Scaling SAT-based Automated Design Debugging with Formal Methods

by

Brian Keng

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 © 2009 by Brian Keng
Abstract

Scaling SAT-based Automated Design Debugging with Formal Methods

Brian Keng
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2009

The size and complexity of modern VLSI computer chips are growing at a rapid pace. Functional debugging is increasingly becoming a bottleneck in the design flow where it can take up to 60% of the total verification time. Scaling existing automated debugging tools is necessary in order to continue along this path of rapid growth and innovation in the semiconductor industry. This thesis aims to scale automated debugging techniques with two contributions. The first contribution introduces a succinct memory model for automated design debugging that dramatically lowers the memory requirements for the debugging problem. The second contribution presents a scalable SAT-based design debugging algorithm that uses a mathematical technique called interpolation to divide the debugging problem into multiple parts across time which greatly reduces the peak memory requirements of the debugging problem. Extensive experiments on real designs demonstrate the benefit of this work.
Acknowledgements

First and foremost I would like to sincerely thank my supervisor Professor Andreas Veneris for being an excellent mentor, guide and teacher throughout my journey into research. You are a constant source of motivation and passion that drives me to achieve my full potential.

I would also like to thank my parents who are a never-ending source of love and support. You have instilled in me the values and lessons that have guided me throughout my life. I am forever grateful. Many thanks to my brother and sister who have been constantly putting a smile on my face whenever I need it.

I would also like to thank my colleagues at Venna and the University of Toronto for all their shared wisdom and constructive feedback. Special thanks to Sean Safarpour, Hratch Mangassarian, Duncan Smith, Terry Yang, Yibin Chen, Evean Qin, Alan Baker, Patrick Halina and Farzad Farzan.

Thank you to my committee members Professor Jason Anderson, Professor Jianwen Zhu and Professor Dimitris Hatzinakos for their great advice and recommendations.

Acknowledgements are due to the Ontario Graduate Scholarship program, Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Toronto for their financial support.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td></td>
<td>List of Algorithms</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Background and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Contribution</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1</td>
<td>A Succinct Memory Model for Automated Design Debugging</td>
<td>6</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Scalable VLSI Debugging with Interpolation</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Thesis Outline</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Verification Overview</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Debugging Overview</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Embedded Memory Semantics</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Boolean Satisfiability</td>
<td>18</td>
</tr>
<tr>
<td>2.5.1</td>
<td>CNF Representation</td>
<td>18</td>
</tr>
<tr>
<td>2.5.2</td>
<td>SAT Algorithms</td>
<td>20</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Unsatisfiable Cores</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Interpolants</td>
<td>25</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Simple gates and their CNF representation ........................................... 19

5.1 Circuit statistics .................................................................................. 61
5.2 Cycle accurate design debugging results for fifo and vga ....................... 65
5.3 Non-cycle accurate design debugging results on mrisc ............................. 66
5.4 Debugging with interpolants circuit statistics .......................................... 67
5.5 Debugging with interpolants results ...................................................... 68
5.6 Debugging with multiple interpolant results ........................................... 74
List of Figures

1.1 Simplified VLSI flow ................................................. 2
1.2 Typical verification cycle ........................................... 4

2.1 Time-frame expanded example ....................................... 12
2.2 Typical debugging flow .............................................. 15
2.3 Dual-port memory interface signals ............................... 17
2.4 Cardinality constraint .............................................. 20
2.5 Resolution graph and interpolant .................................. 25
2.6 SAT-based debugging ............................................... 30

3.1 Proposed memory model for design debugging .................... 39
3.2 Modelling an initial memory configuration ....................... 41
3.3 Non-cycle accurate design debugging with a final memory configuration .......................... 42

4.1 Suffix window debugging for Example 12 ....................... 49
4.2 Suffix window debugging for Example 13 ....................... 50
4.3 Resolution graph and interpolant .................................. 52
4.4 Prefix window debugging with an interpolant .................... 53

5.1 Cycle accurate performance results .............................. 63
5.2 Normalized memory size for fifo3, fifo2-init and vga6-init .......... 63
5.3 Memory model size vs. trace length for fifo ..................... 64
5.4 Performance results for debugging with interpolants ............. 70
5.5 Memory results for debugging with interpolants .......................... 72
5.6 Solutions using multiple interpolants ........................................ 73
List of Algorithms

2.1 The DPLL algorithm ................................................. 21
2.2 Modern SAT algorithm ............................................... 23
2.3 Generating an interpolant ........................................... 31
4.1 Debugging with interpolants ...................................... 58
4.2 Extracting multiple UNSAT cores ................................. 59
Chapter 1

Introduction

1.1 Background and Motivation

The size and complexity of modern Very Large Scale Integration (VLSI) computer chips are growing at a rapid pace. Turning a written specification into physical chip with hundreds of million transistors is no simple task. Computer aided design (CAD) tools are necessary components in this process, automating many steps that would otherwise be intractable manually. Constant research and innovation in these tools have been one of the contributing factors in our ability to design the complex computer systems that are available today.

This design flow to realize a computer chip is a complex process involving many steps, where each step takes a higher level description of the design and synthesizes it into a lower level one. Figure 1.1 shows a simplified version of this process. First, a high level behavioral specification written in C, Matlab or even plain English is converted into a Register Transfer Level (RTL) description such as Verilog or VHDL. Next, the RTL is synthesized into an gate-level netlist. This gate-level description is then synthesized again to a transistor-level netlist which is then placed and routed on the chip fabric. Finally, this physical layout is sent to the fabrication facility to manufacture the chip.

For each synthesis step of this process, a rigorous verification phase is necessary to ensure that no errors are introduced. Functional verification is done between the behavioral specification and the structural RTL description; equivalence checking is done between the RTL


Chapter 1. Introduction

Design Flow

1. Behavioral Specifications
2. RTL Description
3. Logic Synthesis
4. Gate-level Netlist
5. Layout Synthesis
6. Layout
7. Fabrication
8. Chip

Testing and Verification

1. Functional Verification
2. Equivalence Checking
3. Debugging
4. Testing
and the gate-level netlist; and test is done between the gate-level netlist and the physical chip. Ideally, errors or faults are found earlier rather than later in the design process since many of the non-recurring engineering costs such as mask generation have to be incurred again if a bug escapes into the silicon chip.

Functional verification is the first step where errors can be identified in the design flow and thus it is becoming increasingly important to ensure correctness at this stage to reduce costs and design time. However, despite a great deal of innovation over the past decade, functional verification and validation continues to consume up to 70% of the total ASIC development time [34]. Furthermore, after an erroneous behavior is found, the process of finding the root cause of the error, also known as debugging, can consume up to 60% of the total verification time [19]. For example, this translates to 8 months of debugging for a 20 month design cycle.

Functional verification is a cyclic process consisting of two major steps: verification and design debugging. Figure 1.2 shows the cyclic nature of this process. Verification and debugging take the majority of the design time where fixing the design is typically a much more narrow problem once the root cause is found during debugging.

