at least one clock input of a register.

From this definition, a clock-gating node can only be either a clock input or a node that is used to generate an enable signal. Every clock signal is obviously a clock-gating node. Moreover, we can prove the following lemma for a clock-gating node.

**Lemma 3** A node $u$ is a clock-gating node if and only if every path from $u$ to a primary output passes through at least one clock-gating node.

*Proof:* “Only if” direction: If $u$ is a clock-gating node, then every path from $u$ to a primary output contains at least one clock-gating node.
This is trivial as from Definition 6 every path from a clock-gating node $u$ to a primary output contains a clock-signal which, as stated, is a clock-gating node.

“If” direction: If every path from $u$ to a primary output contains at least one clock-gating node, then $u$ is a clock-gating node.

A path from $u$ to a primary output contains a clock-gating node $v$. However, we know that $v$ must pass through a clock signal to get to a primary output. Therefore, A path from $u$ to a primary output must contain a clock signal which implies that $u$ is a clock-gating node.

Example 17 Consider the clock-gating design in Figure 4.2. In order for a node in the clock-gating circuitry to get to a primary output, it has to pass through a clock of one of the register. As we know every clock input is a clock-gating node.

From Lemma 3 a clock-gating node can be formally expressed as

$$u \in CLK \iff \forall P(u \rightsquigarrow y) \exists v, v \in CLK \land v \in P(u \rightsquigarrow y)$$

where $P(u \rightsquigarrow y)$ denotes a structural path of circuit lines from $u$ to a primary output $y$, $v$ represents a clock-gating node and $CLK$ denotes the set of all clock-gating nodes.

For example, $en_1$ in Figure 4.2 is a clock-gating node since every path from $en_1$ to a primary output includes the clock signal of $R_1$. By Definition 6 the clock signal of $R_p$ is also a clock-gating node although $R_p$ is not clock-gated.

Definition 7 A node $u$ is said to be a combinational node if there exists a path from $u$ to a primary output node that does not contain a clock-gating node. This can be formally expressed as

$$u \in COM \iff \exists P(u \rightsquigarrow y) \forall v \in P(u \rightsquigarrow y), v \notin CLK$$

where $COM$ denotes the set of all combinational nodes.

From Definition 7, primary outputs are combinational nodes. Moreover, as the definition of a combinational node is simply a contrapositive of Lemma 3 it is trivial that $CLK \cap COM = \emptyset$.

In fact, we can prove that:
Lemma 4 If node $u$ is a combinational node, then at least one fanout of $u$ is not a clock-gating node.

Proof: This is trivial since every path from $u$ to a primary output has to pass through one of the fanouts of $u$. Therefore, if all fanouts of $u$ are clock-gating nodes then every path from $u$ consists at least one clock-gating node. As a result, $u$ cannot be a combinational node if all of its fanouts are clock-gating nodes.  

Algorithm 4: clock-gating Nodes Computation

\begin{verbatim}
input : $C$
output: $CLK, COM$

foreach $y \in Y$ do
    Nodeslist.pushback($y$);
    COM.pushback($y$);
end

while Nodeslist is not empty do
    $u \leftarrow$ Nodeslist.top();
    visited $\leftarrow u$;
    if $u$ is not a clock signal then
        if $u$ is not a state element then COM.pushback($u$);
        foreach $i$ fanin of $u$ do
            if $i \notin$ visited then
                Nodeslist.pushback($i$);
            end
        end
    end
end

CLK $\leftarrow$ allnodes \ $(COM \cup S)$;
\end{verbatim}

In Figure 4.2, $CLK$ contains all nodes in the clock-gating circuitry including enable signals
Chapter 4. Reviving Erroneous Stability-based Clock-gating

Figure 4.2: Clock-gated Design

(e.g., $en_1$) the clock input $clk$ and all the clock signals of all registers. $COM$ consists of all nodes in the combinational circuitry, including all inputs $x_1, ..., x_n$, outputs $y_1, ..., y_m$. State elements $d_1, ..., d_z$ and $q_1, ..., q_z$ become the remaining components in the circuit.

Algorithm 4 demonstrates how to identify clock-gating and combinational nodes in $C$. First, all outputs in $C$ are added to the combinational set ($COM$) (lines 1-4). Next, the circuit is traversed from primary outputs to primary inputs in order to find all combinational nodes (lines 5-16). Lemma 4 is used to determine if a node belongs to $COM$ (lines 8-9). After all combinational nodes are located, $CLK$ is all other nodes in circuit $C$ except state elements.

Algorithm 4 is guaranteed to terminate because each node is only visited once (line 12) and the number of nodes is finite. Moreover, as Algorithm 4 visits all nodes except clock-gating nodes, it successfully finds all combinational nodes.

4.3.2 Erroneous Components Identification

In this section, we examine scenarios in which clock-gating circuitry is reliable. If $CLK$ is reliable, the erroneous component can therefore only be found in the combinational circuitry.

To identify erroneous components, we first construct an enhanced circuit $C_{en}$ from $C$ such
that $C_{en}$, while maintaining the functionality of $C$, does not employ any clock-gating. Figure 4.3 illustrates how $C_{en}$ is constructed from $C$. To maintain the functionality of $C$, $C_{en}$ retains all registers $R_1, \ldots, R_z$ and the combinational logic $\text{COM}$ from $C$; however, all clock signals of $C_{en}$ are replaced by $clk$. As a result, none of the clock signals of $C_{en}$ are turned off at any clock cycle. Using $C_{en}$, we are able to determine whether a modification in $CLK$ can fix the erroneous behavior presented by the counter-example. If a change in $CLK$ cannot fix the erroneous behavior, $CLK$ is reliable.

Given a erroneous clock-gated circuit $C$, its enhanced circuit $C_{en}$, and a counter-example length $k$, the following theorem gives the condition for which $CLK$ is reliable.

**Theorem 2** Given a counter-example trace of length $k$, if $C$ and $C_{en}$ produce the same states and outputs for all time-frame $t$ less than or equal to $k$, then any change in $CLK$ of $C$ cannot fix the error in $C$.

**Proof:** The assumption in Theorem 2 can be formally expressed as:

$$\forall t \leq k, \neg (S^t_{en} = S^t \land Y^t_{en} = Y^t)$$

(4.1)

where $S_{en}, Y_{en}$ are states and outputs of $C_{en}$.

We prove this theorem by contradiction. For the sake of contradiction, let us assume that there exists a circuit $C_c$, a rectified version of $C$, such that $C_c$ is constructed by just modifying
CLK of C. From the construction of $C_{en}$ and $C_c$, we know that the combinational circuitry and state elements of $C_{en}$ and $C_c$ are the same as $C$.

Assume that time-frame $t$ is the first time-frame that $C_c$ has a discrepancy from $C$ that eventually propagates to the primary outputs and fixes the erroneous behavior. Consequently, for time-frame $t-1$, the states and outputs of $C$, $C_{en}$ and $C_c$ are the same. Therefore, we have:

\[(S_{c}^{t-1} = S_{en}^{t-1} = S_{c}^{t-1}) \land (Y_{c}^{t-1} = Y_{en}^{t-1} = Y_{c}^{t-1}) \] (4.2)

Since $C$, $C_c$, and $C_{en}$ have the same combinational circuitry, state elements, input values from the counter-example and same previous states (as indicated in (4.2)) their combinational nodes have the same values at time-frame $t$. As a result, at time-frame $t$, the discrepancy must be located at a clock-gating node. According to Definition 6, a path from this clock-gating node to a primary output contains a clock signal of a register. Let $R$ denote this register in $C$ and $R_c$, $R_{en}$ denote its counterparts in $C_c$ and $C_{en}$, respectively.

Since we know the discrepancy propagates to the primary output through the clock signal of $R$ at time-frame $t$, the output $q_c^t$ of $R_c$ should be different from its counterparts $q^t$ and $q_{en}^t$. Otherwise, the difference in the clock signal of $R$ and $R_c$ does not propagate to the output and cannot fix the erroneous behavior as stated.

In case $C_{en}$ and $C$ produce different states or outputs, we cannot make any statement on the erroneous components. The reason for this is because the erroneous node can still be located in the combinational circuitry but it propagates through the clock-gating circuitry. As a result, when clock-gating circuitry is removed from $C$ to construct $C_{en}$, the erroneous behavior cannot be observed in $C_{en}$ even though the erroneous gate still exists in it.

\[(q_c^t \neq q^t) \land (q_c^t \neq q_{en}^t) \] (4.3)

First, consider that the clock signal of $R_c$ is turned on. At time-frame $t$, $C_c$ functions exactly as $C_{en}$. This is because $C$ and $C_{en}$ share the same primary inputs from the counter-example, same last states as indicated in (4.2) and the same clock signal for $R_c$ and $R_{en}$ (both ON). Therefore, $q_c^t$ has the same value as $q_{en}^t$. This is a contradiction with (4.3).
Second, consider the case where clock signals of both \( R_c \) and \( R \) are turned off at time-frame \( t \). Using the same argument as the previous case, \( R \) and \( R_c \) have the same output as well, (i.e., \( q^t_c = q^t \)). Therefore, this also leads to a contradiction.

Finally, consider the case in which the clock signal of \( R_c \) is off whereas the clock signal of \( R \) is on. If clock signal of register \( R_c \) is turned off under STC, its output should be the same as when the clock is turned on (Definition 3). However, in this case because of Equation 4.3, they are different. Therefore, \( C_c \) is still erroneous in this case. This also leads to a contradiction since we assume that \( C_c \) is a rectified version of \( C \).