The verification phase attempts to find erroneous behaviors in the RTL design with respect to its specification. Recently, many new techniques have been introduced to greatly enhance the efficiency in which verification can be done. Simulation based techniques [1, 6, 34] are still heavily used to directly exercise common behaviors in a design, while constrained random simulation [41] techniques aid in covering a broad range of a design’s state space. Assertion-based verification [18], where behavioral specifications are encoded in a formal language, has been effective at reducing verification time. Adopting an assertion-based verification methodology not only allows greater behavioral coverage, it also localizes the erroneous behavior to allow easier debugging and correction. Assertions can be used not only with traditional dynamic simulation flows but also with formal techniques such as model checking [7, 30] and equivalence checking [5, 21].

Once an erroneous behavior is found during verification, the next step is to localize the root
cause of the error. This step is called design debugging. Locating the root cause of an observed erroneous behavior on the output of a module, or in many cases a primary output, can be a difficult task because the engineer must search across both space (gates, RTL code) and time (clock cycles). This places a large burden on the engineer to thoroughly understand not only the design but also the temporal behavioral specifications. With the shift toward System-on-Chips (SOCs), designs can contain many different functional blocks and embedded memories, leading to a huge increase in the number of gates and clock cycles that need to be analyzed. This increase in size and complexity quickly becomes an unwieldy task for any engineer.

Design debugging has had a great deal of research in the past twenty years. Simulation-based techniques were among the earliest methods to localize errors in a design. They have been effective at localizing errors in certain situations but do not scale well to today’s industrial designs. More recently, design debugging based on formal frameworks using Boolean Satisfiability (SAT) and its related techniques have shown great promise in scaling towards industrial use. In these techniques, the debugging problem is formulated in
terms of a propositional SAT problem (or a related language) and given to a SAT solver. With the great advances in solving SAT instances [16, 29, 32, 42], these formulations allow for greater scalability and quality of results compared to earlier techniques.

Despite the great advances academia has produced in design debugging, most engineers in the industry still perform debugging manually, typically through a waveform viewer or some other graphical representation. One reason for this is that many of the academic solutions to debugging have not been able to scale to industrial problems. However, the industry has adopted several methods to aid designers in their manual debug effort.

Manual debugging tools [4] have gained wide-spread acceptance in the industry, helping engineers visualize the cause of the error and trace the erroneous behavior through time. More recently, the industry has been moving towards assertion-based verification [18]. This shift in methodology helps localize the error by placing assertions throughout the design, increasing the chances the observation point is closer to the error. In addition, assertions succinctly encode behavioral specifications helping the engineer understand the nature of the error. These methods aid the engineer in reducing the size of the debugging problem as well as trying to effectively understand and process it. However, both these methods still leave the job of localizing the error to be done manually by the engineer.

Debugging is one of the few steps in the VLSI design cycle that is still performed manually, creating a large bottleneck as other steps in the design flow become more and more efficient with the help of automation. This is why scaling automated design debugging methods to automatically localize errors remains of great interest to both the academic and industrial communities. By reducing the time spent on debugging with automated techniques, designers will be able to improve their time to market, increase their feature set and reduce the overall cost of creating VLSI designs.

1.2 Contribution

This thesis aims to scale automated debugging techniques in two important ways. The first contribution aims to tackle the problem of the growing size of modern designs, specifically
targeting designs with large embedded memories. These embedded memories give rise to an exponential increase in the state-space of a design, dramatically increasing the size and debugging complexity of the system. The second contribution aims to tackle the problem of the complexity of the errors, specifically scaling automated debugging techniques to handle large debugging problems involving many clock cycles. By addressing two key factors, the size of the design with embedded memories and the number of clock cycles, this thesis attempts to bring automated debugging techniques one step closer to solving practical industrial problems.

In summary, this work makes the following contributions:

- It introduces a succinct memory model for automated design debugging. By modelling embedded memories efficiently, the size and complexity of the debugging problem are greatly reduced, dramatically lowering the memory requirements for the debugging problem.

- It presents a scalable SAT-based design debugging algorithm that uses a mathematical technique called interpolation to divide the debugging problem into multiple parts across time, reducing the number of simultaneous clock cycles that need to be analyzed. This allows for each sub-problem to be solved in a shorter amount of time while greatly reducing the peak memory requirements of the debugging problem.

In the next few sub-section, both of these contributions are discussed in greater detail.

1.2.1 A Succinct Memory Model for Automated Design Debugging

Embedded memories greatly increase the complexity of a design by exponentially increasing the state space by the number of state bits. A naïve approach to modelling memory in design debugging involves representing each memory bit as a flip flop. This greatly increases the complexity and size of a design making the debugging problem much more difficult.

The first part of this thesis proposes a novel memory model for design debugging that avoids the need to represent each memory bit explicitly. This is done in a SAT-based design debugging framework where values from simulation of the erroneous circuit are used to produce a significantly smaller SAT encoding.
The major contributions of this part can be summarized as follows:

- The first succinct memory model for design debugging, which replaces the whole memory with a number of clauses that is linear in the trace length $k$ and a parameter $B$ of the memory model.
- An extension to this memory model to handle an arbitrary initial memory configuration.
- A method to handle non-cycle accurate design debugging with embedded memories, where trace output values are not available for each clock cycle and only the expected final state of memory is known.

An extensive set of experimental results on industrial designs with embedded memories demonstrate the benefit of this work. The proposed cycle accurate debugging scheme achieves a peak memory reduction of several orders of magnitude compared to an explicit memory representation. And the non-cycle accurate debugging results show a significant reduction in both run-time and peak memory when compared to an explicit memory model.

When compared with previous methods, a 96% average reduction in the memory model size is observed as well a 20% improvement in run-time. Non-cycle accurate debugging results show a 98% decrease in the size of the memory model and comparable run-times to the explicit state representation.

### 1.2.2 Scalable VLSI Debugging with Interpolation

A major contributor to the difficulty of the debugging problem is the number of clock cycles that need to be analyzed which can reach thousands of clock cycles in length \cite{park2021}. The second part of this thesis proposes a novel scalable SAT-based design debugging algorithm which leverages a mathematical technique called interpolation which over-approximate sets of constraints that model the erroneous behavior, significantly reducing the memory-intensive circuit replication at any given time. This is accomplished by dividing the error trace into several parts, or \emph{windows}, and analyzing each window of clock cycles separately. To allow for each window to be properly constrained with the erroneous behavior, interpolants are used to over-approximate sets of
constraints that model clock cycles within close proximity to the observed error. The analysis begins with a window at the end of the error trace. If the analysis does not yield complete results, it proceeds by moving the window backwards iteratively. The interpolant is calculated from the unsatisfiable (UNSAT) core resulting from previously analyzed windows.

The major contributions for this part can be summarized as follows:

- A scalable SAT-based design debugging algorithm that uses interpolants to over-approximate sets of constraints that model the erroneous behavior drastically reducing the number of simultaneous time-frames that need to be modelled.