Therefore, given a counter-example length \( k \), if \( C \) and \( C_{en} \) produce the same states and outputs for all time-frames \( t, t < k \), then no change in \( CLK \) of \( C \) can fix its error (i.e., no such \( C_c \) exists).

**Corollary 1** Given an erroneous clock-gated circuit \( C \), its enhanced circuit \( C_{en} \), and a counter-example trace of length \( k \). If \( C_{en} \) and \( C \) produce the same states and primary outputs for the counter-example trace, then the erroneous gate \( g \) is a combinational node.

**Proof:** This corollary is a direct implication from Theorem 2. Note that Theorem 2 proves that if \( C \) and \( C_{en} \) produce the same states/outputs, any change in \( CLK \) cannot fix the error in \( C \). Hence, given such condition, the erroneous node has to be a combinational node.

Algorithm 5 shows our preprocessing step after \( CLK \) and \( COM \) computed. First, the enhanced circuit \( C_{en} \) is constructed from \( C \) (line 1) by copying all nodes and edges from \( C \) except for \( CLK \) nodes and edges between them. The clock signals of all state elements in \( C_{en} \) are replaced by the primary clock input \( clk \). Next we run simulation on both circuits using the trace from the counter-example and compare their outputs and states for all time-frames \( t, t < k \) (lines 5-8). If there is a mismatch between \( C \) and \( C_{en} \), the algorithm returns false, indicating that we cannot assume anything about the erroneous region. Otherwise, if all states and outputs are equal, the algorithm returns true indicating that the erroneous gate must be a combinational node.

As mentioned, clock-gating insertion is usually done at the gate-level netlist. Thus, the
debugging strategy for clock-gating must work well at the gate-level. Based on this observation, we will use the debugging Partial Max-SAT formulation instead of the error-select variable approach. This is because at the gate-level, the number of error-select variables required could equal the total number of gates which is millions in modern chips. This significantly increases the complexity of the problem. Hence, the techniques presented in this chapter are different from the ones presented in Chapter 3 as those are developed for debugging at the RTL block level.

4.4 Clock-gating Debugging

This section shows how to debug clock-gating designs using the Partial Max-SAT formulation presented in Section 2.4.1. In this technique, the problem is encoded as a Partial Max-SAT instance \( \text{Debug}_{\text{Max-SAT}} \) as shown in Section 2.9. During the encoding process, it is necessary to use time-frame expansion to model the sequential behavior of the circuit. However, the time-frame expansion technique in Section 2.4 assumes the same clock signal for all registers and it does not model the functionality of a clock-gated register. Figure 4.4 shows a transformation that allows us to use time-frame expansion for a clock-gated circuit.

Figure 4.4(a) shows a clock-gated register. As this thesis only focuses on functional errors,
we only need to model the functional behavior of a clock-gated register. Figure 4.4(b) shows the logical equivalence of a clock-gated register that is used in our encoding process. Given the same input values on \( d, en \) and \( clk \), \( q \) in Figure 4.4(b) has the same value as \( q \) in Figure 4.4(a). Moreover, the register in Figure 4.4(b) like non-clock-gated registers, uses the clock primary input \( clk \) as its clock signal. Hence, this transformation allows us to capture the logical behavior of clock-gating while still guaranteeing that all registers have the same clock signal \( clk \) so that time-frame expansion can be applied.

After the transformation has been applied, the problem is encoded as an \textsc{UNSAT} instance \( \text{Debug}_{\text{Max-SAT}} \), where each solution is a set of clauses that correspond to a potentially erroneous gate. The procedure to find all solutions for a debugging problem using the Partial Max-SAT solver is presented in Section 2.4.2. This procedure is optimized for clock-gating designs as follows.

In the previous section, a preprocessing step to identify erroneous components is presented. The information from the preprocessing step is used to reduce the complexity of the debugging problem. Particularly, if \( COM \) is detected as the only erroneous component, we set all clauses that correspond to \( CLK \) as hard clauses. This is because these clauses cannot be solutions for the Partial Max-SAT instance. The increase in the number of hard clause reduces the number of soft clauses and hence reduces the complexity of the problem.

To further increase the number of hard clauses, we employ the sliding window technique presented in 25. First, the problem is partitioned into multiple time windows where each

![Figure 4.4: Clock-gating Transformation](image-url)
window $W$ spans across a fixed number of time-frames. Each window is visited from the latest to the earliest. Only the clauses in the current analyzed window are set as soft clauses, while all other windows are set as hard clauses. Statistically, by applying the sliding window technique, the number of soft clauses decreases from $h \cdot |W|$ to $|W|$ at any given time, where $h$ denotes the number of windows and $|W|$ denotes the number of soft clauses in a window.