- This method is proven to be complete in that it finds all error locations whose functions can be modified to correct the erroneous behavior for a given error trace and number of errors.

- An additional technique to generate multiple interpolants is introduced to reduce the number of error locations returned, thus improving the quality of the debugging results.

An extensive set of experiments on industrial hardware designs and long error traces illustrates the benefits of this work. These experiments show that there is favourable trade-off between the quality of the debugging results, confirming the efficacy of the technique for scaling of existing SAT-based debugging methodologies to handle modern VLSI designs.

In particular, it is shown that a conservative partitioning of the error trace yields an average 34% reduction in memory and 24% reduction in run-time compared to traditional SAT-based debugging, while the number of returned error locations is only increased on average by 1% of the total number of suspects. For a more aggressive partitioning scheme, averages of 57% reduction in memory and 23% reduction in run-time are achieved at the cost of increasing the relative number of error locations returned by 2%.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides background material on verification and debugging as well as other necessary concepts used in this thesis. Chapter 3 presents a succinct
memory model for automated design debugging that dramatically reduces the peak memory requirements for using embedded memories. A novel automated design debugging algorithm is presented in Chapter 4 that uses interpolants to partition the problem into smaller subproblems. Chapter 5 presents the experimental results of the contributions and Chapter 6 concludes this work and discusses future directions.
Chapter 2

Background

2.1 Introduction

This chapter presents background material that is relevant to the contribution in this thesis. We assume the reader is familiar with basic Boolean Satisfiability concepts, otherwise please refer to Section 2.5 for an overview of the topic.

The rest of the chapter is organized as follows. Sections 2.2 and 2.3 discuss preliminaries on verification and debugging. Next, background information on memory semantics is presented in Section 2.4 that are necessary to understand the first contribution. Section 2.5 discusses the fundamentals of Boolean Satisfiability that are necessary to understand the concepts in this thesis. Section 2.6 discusses a mathematical technique that is used in one of the contributions called interpolation. Section 2.7 summarizes the basic SAT-based debugging formulation. Finally, Section 2.8 summarizes this chapter.

2.2 Verification Overview

The aim of functional verification is to determine whether the implementation of a design conforms to its specification. Although many different types of errors exist, this thesis addresses functional errors at the RTL. Specifications such as power consumption and timing errors are not considered. As such, the following definition of an error is used:
**Definition 1** An error is at least one discrepancy between a primary output of a design and the specification of a design when the same initial state and primary input stimulus are applied.

A discrepancy takes the form of conflicting Boolean values at the primary outputs of a design. Although this definition explicitly mentions primary outputs, it can be applied to any type of observable error in a sequential design by simply taking an arbitrary observation point and creating a primary output.

In verification and debugging, it is necessary to model the behavior of a sequential circuit. This can be done using a method called the Iterative Logic Array (ILA), also known as *time-frame expansion*. Time-frame expansion models a sequential circuit by extracting out the combinational part of a circuit and replicating it for the number of clock cycles that need to be modelled.

To extract the combinational component, the current-state variables are treated as pseudo-inputs, the next-state variables are treated as pseudo-outputs and the state elements are removed. Next, the combinational component is replicated such that the next-state variables of time-frame $i$ are connected to the current-state variables of time-frame $i + 1$. The following example demonstrates this fact.

**Example 1** Figure 2.1 shows how to generate a time-frame expanded circuit. Figure 2.1(a) shows the original sequential circuit. First, the combinational component is separated from the sequential components by extracting the flip-flops, this is shown in Figure 2.1(b). The pseudo-inputs and pseudo-outputs are implicitly shown to be wires coming out of the dotted box. Next, the combinational component is replicated into a two time-frame expanded circuit shown in Figure 2.1(c).

After verification fails and an erroneous behavior in a design is found, an error trace or counter-example is produced describing the initial state, input and expected output that demonstrates the error. The error trace contains precisely the information needed to excite and observe the erroneous behavior. By simulating the erroneous design with the initial state and primary inputs of an error trace, one can observe a mismatch between the generated primary outputs.
Figure 2.1: Time-frame expanded example
and the expected ones in the error trace i.e. an erroneous behavior. We formally define an error trace here.

**Definition 2** An error trace, \( V^k_0 \), of length \( k+1 \), for clock-cycles \( 0 \) to \( k \) consists of an initial state predicate \( (S^0) \), a vector of primary input predicates \( (X^0, \ldots, X^k) \) and a vector of correct or expected primary output predicates \( (Y^0, \ldots, Y^k) \) from 0 to \( k \), all of which can be written as follows:

\[
V^k_0 = (S^0, X^0, \ldots, X^k, Y^0, \ldots, Y^k)
\] (2.1)

A subsequence, or window, of the error trace can be similarly defined as follows.

**Definition 3** A window of an error trace for clock-cycles \( p \) to \( q \), is defined as a consecutive subsequence of clock cycles containing an initial state predicate for clock cycle \( p \) and a vector of primary input and correct output predicates for clock cycles \( p \) to \( q \) written as follows:

\[
V^q_p = (S^p, X^p, \ldots, X^q, Y^p, \ldots, Y^q)
\] (2.2)

Where \( S^p \) is calculated by applying the initial state predicate and the first \( p \) primary input predicates to the transition relation, i.e. simulating the erroneous circuit for \( p \) cycles.

Using this notation, a prefix window of length \( p \) for this trace can be written as \( V^{p-1}_0 \) and a suffix window of length \( k - p + 1 \) can be written as \( V^k_p \). We will occasionally omit the term window and use the term suffix or prefix in place of suffix window or prefix window respectively.

A cycle accurate error trace is one where expected primary output values are defined for each clock cycle. If this is not the case, an error trace is said to be non-cycle accurate.

For this work, it is assumed that the error is first observed in the last clock cycle of the error trace. If this is not the case, a shorter error trace can be trivially generated by taking the shortest prefix that exhibits the erroneous behavior.

**Example 2** For the time-frame expanded circuit in Figure 2.1(c), the following is an error trace demonstrating an erroneous behavior in the circuit:

\[
V^1_0 = (x^0_0, (\overline{x^0_0} \land \overline{x^1_1}, x^1_0 \land x^1_1), (\overline{y^0_0}, \overline{y^1_0}))
\]
Notice that in the second time-frame, the erroneous circuit generates $y_0$ which is incorrect according to $V_0$. In addition, the prefix window $V_0^0 = \langle s_0^0, \langle x_0^0 \land x_1^0 \rangle, \langle y_0^0 \rangle \rangle$ and suffix window $V_1^1 = \langle s_1^1, \langle x_0^1 \land x_1^1 \rangle, \langle y_1^1 \rangle \rangle$ can be generated from this error trace.

### 2.3 Debugging Overview

In this thesis, the term debugging is synonymous with the definition of functional design debugging and fault diagnosis that is common in literature [1, 37]: the process of identifying the source(s) of errors in a given erroneous design.