Algorithm 6 outlines our debugging process. Our algorithm takes in the unsatisfiable instance of $\text{Debug}_{\text{Max-SAT}}$, the mapping from CNF clauses to gates and the error cardinality $N$. Each window is analyzed by traversing from the last window to the first (lines 2-13). When a window is analyzed, all other windows are set as hard clauses (line 5). In each window, several solutions are found. Each solution is mapped back to a gate in $C$ and stored in the erroneous gates set called $\text{ERROR}$ (line 9). Next, the instance $\text{Debug}_{\text{Max-SAT}}$ is updated with new hard clauses which are previous solutions (line 10). The algorithm stops when no more solutions can be found or all the solutions of the error cardinality $N$ have been discovered (line 17).

### 4.5 Clock-gating Node Rectification

In this section we propose a technique that allows us to suggest rectifications for erroneous clock-gating nodes. First, all the available fixes for the erroneous clock-gating node are listed. Each fix is verified that it corrects the erroneous behavior exhibited by the counter-example. Moreover, the fix is verified to ensure that it does not introduce any new errors. All qualified fixes are short-listed and ranked based on their power-saving level. These qualified fixes are called potential fixes.

Unlike the combinational component, clock-gating circuitry is much smaller in terms of the number of gates involved and they are usually added manually by the designer. Due to the nature of clock-gating circuitry construction, a dictionary-based rectification technique is adequate to correct the error. In order to fix the error, we simply switch the erroneous gate using a dictionary-based approach and verify the correctness of the clock-gating cone that contains the erroneous node.

A fix that is derived from the dictionary of gates is not guaranteed to be the appropriate
Algorithm 6: Clock-gating Fault Localization

input: CNF Debug, mapping M, N, Trace

output: set of erroneous gates ERROR

1 current ← h;
2 repeat
3 for t = 1 to h do
4 if t = current then setSoftClause (window t);
5 else setHardClause (window t);
6 end
7 repeat
8 sol ← Max-SAT(Debug);
9 ERROR ← ERROR ∪ M(sol);
10 Debug ← setHardClause (sol);
11 until (sol ≠ ∅) ∧ (size(M(sol)) ≤ N);
12 current −− ;
13 until current = 1;

correction for the erroneous node. Therefore, a post-processing step is required to filter out spurious fixes. First, a potential fix has to correct the erroneous behavior of C. To ensure this, we apply the fix to C and simulate it using the counter-example. If the simulated outputs do not agree with values from the golden model, then the fix is discarded and the next available fix is examined. Otherwise, the fix is examined further to assert that it does not introduce new bugs.

Of course, Sequential Equivalence Checking (SEC) can be used to ensure with confidence that the fix does not introduce new errors. However, a full-blown SEC step is costly and for one erroneous gate, multiple fixes must be checked. This makes SEC an impractical approach. Since our fixes are only on clock-gating nodes, we can simply focus on the clock-gating circuitry. After the fix is applied, for any arbitrary trace (different from the counter-example trace), the clock-gating circuitry can:
1. produce the same output as $CLK$ of $C$;

2. turn on the clock signal while $CLK$ turns it off;

3. turn off the clock signal while $CLK$ turns it on.

The following can be deduced for the above three scenarios.

1. Here, because the clock signals are the same, no error is introduced.

2. In this case, the clock signal in $C$ is turned off under STC. Based on Definition 3 of STC, turning on the clock signal does not modify any states/outputs. Therefore, no error is introduced.

3. In this case, the clock signal in $C$ is turned on, while the fixed clock-gating circuitry turns it off. This could alternate the register value and potentially introduce new errors.

In order to conservatively ensure that no error is introduced, one can simply assert that the third scenario, as discussed above, does not occur. To prevent it, after a fix is applied the clock-gating circuitry must not turn off the clock signal when $CLK$ of $C$ has it on.

Let us denote $CLK_{fix}$ as the clock-gating circuitry after the fix has been applied. The symbol $en$ is used to denote the enable signal that has the potentially erroneous gate in its fanin cone. The $en_{fix}$ denotes the counterpart of $en$ in $CLK_{fix}$. If the gate propagates to more than one enable signal, we extend our check to each enable signal. However, to simplify our presentation without the loss of generality, let us assume that the gate only propagates to one enable signal $en$. Our goal is to ensure that when $en = 1$ (ON), $en_{fix} = 1$ (ON). This can be formally expressed as

$$ en = 1 \implies en_{fix} = 1 \quad (4.4) $$

Overall, if Property 4.4 is satisfied, the fix does not introduce new errors and therefore, it is a potential fix.

Algorithm 7 shows our rectification process for clock-gating circuitry. After the debugging step returns all possible erroneous gates, we only select gates in $CLK$ for further analysis (line 1). In order to fix the potentially erroneous gate, we replace it with different gates in the
dictionary. Each possible fix is verified such that it corrects the defective behavior exhibited by the counter-example. If the fix passes the first check, we construct $CLK_{fix}$ and check if the condition in (4.4) is satisfied as well. All qualified fixes are stored in $CORRECTION$ (line 6). The loop terminates when all the available fixes are tried (line 3). Next, all potential fixes are ranked based on their power-saving level (line 12).