The source of error can be any component (e.g. gates, RTL source code, design modules) depending on the implementation of the design. We use the generic term suspect to specify such potential locations that can be the cause of the error as seen in the next definition.

**Definition 4** A component with output function $f$ is a suspect with respect to an error trace, if and only if there exists another function $g$ that can remove the discrepancy on the primary output in the error trace.

It should be noted that a suspect is not necessarily the actual error that was introduced into the design. Instead, it may be an equivalent location [23, 40] that may also be changed to fix the observed error. Each one of these equivalent locations, as far as the error trace is concerned, is correct. However, only one of them may be the actual location that correctly fixes the error according to the full specification.

Debugging starts after an error trace is produced by a verification tool demonstrating some erroneous behavior in the design. This is shown in Figure 2.2. The verification tool take as input the design under test (DUT) as well as the high level specification. If it fails and finds an erroneous behavior, an error trace is generated. The error trace is typically produced by a verification tool such as a simulation testbench or a model checker. The debugging process takes the error trace as well as the DUT and returns suspects as its output.

An automated design debugging algorithm aims to find all suspects, or groups of suspects,
Figure 2.2: Typical debugging flow
that can explain the discrepancy in the given error trace. However, since the complexity of the debugging problem grows exponentially with respect to the number of errors \[40\], automated debugging algorithms typically limit the number of errors with an error cardinality. The error cardinality, denoted as \( N \), is the number of distinct suspect components that are contained in a solution set returned by an automated debugging algorithm.

Using the previous ideas, this thesis defines a complete a design debugging algorithm as follows.

**Definition 5** An automated design debugging algorithm is said to be complete for a given error trace and error cardinality, if and only if it returns all suspects (or groups of suspects) that can fix the discrepancy in the error trace.

The above definition ensures that any complete algorithm returns all the suspects which could be responsible, but it does not preclude the addition of spurious suspects. This means that even though a complete algorithm will find all equivalent errors, even if additional suspects are found that may not fix the observed error in the trace, an algorithm is still considered to be complete using the above definition. The quality of the debugging results can be quantitatively measured by the number of suspects returned. The term resolution is used to qualitatively describe the total number of suspects returned, where fewer suspects correspond to better resolution.

### 2.4 Embedded Memory Semantics

A simple model of a dual-port embedded memory is shown in Figure 2.3. The embedded memory is connected to the design through several interface signals. A thin line (lowercase variable) indicates a single bit signal and a bold line (uppercase variable) indicates a bus signal. At any given clock cycle, both a write and a read from memory can occur. Write Data (\( WD \)) can be written to memory at Write Address (\( WA \)) when signal write enable (\( we \)) is set to 1 and the data written becomes available to read from in the next clock cycle. Similarly, Read Data (\( RD \)) can be read from memory at Read Address (\( RA \)) if signal read enable (\( re \)) is set to 1. The read data becomes available in the current clock cycle. A subscripted variable (e.g.
$re_p$ indicates the value of that variable during the subscripted time-frame (e.g. time-frame $p$). Note that this simple model of memory can be extended to handle more complex models by adding additional latencies to when the read and write data appear at the outputs.

The complete memory semantics can be described in terms of these memory interface signals as follows. The data read from memory at time-frame $q$ ($RD_q$) is equal to the data written to memory at time-frame $p$ ($WD_p$), where $q > p$, if the following conditions are met:

(a) The read and write are done from and to the same memory address (i.e. $WA_p = RA_q$).

(b) The write enable is active at time-frame $p$ (i.e. $we_p = 1$).

(c) The read enable is active at time-frame $q$ (i.e. $re_q = 1$).

(d) There are no other time-frames in between during which the same address is written to.

When this situation occurs, the data is said to be forwarded from $p$ to $q$, or there exists a forwarding path $p \rightarrow q$, or that time-frame $q$ is reading from time-frame $p$, or equivalently, time-frame $p$ is writing to time-frame $q$. The writing time-frame $p$ is occasionally referred to as the data forwarding source.

This can be formally expressed with the following set of constraints:

$$\bigwedge_{p=1}^{k} \bigwedge_{q=p+1}^{k} \left\{ \left( (WA_p = RA_q) \land we_p \land re_q \land \bigwedge_{l=p+1}^{q-1} ((WA_l \neq RA_l) \lor we_l) \right) \rightarrow (WD_p = RD_q) \right\}$$

(2.3)
2.5 Boolean Satisfiability

This section briefly describes relevant background knowledge and terminology of Boolean Satisfiability (SAT) for this thesis. The following is a formal definition of the Boolean Satisfiability problem.

Definition 6 Given a propositional logic formula $\Phi$ with $n$ Boolean variables $v_1, v_2, ..., v_n$, the Boolean Satisfiability problem asks whether there exists an assignment to its variables that evaluates $\Phi$ to 1 or whether no such assignment exists. If such an assignment exists then the problem is said to be satisfiable (SAT) otherwise unsatisfiable (UNSAT).

The Boolean Satisfiability problem has great historical importance as being the first problem to be classified as NP-Complete [11]. It also has great practical importance as many problems can be modelled as a SAT instance. This is especially true in VLSI CAD problems such as circuit optimization and routing. Recent advances in modern SAT solvers have allowed them to solve industrial instances with millions of variables and clauses efficiently, making them a popular tool for solving many VLSI problems [16, 32].

2.5.1 CNF Representation

Modern SAT solvers accept SAT instances in conjunctive normal form (CNF), which is a conjunction of clauses, each made up of a disjunction of literals. The following example shows a simple SAT problem in CNF.

Example 3 The following is a Boolean formula written in CNF.

$$\Phi = (a \lor \bar{b}) \land (\bar{a} \lor \bar{b} \lor \bar{c}) \land (b \lor \bar{c})$$  \hspace{1cm} (2.4)

$\Phi$ is SAT because $\{a = 1, b = 0, c = 0\}$ is a satisfying assignment evaluating $\Phi$ to 1.

Logic circuits can be efficiently converted into CNF in linear time [24] by introducing auxiliary variables for each net in the circuit. Table 2.1 shows how common logic gates can be converted to CNF.
<table>
<thead>
<tr>
<th>Gates</th>
<th>CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = \text{AND}(x_1, x_2, ..., x_n)$</td>
<td>$(x_1 + \overline{y})(x_2 + \overline{y})...(x_n + \overline{y})$</td>
</tr>
<tr>
<td></td>
<td>$(\overline{x_1} + \overline{x_2} + ... + \overline{x_n} + y)$</td>
</tr>
<tr>
<td>$y = \text{NAND}(x_1, x_2, ..., x_n)$</td>
<td>$(x_1 + y)(x_2 + y)...(x_n + y)$</td>
</tr>
<tr>
<td></td>
<td>$(\overline{x_1} + \overline{x_2} + ... + \overline{x_n} + \overline{y})$</td>
</tr>
<tr>
<td>$y = \text{OR}(x_1, x_2, ..., x_n)$</td>
<td>$(\overline{x_1} + y)(\overline{x_2} + y)...(\overline{x_n} + y)$</td>
</tr>
<tr>
<td></td>
<td>$(x_1 + x_2 + ... + x_n + y)$</td>
</tr>
<tr>
<td>$y = \text{NOR}(x_1, x_2, ..., x_n)$</td>
<td>$(\overline{x_1} + \overline{y})(\overline{x_2} + \overline{y})...(\overline{x_n} + \overline{y})$</td>
</tr>
<tr>
<td></td>
<td>$(x_1 + x_2 + ... + x_n + y)$</td>
</tr>
<tr>
<td>$y = \text{XOR}(x_1, x_2)$</td>
<td>$(\overline{x_1} + x_2 + y)(x_1 + \overline{x_2} + y)$</td>
</tr>
<tr>
<td></td>
<td>$(x_1 + x_2 + \overline{y})(\overline{x_1} + \overline{x_2} + \overline{y})$</td>
</tr>
<tr>
<td>$y = \text{XNOR}(x_1, x_2)$</td>
<td>$(\overline{x_1} + x_2 + \overline{y})(x_1 + \overline{x_2} + \overline{y})$</td>
</tr>
<tr>
<td></td>
<td>$(x_1 + x_2 + y)(\overline{x_1} + \overline{x_2} + y)$</td>
</tr>
<tr>
<td>$y = \text{NOT}(i)$</td>
<td>$(i + y)(\overline{i} + \overline{y})$</td>
</tr>
<tr>
<td>$y = \text{BUFFER}(i)$</td>
<td>$(i + \overline{y})(\overline{i} + y)$</td>
</tr>
<tr>
<td>$y = \text{MUX}(s, x_1, x_2)$</td>
<td>$(x_1 + \overline{y} + s)(\overline{x_1} + y + s)$</td>
</tr>
<tr>
<td></td>
<td>$(x_2 + \overline{y} + s)(\overline{x_2} + y + s)$</td>
</tr>
</tbody>
</table>

Table 2.1: Simple gates and their CNF representation

Besides simple logic gates, many higher level constraints in CNF can be generated using circuit constructs. One commonly used constraint is a cardinality constraint which ensures that for a set of variables, exactly $N$ variables are set to 1. One space efficient way to implement this is shown in Figure 2.4, with a tree of full adders that feed into a comparator. This encoding has a space of $O(n)$ in the number of variables input into the cardinality constraint [37]. The tree of adders and comparator can be formulated in CNF by using simple logic gates from Table 2.1.
2.5.2 SAT Algorithms

Most modern SAT solvers use a variation of the search-based Davis Putnam Logemann Loveland (DPLL) algorithm \[13\]. The basic form of this algorithm is shown in Algorithm 2.1. The SAT problem can be thought of as a binary tree where each level represents a variable, each internal node represents a partial assignment, and each edge represents an assignment to true or false. The DPLL algorithm can be thought of as doing a depth first search along this tree.

At each step of the algorithm, it performs two checks. One to see if there are any unsatisfied clauses and returns UNSAT immediately. The other checks to see if all the clauses are satisfied and returns SAT. Otherwise, it will pick a variable that has not been assigned yet and recursively call itself to explore another node in the tree.

The algorithm will eventually explore every possible branch making this algorithm complete. However due to this fact, in the worst case this algorithm will take an exponential amount of time with respect to the number of variables. Despite this worse case behavior, modern advances have greatly enhanced the capacity of SAT solvers to efficiently handle SAT instances with millions of variables and clauses.

Modern solvers use a variation of the DPLL algorithm that introduce improvements to three
Algorithm 2.1 The DPLL algorithm

1: procedure DPLL\_SAT(Partial Assignment A)
2:   if contains\_unsatisfied\_clauses() then
3:      return UNSAT
4:  else if all\_clauses\_satisfied() then
5:     return SAT
6:  else
7:     v ← next\_unassigned\_variable()
8:     return DPLL\_SAT(A ∪ (v=1)) ∧ DPLL\_SAT(A ∪ (v=1))
9:  end if
10: end procedure

major areas of the algorithm: decision heuristics, Boolean constraint propagation and conflict analysis.

The first improvement is to the order in which variables are selected, also known as decision heuristics. Due to the search nature of the DPLL algorithm, the order in which variables are selected becomes extremely important. Most modern solvers use a variation of the variable state independent decaying sum (VSIDS) decision heuristic which was used in zChaff\[32\]. This heuristic gives preference to literals that appear most often in the SAT problem while giving heavier weight to recently discovered information. One important aspect of this heuristic is that it keeps running totals of the literal counts which can be computed very quickly.

The second improvement relates to the process of determining assignments to variables that are implied by the current partial assignment. It relates to the generation of unit clauses from the current partial assignment. A unit clause is produced if in the current partial assignment, all but one literal evaluates to zero, and the remaining literal is unassigned. For this type of clause, the remaining literal must be set to true or else the clause would not be satisfied. However, setting the literal to true may cause other clauses to become unit clauses and the process can repeat.
Example 4 Consider the SAT instance with the partial assignment $a = 0$:

$$(a \lor b) \land (b \lor \overline{c}) \land (\overline{c} \lor \overline{d})$$

Since $a = 0$, the clause $(a \lor b)$ become a unit clause and we can imply that $b = 1$. But this makes $(b \lor \overline{c})$ a unit clause, implying that $c = 0$. This in turn, implies that $d = 0$. All clauses are now satisfied, so the instance is SAT by only deciding on one variable and running Boolean constraint propagation.

Boolean constraint propagation allows huge amounts of the search tree to be pruned. The authors of [32] estimate that 90% of the run-time is spend inside this one routine. Other anecdotal evidence suggests a higher percentage. Major improvements to Boolean constraint propagation involve engineering efficient implementations by choosing the right data structures and updating algorithms.

The third major improvement relates to the way in which modern algorithms handle a conflict in Boolean constraint propagation. When a conflict arises, in the basic DPLL algorithm, the search backtracks one level up the tree and tries the other branch. A major contribution in this area involves a smarter way to backtrack which "learns" based on the decisions and implications made up to this point in the algorithm. It works by building an implication graph that shows how variables and clauses interact to derive a conflict. From this graph, it is possible to backtrack non-chronologically by determining the most-recent cause of the current conflict which is not necessarily the most recent decision. Conflict analysis and non-chronological backtracking have shown to be very effective in pruning out large parts of the search space resulting in a major improvement to modern SAT solvers.

Algorithm 2.2 shows the DPLL variant that most modern SAT solvers use. Three subroutines corresponding to the major improvements mentioned above are shown in the algorithm: \texttt{decide()} corresponds to decision heuristics, \texttt{bcp()} corresponds to Boolean constraint propagation and \texttt{analyze.conflict()} corresponds to conflict analysis and non-chronological backtracking.