---

**Algorithm 7:** Clock-gating Node Rectification

**input:** $C$, $CLK$, $ERROR$, dictionary

**output:** $gates$, $CORRECTION$

1 foreach $g \in ERROR$ do
2      if $g \in CLK$ then
3          repeat
4              $CLK_{fix} \leftarrow Rectify(dictionary, CLK)$ ;
5              if $Verify(CLK, CLK_{fix})$ then
6                  $CORRECTION \leftarrow E_{fix}$ ;
7              end
8          trial++ ;
9          until $trial = dictionary.size()$;
10      end
11 end
12 $Rank(CORRECTION)$ ;

---

### 4.6 Experimental Results

This section presents the experimental results for the proposed clock-gating debug framework. The presented framework is implemented on state-of-art Max-SAT solver from \[40\]. All experiments are run on an Intel Core i5 3.1 GHz quad-core workstation with 8 GB of RAM; timeout is set at 7200 seconds.

Several ISCAS-89 and ITC-99 benchmark circuits are used in our experiments. The clock-gated versions of the circuits are implemented using Synopsys Design Compiler utilizing the
Chapter 4. Reviving Erroneous Stability-based Clock-gating

The gate level power consumption is measured using Power Compiler under normal loading at room temperature. For each design, debugging instances are generated by injecting different design errors such as incorrect gate function, wrong gate input etc., both in the combinational logic as well as in the clock-gating circuitry. Each circuit is simulated using a test bench. The simulated output values are compared with the expected output values of the circuit. As soon as an inconsistency is detected between the expected and observed values of the circuit, the simulation is terminated and the trace is recorded. The circuit is then converted into CNF using the Tseitin transformation\(^5\). For each instance, the experiments are conducted with and without preprocessing. The sliding window technique is applied in all experiments. When potential erroneous gates are found the clock-gating circuitry, rectifications are derived using techniques in Section 4.5.

Table 4.1 and Table 4.2 show the information of all debugging instances. The first column gives the instance name. The next three columns respectively show the length of the counter example \(k\), the total number of gates and the number of potentially erroneous gates discovered. ‘TO’ indicates the experiments time out in both cases with and without preprocessing. Similarly, ‘MO’ indicates the experiments mem out in both cases.

Table 4.3 shows the results of instances where Algorithm 5 determines that the error resides in the combinational logic (COM). The first column of Table 4.3 lists the instance name. Column 2, 3 respectively specify the number of soft clauses, number of hard clauses of each analyzing window. Column 4 shows the total debugging run-time without the preprocessing step. The next four columns shows the number of soft clauses, number of hard clauses of each analyzing window, the preprocessing time and run-time with the preprocessing step. The improvement in the number of hard clauses and the run-time is stated in Column 8 and 9. It can be seen that the preprocessing step, while it adds very little overhead, can greatly reduce the complexity of the problem indicated in the increasing number of hard clauses. For example, for instance “s838_1” after the preprocessing step, the number of hard clauses is increased by 17 times. This allows us to solve this instance under the timeout limit, whereas without the preprocessing step this cannot be accomplished. Besides the “s838_1” instance, on average the preprocessing step reduces 12% debugging run-time of instances in Table 4.3.
Table 4.1: Instance Information - Table 1

<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Trace Length</th>
<th># nodes</th>
<th># sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382_1</td>
<td>15</td>
<td>172</td>
<td>36</td>
</tr>
<tr>
<td>s382_2</td>
<td>113</td>
<td>172</td>
<td>62</td>
</tr>
<tr>
<td>s298_1</td>
<td>43</td>
<td>147</td>
<td>21</td>
</tr>
<tr>
<td>s298_2</td>
<td>43</td>
<td>147</td>
<td>186</td>
</tr>
<tr>
<td>s344_1</td>
<td>225</td>
<td>154</td>
<td>15</td>
</tr>
<tr>
<td>s344_2</td>
<td>11</td>
<td>154</td>
<td>30</td>
</tr>
<tr>
<td>s349_1</td>
<td>39</td>
<td>133</td>
<td>34</td>
</tr>
<tr>
<td>s349_2</td>
<td>11</td>
<td>133</td>
<td>25</td>
</tr>
<tr>
<td>s9234_1</td>
<td>51</td>
<td>1125</td>
<td>240</td>
</tr>
<tr>
<td>s9234_2</td>
<td>10</td>
<td>1125</td>
<td>33</td>
</tr>
<tr>
<td>s1423_1</td>
<td>77</td>
<td>989</td>
<td>TO</td>
</tr>
<tr>
<td>s1423_2</td>
<td>1449</td>
<td>989</td>
<td>MO</td>
</tr>
<tr>
<td>s838_1</td>
<td>17</td>
<td>412</td>
<td>64</td>
</tr>
<tr>
<td>s838_2</td>
<td>39</td>
<td>412</td>
<td>261</td>
</tr>
<tr>
<td>s420_1</td>
<td>1977</td>
<td>205</td>
<td>TO</td>
</tr>
</tbody>
</table>