The algorithm first assigns a value to the next unassigned variable on line 3. If there are
Chapter 2. Background

Algorithm 2.2 Modern SAT algorithm

1: procedure MODERN_SAT
2:     while SAT do
3:         if !decide() then
4:             // No unassigned variables
5:             return SAT
6:         end if
7:         while !bcp() do
8:             // Conflict occurred during bcp()
9:             if !analyze_conflict() then
10:                return UNSAT
11:         end if
12:     end while
13: end while
14: end procedure

no more unassigned variables, the problem is SAT. Otherwise, it performs Boolean constraint propagation seen on line 7. If a conflict occurs and results in an unsatisfiable instance, it returns UNSAT. Otherwise, it will learn new clauses and continue on with the algorithm.

A variation of a SAT solver is an all-solution SAT solver which will find all satisfying assignments to a SAT instance. A regular SAT solver can be easily extended by adding a blocking clause which is a disjunction of the negation of all the assigned literals. This prevents the same solution from being found again. After this solution is added, the solver is run again to find the next solution. When the solver returns UNSAT, all solutions have been found.

An important practical point to consider is that in many cases, a full assignment to all of the variables in a SAT instance is not necessary. Instead, only certain variables are regarded as important. In these cases, the blocking clause should contain only the literals from the important variables. Many modern solvers have this feature built in to allow use with practical problems.

Modern SAT solvers all use this same basic algorithm with modifications to the three main
subroutines. A great deal of the challenge of speeding up SAT solvers is in the implementation which can greatly vary the performance even with the techniques listed here.

2.5.3 Unsatisfiable Cores

An UNSAT core $U$ is a subset of clauses that is unsatisfiable in an UNSAT propositional Boolean formula written in CNF. Using the resolution rule, clauses in an UNSAT core can be combined to imply the empty clause. The resolution rule in propositional logic takes two clauses with complimentary literals of the same variable and produces a new clause implied by the original two by a disjunction of all the other literals in the original clauses. The following example illustrates the resolution rule.

**Example 5** In the following equation $a$ and its compliment exists in the two original clauses. A third clause can be implied which takes a disjunction of all the other literals in the original clauses to produce the new one.

$$(a \lor b) \land (\bar{a} \lor c) \rightarrow (b \lor c)$$

A resolution graph demonstrates how clauses in an UNSAT core can be combined using the resolution rule to generate the empty clause. This serves as a proof of unsatisfiability of a SAT instance. The root nodes of the graph are the original clauses, the intermediate nodes correspond to implied or learnt clauses of the SAT instance and the leaf node is the empty clause. Modern DPLL SAT solvers can generate a proof of unsatisfiability along with a corresponding resolution graph that shows that a SAT instance is unsatisfiable [43]. The next example demonstrates these concepts.

**Example 6** The following is an UNSAT Boolean formula written in CNF.

$$\Phi = d \land (a \lor b) \land (\bar{a} \lor \bar{b}) \land (a \lor \bar{c}) \land (\bar{a} \lor c) \land (b \lor \bar{d}) \land (c \lor \bar{d}) \land (\bar{b} \lor \bar{c} \lor d)$$

An UNSAT core, $U$, of $\Phi$ is: $d \land (a \lor b) \land (\bar{a} \lor \bar{b}) \land (a \lor \bar{c}) \land (\bar{a} \lor c) \land (b \lor \bar{d}) \land (c \lor \bar{d})$. By resolving these clauses together, a resolution graph can be generated as shown in Figure 2.5(a). Each root node in the graph represents an original clause in $U$, while each intermediate node represents a clause that
can be implied, or learned from the original clauses. The UNSAT core implies the empty clause resulting an UNSAT instance.

2.6 Interpolants

An interpolant \[12\] is a Boolean formula that can be generated from an UNSAT core. For a given unsatisfiable formula whose clauses can be partitioned into two subsets, \(A\) and \(B\), an interpolant is a formula, \(P\), with the following properties:

(a) \(A \rightarrow P\)

(b) \(B \land P\) is unsatisfiable.

(c) \(P\) only contains common variables of \(A\) and \(B\).
An interpolant can be thought of as an over-approximation of $A$ that retains the information that made the original instance UNSAT. This can be seen when considering two facts. The first is that the interpolant only involves the common variables of $A$ and $B$, so it can be thought as of a projection of the portion of the UNSAT core from $A$ onto the common variables. Secondly, it is derived primarily from clauses that were originally in $A$ that relate directly to the UNSAT core, so it can be thought of as an over-approximation of $A$ that uses only the relevant information from the UNSAT core. Interpolants have been widely used in many VLSI CAD applications such as model checking applications [30] and more recently in synthesis applications [31].

There exists an algorithm [30] that can generate an interpolant as a Boolean circuit whose gates correspond to the vertices in the resolution graph and whose inputs correspond to the common variables. The pseudo-code for the algorithm is shown in Algorithm 2.3. For each node in the resolution graph, a gate is generated based on whether it is a root or intermediate node and how it is related to the clauses in $A$. If the node is a root node and in $A$, it creates an OR gate of the common literals (line 7). For other root nodes, that net is 1 (line 9). For the intermediate nodes, if the resolvent variable is not a common variable and in $A$, then an OR gate is used to connect the two predecessor nets (line 15). Otherwise, an AND gate is used (line 17). The gate that is generated from the leaf node in the resolution graph corresponds to the single output of the Boolean circuit that is generated (line 21). Setting the output to 1 ensures that the interpolant constrains the problem correctly.

This algorithm takes $O(V + L)$ time, where $V$ is the number of vertices in the resolution graph and $L$ is the total number of literals in the proof. However, in the worst case, the size of the resolution graph can be exponential in the size of the problem.

**Example 7** The SAT instance in Example 6 can be divided into two sets $A$ and $B$ as such:

\[
A = d \land (b \lor \bar{d}) \land (c \lor \bar{d}) \land (\bar{b} \lor \bar{c} \lor d)
\]

\[
B = (a \lor b) \land (\bar{a} \lor \bar{b}) \land (a \lor \bar{c}) \land (\bar{a} \lor c)
\]

(2.5)

Figure 2.5(b) shows the interpolant that is generated from the resolution graph from Example 6. The steps from Algorithm 2.3 for each node of the resolution graph are listed as follows:
• $d$: This clause belongs to $A$ but does not contain any common variables, so having no inputs to an OR gate results in a constant $0$.

• $(b \lor \bar{d})$: This clause belongs to $A$ and has one common variable $b$. The result of this net is just $b$.

• $b$: The resolvent variable is $d$ which belongs to $A$ and is not a common variables, so an OR is generated from the predecessor nets.

• $(\bar{a} \lor \bar{b})$: This clause belongs to $B$, so the result is a constant $1$.

• $\bar{a}$: The resolvent variable is $b$ which is a common variable so an AND gate is used.

• $(a \lor \bar{c})$: This clause belongs to $B$, so the result is a constant $1$.

• $\bar{c}$: The resolvent variable is $a$ which is only in $B$, so a AND gate is used.