Table 4.1 shows the results of the remaining instances. In these cases, no statement can be made about the components where the erroneous gate locates. Thus, the number of hard clauses and the number of soft clauses are the same for both with and without preprocessing. Column 1 gives the name of the instances. Column 2 and 3 show the number of hard clauses and the number of soft clauses in each analyzing window. Column 4 specifies the preprocessing time. Column 5 shows the number of potentially erroneous gates in the clock-gating circuitry. Column 6 displays the power consumption of the erroneous circuit. The next column, Column 7, shows the power consumption when we remove clock-gating enable signals that are fanouts of the potentially erroneous gate. This depicts the scenario where engineers remove
Table 4.2: Instance Information - Table 2

<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Trace Length</th>
<th># nodes</th>
<th># sols</th>
</tr>
</thead>
<tbody>
<tr>
<td>s420_2</td>
<td>30</td>
<td>205</td>
<td>48</td>
</tr>
<tr>
<td>b03_1</td>
<td>27</td>
<td>101</td>
<td>53</td>
</tr>
<tr>
<td>b03_2</td>
<td>23</td>
<td>101</td>
<td>40</td>
</tr>
<tr>
<td>b04_1</td>
<td>27</td>
<td>695</td>
<td>43</td>
</tr>
<tr>
<td>b04_2</td>
<td>9</td>
<td>695</td>
<td>27</td>
</tr>
<tr>
<td>b05_1</td>
<td>9</td>
<td>1477</td>
<td>333</td>
</tr>
<tr>
<td>b05_2</td>
<td>137</td>
<td>1477</td>
<td>67</td>
</tr>
<tr>
<td>b07_1</td>
<td>89</td>
<td>437</td>
<td>165</td>
</tr>
<tr>
<td>b07_2</td>
<td>51</td>
<td>437</td>
<td>57</td>
</tr>
<tr>
<td>b08_1</td>
<td>94</td>
<td>141</td>
<td>46</td>
</tr>
<tr>
<td>b08_2</td>
<td>30</td>
<td>141</td>
<td>190</td>
</tr>
<tr>
<td>b09_1</td>
<td>39</td>
<td>171</td>
<td>105</td>
</tr>
<tr>
<td>b09_2</td>
<td>35</td>
<td>171</td>
<td>180</td>
</tr>
<tr>
<td>b12_1</td>
<td>153</td>
<td>1063</td>
<td>726</td>
</tr>
<tr>
<td>b12_2</td>
<td>15</td>
<td>1063</td>
<td>TO</td>
</tr>
</tbody>
</table>

clock-gating implementations to correct the error(s). Column~8 shows the power consumption when one of the suggested rectifications is applied. In this case, the rectification that gives the lowest power consumption is selected. The percentage of power-saving that is reclaimed by the rectification is indicated in Column~9. The last column specifies the run-time of the entire debugging flow. As stated, the SAT formulas for these instances are exactly the same with or without the preprocessing step. This leads to the same amount of time spent in debugging step and rectification step with or without the preprocessing step. Thus, for simplicity, we will only report the total debugging run-time without the preprocessing step in Column~10. The total debugging run-time with the preprocessing step is approximately equal with the values in
Column~10 as the run-time of the preprocessing step is really small (<1s) for most instances. The only instance that has a noticeable preprocessing time is “s1423.2”. However, “s1423.2” times out with or without the pre-processing step. Nonetheless, the total debugging run-time with preprocessing can be computed by adding values of Column~4 and Column~10. Overall, the average reduction in debugging run-time is 6% and 80% of the power-savings are retained.

4.7 Summary

This work proposes a methodology to debug erroneous clock-gating designs. The preprocessing step identifies the relative erroneous region. The debugging problem is then encoded as a Partial Max-SAT instance where each solution corresponds to a possible candidate of the error. After all the candidates are collected, qualified rectifications are suggested for ones in the clock-gating logic. Finally, an extensive set of experiments on standard benchmark circuits demonstrates the practicality of the proposed framework.
Table 4.3: Debugging with Pre Processing Results