• $(c \lor \bar{d})$: This clause belongs to $A$ and contains the common variable $c$. The result of this net is just $c$.

• $(\bar{d})$: The resolvent variable is $c$ which is a common variable so an AND gate is used.

• $d$: This clause belongs to $A$ but does not contain any common variables, so having no inputs to an OR gate results in a constant $0$.

• $()$: The resolvent variable is $d$ which belongs to $A$ and is not a common variables, so an OR is generated from the predecessor nets.

As seen above, each root node in $B$ is essentially replaced with a constant $1$. However, the root nodes in $A$ may play a role if they contain common variables. This allows the interpolant to contain information in the UNSAT core that relate to clauses in $A$. Many of the root nodes evaluate to constants so the resulting interpolant can be optimized to generate a concise over-approximation of the clauses in $A$ from the UNSAT core.
2.7 SAT-based Debugging

This section briefly describes background and notation for SAT-based design debugging that is relevant to our contribution. SAT-based design debugging [37] is a complete method that encodes the design debugging problem into a SAT instance for a given error trace and number of errors. The satisfying assignments of the SAT instance correspond to suspects which can be replaced with non-deterministic functions to correct the erroneous behavior in the error trace.

The SAT instance is created in several steps. First, the transition relation is enhanced by introducing a set of suspect variables, $E = \{e_0, \ldots, e_n\}$, where each $e_i$ corresponds to the $i^{th}$ potential error location. The suspect variables are then added to the transition relation such that if $e_i = 1$ then the $i^{th}$ potential error location is disconnected from its fan-in and become free variables. This can be achieved either through a hardware construction using multiplexors, an if-then-else construct or directly in conjunctive normal form (CNF). The enhanced transition relation is denoted by $T_{en}(s^i, s^{i+1}, x^i, y^i, E)$.

Next, $T_{en}$ is unrolled as a time-frame expanded model for the length of the error trace, such that the next-state of time-frame $i$ is connected to the current-state of time-frame $i+1$. Note that the suspect variables are not replicated since they represent the same location regardless of the time-frame. The error trace predicates are then applied to the initial state, input and output variables of the replicated enhanced transition relation.

Finally, the number of simultaneous active suspect variables, denoted as the error cardinality, is constrained to a given constant $N$ using cardinality constraints $\Phi_N(E)$ which can be generated from a network of adders [37]. Given an error trace $V^k_0$ or a window of an error trace $V^q_p$, design debugging can encoded by the following SAT problems respectively:

$$
\text{Debug}^k_0 = S^0(s^0) \land \Phi_N(E) \land \\
\left( \bigwedge_{i=0}^{k} X^i(x^i) \land Y^i(y^i) \land T_{en}(s^i, s^{i+1}, x^i, y^i, E) \right) \tag{2.6}
$$

$$
\text{Debug}^q_p = S^p(s^p) \land \Phi_N(E) \land \\
\left( \bigwedge_{i=p}^{q} X^i(x^i) \land Y^i(y^i) \land T_{en}(s^i, s^{i+1}, x^i, y^i, E) \right) \tag{2.7}
$$
Note that for $N = 0$, $\text{Debug}_0^k$ is UNSAT, since the error trace applied to the erroneous design without any active error suspect variables cannot produce the correct outputs defined in the error trace.

In a satisfying assignment of Equation 2.6, each active suspect variable corresponds to a possible component (gate, module etc.) whose function can be changed to correct the erroneous behavior. To find all such suspects, for each satisfying assignment, a blocking clause is added to the debugging instance to block the active suspect variables from appearing again as a satisfying assignment. This instance is then sent again to the solver. When the solver eventually returns UNSAT, all possible suspects have been found.

**Example 8** Figure 2.6 shows a two time-frame expanded circuit of an erroneous two gate design with one state element. The suspect variables $\{e_1, e_2\}$ are denoted as enables on the side of each gate. The incorrect gate is $g_2$ which should be a buffer instead of an inverter. The error trace:

$$\mathcal{V}_0^1 = \langle s_0^0, (x_0^0 \land x_2^0, x_1^1 \land x_2^1), (y_1^1 \land y_2^1) \rangle$$

demonstrates an erroneous behavior of the circuit. For $N = 1$, a satisfying assignment for the suspect variables $\{e_1, e_2\}$ is $\overline{e_1} \land e_2$. Adding the blocking clause $\overline{e_2}$ to the problem causes it to be UNSAT. This implies that $g_2$ is potentially the only gate that can be modified to correct the erroneous behavior.

### 2.8 Summary

This section presented background material necessary for understanding the contributions of this thesis. First, definitions and concepts about verification and debugging were presented. Next, a brief discussion on memory semantics was discussed which is relevant to the first contribution of this thesis. This was followed by an introduction to the fundamentals of Boolean Satisfiability which are used heavily utilized in this thesis. After that, a discussion on a mathematical technique called interpolation was introduced. Finally, a description of SAT-based debugging was given which serves as a basis for the contributions described in this thesis.
Figure 2.6: SAT-based debugging
Algorithm 2.3 Generating an interpolant

1: \( G := \) resolution graph

2: \( P := \) interpolant

3: procedure \( \text{GENERATEINTERPOLANT}(G) \)

4: \hspace{1em} for vertex \( c \in G \) do

5: \hspace{2em} if \( c \) is a root node then

6: \hspace{3em} if \( c \in A \) then

7: \hspace{4em} \( \text{net}(c) = \text{OR}(\text{common literals of } c) \)

8: \hspace{3em} else

9: \hspace{4em} \( \text{net}(c) = 1 \)

10: \hspace{2em} end if

11: \hspace{1em} else

12: \hspace{2em} \( c_1, c_2 := \) predecessors of \( c \)

13: \hspace{2em} \( v := \) resolvent variable used to imply \( c \)

14: \hspace{2em} if \( v \) is not a common variable and only appears in \( A \) then

15: \hspace{3em} \( \text{net}(c) = \text{OR}(c_1, c_2) \)

16: \hspace{3em} else

17: \hspace{4em} \( \text{net}(c) = \text{AND}(c_1, c_2) \)

18: \hspace{3em} end if

19: \hspace{2em} end if

20: \hspace{1em} end for

21: return \( \text{net}(\text{leaf node}) \)

22: end procedure
Chapter 3

A Succinct Memory Model for Automated Design Debugging

3.1 Introduction

With the shift towards System-on-Chips (SoCs), embedded memories are becoming prevalent in a variety of different designs. This makes the tasks of verification and debugging much more difficult because embedded memories give rise to an exponential increase in the state-space of a design, dramatically increasing the complexity of the system. This compounds the state-space explosion problem which already limits the practical application of many verification tools [10].