<table>
<thead>
<tr>
<th>Instance name</th>
<th>w/o preprocessing</th>
<th>w preprocessing</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># hard clauses</td>
<td># soft clauses</td>
<td>time (s)</td>
</tr>
<tr>
<td>s382_1</td>
<td>165</td>
<td>11176</td>
<td>3</td>
</tr>
<tr>
<td>s298_1</td>
<td>473</td>
<td>28274</td>
<td>13</td>
</tr>
<tr>
<td>s344_1</td>
<td>4950</td>
<td>140916</td>
<td>22</td>
</tr>
<tr>
<td>s349_1</td>
<td>858</td>
<td>22773</td>
<td>9</td>
</tr>
<tr>
<td>s9234_1</td>
<td>3927</td>
<td>247493</td>
<td>467</td>
</tr>
<tr>
<td>s1423_1</td>
<td>1848</td>
<td>298915</td>
<td>TO</td>
</tr>
<tr>
<td>s838_1</td>
<td>629</td>
<td>27668</td>
<td>TO</td>
</tr>
<tr>
<td>s420_1</td>
<td>41517</td>
<td>1620932</td>
<td>TO</td>
</tr>
<tr>
<td>b03_1</td>
<td>270</td>
<td>17480</td>
<td>35</td>
</tr>
<tr>
<td>b04_1</td>
<td>567</td>
<td>72884</td>
<td>115</td>
</tr>
<tr>
<td>b05_1</td>
<td>351</td>
<td>41285</td>
<td>164</td>
</tr>
<tr>
<td>b07_1</td>
<td>979</td>
<td>157555</td>
<td>1336</td>
</tr>
<tr>
<td>b08_1</td>
<td>1410</td>
<td>62399</td>
<td>127</td>
</tr>
<tr>
<td>b09_1</td>
<td>156</td>
<td>31408</td>
<td>240</td>
</tr>
<tr>
<td>b12_1</td>
<td>1989</td>
<td>711695</td>
<td>1665</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>349.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 4. Reviving Erroneous Stability-based Clock-gating

### Table 4.4: Debugging with Rectification Results

<table>
<thead>
<tr>
<th>Instance Name</th>
<th># hard clauses</th>
<th># soft clauses</th>
<th>Pre-pros time (s)</th>
<th># rect. solns</th>
<th>orig w/o cg (μW)</th>
<th>w/o rect. cg (μW)</th>
<th>improv (%)</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382_2</td>
<td>165</td>
<td>11176</td>
<td>&lt;1</td>
<td>3</td>
<td>8.8</td>
<td>13.6</td>
<td>9.0</td>
<td>10</td>
</tr>
<tr>
<td>s298_2</td>
<td>473</td>
<td>28274</td>
<td>&lt;1</td>
<td>2</td>
<td>4.78</td>
<td>5.5</td>
<td>5.34</td>
<td>22.2</td>
</tr>
<tr>
<td>s344_2</td>
<td>4950</td>
<td>140916</td>
<td>&lt;1</td>
<td>3</td>
<td>5.8</td>
<td>6.02</td>
<td>5.8</td>
<td>100</td>
</tr>
<tr>
<td>s349_2</td>
<td>858</td>
<td>22773</td>
<td>&lt;1</td>
<td>1</td>
<td>5.81</td>
<td>6.0</td>
<td>5.81</td>
<td>100</td>
</tr>
<tr>
<td>s9234_2</td>
<td>3927</td>
<td>247493</td>
<td>&lt;1</td>
<td>1</td>
<td>69.7</td>
<td>69.9</td>
<td>69.7</td>
<td>100</td>
</tr>
<tr>
<td>s1423_2</td>
<td>1848</td>
<td>298915</td>
<td>64</td>
<td>-</td>
<td>38.1</td>
<td>40.7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>s838_2</td>
<td>629</td>
<td>27668</td>
<td>1</td>
<td>2</td>
<td>14.1</td>
<td>14.4</td>
<td>14.36</td>
<td>13.3</td>
</tr>
<tr>
<td>s420_2</td>
<td>41517</td>
<td>1620932</td>
<td>1</td>
<td>3</td>
<td>7.45</td>
<td>8.33</td>
<td>7.46</td>
<td>98.8</td>
</tr>
<tr>
<td>b03_2</td>
<td>270</td>
<td>17480</td>
<td>1</td>
<td>2</td>
<td>12.47</td>
<td>14.99</td>
<td>13.9</td>
<td>43.2</td>
</tr>
<tr>
<td>b04_2</td>
<td>567</td>
<td>72884</td>
<td>&lt;1</td>
<td>2</td>
<td>57.03</td>
<td>62.90</td>
<td>57.03</td>
<td>100</td>
</tr>
<tr>
<td>b05_2</td>
<td>351</td>
<td>41285</td>
<td>4</td>
<td>3</td>
<td>9.8</td>
<td>12.4</td>
<td>9.8</td>
<td>100</td>
</tr>
<tr>
<td>b07_2</td>
<td>979</td>
<td>157555</td>
<td>1</td>
<td>3</td>
<td>15.8</td>
<td>18.3</td>
<td>15.8</td>
<td>100</td>
</tr>
<tr>
<td>b08_2</td>
<td>1410</td>
<td>62399</td>
<td>&lt;1</td>
<td>2</td>
<td>10.07</td>
<td>11.2</td>
<td>10.1</td>
<td>97.3</td>
</tr>
<tr>
<td>b09_2</td>
<td>156</td>
<td>31408</td>
<td>&lt;1</td>
<td>3</td>
<td>15.10</td>
<td>15.3</td>
<td>15.16</td>
<td>70</td>
</tr>
<tr>
<td>b12_2</td>
<td>1989</td>
<td>711695</td>
<td>1</td>
<td>-</td>
<td>52.07</td>
<td>52.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18.2</td>
<td>19.9</td>
<td>18.4</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions and Future Work

5.1 Contributions

Over the past decade, SAT solvers have become powerful engines in various VLSI CAD tools. Many VLSI related problems have been formulated as SAT instances and solved in that manner. In design debugging, SAT solvers have played important role as they are backbone of many automated design debugging techniques. Despite several advances in SAT-based design debugging techniques over the last decade, the use of these techniques in the industry is still very limited. There are two reasons for this. First, these techniques cannot handle the complexity of industrial strength designs. Second, the existing techniques are not optimized for different design types, such as low-power design, and thus it is inefficient to use these techniques with those design types.

The goal of this research is to increase SAT-based design debugging adoption in the industry in two novel ways. The first contribution improves the performance of automated design debugging techniques to close the gap between the techniques and the requirements of the industry. The second contribution adapts existing debugging techniques for low-power design, targeting clock-gated circuits.

In the first part of this thesis, we present a debugging framework that uses dominance relationships and non-solution implications. Particularly, block $b_i$ is said to be a reverse dominator
of block $b_j$ if every path from $b_i$ to a primary output contains a node in $b_j$. Furthermore, if $b_j$ is a non-solution block, then $b_i$ is also a non-solution block. A new decision heuristic where error-select variables are decided first is proposed to learn non-solution blocks early in the solving process. The restart mechanic in a SAT solver is also enhanced for non-solution learning. With non-solution blocks learned from the modified SAT solver, non-solution implications are used much more effectively. All learned non-solution blocks are pruned from the debugging search space to reduce the complexity of the problem. When applying the proposed framework, improvement is observed in all experiments. Overall, there is a 42% average reduction in solving run-time compared to the state-of-the-art SAT-based framework.

In the second part of this dissertation, we present a debugging methodology for stability-based clock-gating design. In this framework, a preprocessing step is performed initially to identify erroneous components. As clock-gating insertion is performed at the gate-level netlist, partial Max-SAT design debugging formulation is selected as the debugging method. The discovery of erroneous components reduces the complexity of the Partial Max-SAT instance. Finally, rectifications are suggested for clock-gating nodes that may be responsible for the error. Overall, this method reduces the debugging time on low-power design by 6%. Furthermore, suggested rectifications can retain part of the power-savings. On average, 80% of the power-savings can be retained.

### 5.2 Future Work

In our work using non-solution implications and reverse domination in a modern SAT-based debugging environment, the decision heuristic only moves error-select variables to the top of the decision tree. However, this does not optimize the order of error-select variables in the decision tree. If a block that has many reverse dominators is learned as a non-solution, it implies that many other blocks are also non-solutions. Hence, blocks that have many reverse dominators are good candidates to be decided first. Moreover, blocks that are easy to learn as solutions or non-solutions are also good candidates to be decided first. As a result, we would like to study which error-select variables should be decided first.
The non-solution concept in Chapter 3 only indicates blocks that are not part of any solutions. However, this concept can be extended to represent a non-solution set. For example, given the error cardinality $N = 2$, the set $\langle b_1, b_2 \rangle$ is not a solution of cardinality $N = 2$, whereas $\langle b_1, b_3 \rangle$ is a solution. According to the concept presented in Chapter 3, $b_1$ is not a non-solution. However, the set $b_1, b_2$ can still be used to imply other non-solution sets. Thus, another future direction for this research is to develop techniques that allow us to learn non-solution sets.

In the work on debugging clock-gating design, if $C_{en}$ produces different states or outputs from $C$, we cannot confidently determine the location of the erroneous gate. This limits the application of our preprocessing step. Our goal is to have a preprocessing step that is able to determine whether the erroneous gate is either a combinational node or a clock-gating node.

Moreover, the presented techniques in Chapter 4 only work on stability-based clock-gating. If a debugging methodology can be developed for ODC clock-gating, it improves the adoption of SAT-based design debugging for low-power design.

Finally, our rectification method is based on a dictionary approach. Thus, it may not produce the correct fix(es) for the error(s). However, instead of suggesting rectifications, one could provide the fixed output value for the erroneous gate. This assists the designers to derive the correct rectification for the error. To expand, after an erroneous gate $g$ is identified, a fault multiplexer model presented in [49] can be employed to compute the fixed value for $g$. 
Bibliography