This chapter presents a succinct memory model for embedded memories in a SAT-based debugging framework. It aims to reduces the number of constraints needed to model embedded memories in the SAT problem. The chapter is organized as follows. Section 3.2 describes how embedded memories are modelled as well as some previously proposed work in the area. Section 3.3 presents a detailed description of how to construct the proposed memory model for a general case cycle accurate debugging instance. Section 3.4 describes an extension to handle an arbitrary initial memory configuration. Section 3.5 describes another extension to handle non-cycle accurate debugging with a final memory configuration and finally Section 3.6 concludes the chapter.
3.2 Memory Models and Previous Work

A memory model for a SAT instance can be any set of constraints that obey the embedded memories semantics in Equation 2.3. This will involve keeping the control and interface signals since they are required by the memory semantics and connect to the main part of the design. However, how the memory array is modelled can vary depending on the specific memory model.

The explicit approach for modelling memory consists of representing each bit of memory as a state bit (flip-flop) and modelling the necessary address decoding and control circuitry. This model is inefficient in many cases for two reasons. First, the address and data decoding circuitry typically require a large multiplexor tree which can be prohibitively large for typical embedded memory sizes. Second, SAT instances typically are only modelled for a relatively small number of time-frames when compared to the number of addresses in the memory. Combined with the fact that only a fixed number of memory accesses are done in each time-frame, most of the memory locations will not even be used in a typical SAT instance. This results in a large amount of constraints being unnecessarily added. These reasons typically render the problem of using explicit models intractable even for modest sized memories.

A memory model for Bounded Model Checking (BMC) is described in [20]. In this thesis, the SAT-based memory model of [20] is referred to as the crossbar model. The crossbar model removes the memory array but maintains the control and interface signals. Constraints are then added to each time-frame of the SAT instance to ensure that the memory semantics are preserved according to Equation 2.3. These constraints give the ability to forward any \( WD_p \) to any future \( RD_q \) such that \( q > p \) while preserving the memory semantics. [20] adds a forwarding path between each time-frame and all its previous time-frames resulting in a number of CNF clauses that grows quadratically \( (O(k^2)) \) with the number of unrollings \( k \).

More recently, [28] presents a set of memory abstraction algorithms for use in the verification of RTL models that must be specified within their tool-specific language. This technique abstracts memory to a manageable size and allows comparison of equality between different
memory modules. Where a comparison of memories is not needed, this abstraction gives comparable savings to previous work.

Although these memory models significantly reduce the size of the state-space compared to the naïve approach, the number of required constraints to model memory grows quadratically in the size of the unrolling $k$. This can lead to excessive memory requirements for large error traces.

### 3.3 A Succinct Memory Model for Debugging

As designs with embedded memories become more prevalent, the need for scalable memory models for design debugging becomes crucial. Since SAT-based design debugging involves replicating the circuit many times, having an explicit state representation quickly becomes infeasible. Larger trace lengths also pose a problem for memory models with a space complexity of $O(k^2)$. In this section, a memory model that is asymptotically more efficient in the trace length is introduced. A unique feature of the proposed method is that it makes use of the simulated values in the erroneous circuit to guide the debugging process. The given formulation is for a simple dual-port memory but can be easily extended to a generic multi-port memory.

The proposed memory model requires a setup step as well as a modelling step. The setup step requires the values of the control interface signals ($WA, we, RA, re$) for each clock cycle when simulating in the error trace with the DUT. This can be obtained by simulating the initial state and stimulus from the error trace with the erroneous circuit and observing the required values. The modelling step, like other memory models, requires the control and interface signals to be kept and the memory array to be replaced by a set of CNF constraints to be added to the debugging instance. These constraints will allow memory to be forwarded according to the memory semantics.

The memory model works by providing forwarding paths for data to move across timeframes. Two key ideas are used in this memory model. The first idea is that although the design can access any part of memory, it can only read or write once per time-frame. This
allows the model to limit the number of forwarding paths that need to be considered. The second idea is that many times the error does not affect the memory accesses in the error trace, so values from the simulated design with the error trace can be used. This idea uses the fact that the debugging problem can only deviate a certain amount from the simulation of the erroneous circuit, which allows the model to reduce even further the number of forwarding paths needed to model memory.

The constraints added to the debugging instance allow two types of data forwarding across time-frames:

(a) Forwarding according to the simulation of the design in the error trace.

(b) Forwarding through newly introduced buses, used to fix memory access errors.

The intuition behind the use of a simulated forwarding path in the error trace lies in the fact that many design bugs do not affect memory accesses. In fact, if the error does not excite a change in memory read/write addresses for a given error trace, then constraining the read/write memory accesses to the ones observed in simulation will still result in a valid debugging instance. This is because the error is excited only on the data path, so changes to the read/write memory addresses are unnecessary.

On the other hand, if the error affects the memory addresses, a parameterizable number of extra buses $B$ are introduced, which allow for changes to the memory accesses observed during the simulation of the error trace. This gives the SAT solver the flexibility to choose alternative forwarding paths if the error affects the memory addresses.

The idea is to increase the number of extra buses $B$ until the SAT solver has enough flexibility to fix all forwarding errors and find the error location. If the actual error is not identified using a certain value of $B$ by the designer, $B$ is incremented and the debugging process runs again. On subsequent debugging runs, SAT solutions from previous runs are blocked, since they have been shown not to be the error source. Experimental results show that small values for $B$ are typically sufficient to find most design bugs.

The constraints used in this memory model can be categorized into three different types of
Chapter 3. A Succinct Memory Model for Automated Design Debugging

3.3.1 Forwarding using Simulated Values

For each time-frame \( q \), let \( l(q) \) be the index of the most recent time-frame \( p < q \) such that there exists a forwarding path \( p \leadsto q \) during the simulation of the design using the error trace. Formally, \( l(q) = \max\{p | \exists \text{ path } (p \leadsto q) \text{ in the error trace}\} \). Let \( se_q \) denote the simulation enable variable that enables the forwarding of data to time-frame \( q \) according to simulated values. The following constraints, expressible in \( O(k) \) clauses, describe the functionality of the \( se_q \)'s:

\[
\bigwedge_{1 \leq q \leq k} [se_q \rightarrow (WA_l(q) = RA_q) \land (WD_l(q) = RD_q)] \tag{3.1}
\]

Note that \( l(q) \) does not exist for time-frame \( q \) if no previous time-frame writes to it during simulation.

If the data forwarding path \( l(q) \leadsto q \) is not affected by the error, the SAT solver can choose to assign \( se_q = 1 \) to allow this forwarding to occur. On the other hand, if a design bug has excited a change in the forwarding addresses, the simulated forwarding paths might be erroneous. Therefore setting \( se_q = 1 \) might not result in a satisfying assignment. In order for the SAT solver to be able to find a correct forwarding path, new bus variables are introduced, which enable alternative means of forwarding.

3.3.2 Forwarding using Bus Variables

We introduce \( B \) data and address buses, \( BD_b \) and \( BA_b \) \((1 \leq b \leq B)\), that allow the creation of new forwarding paths in the case of a mismatch in the simulated memory accesses. Each time-frame is allowed to write to or read from these bus variables. This added flexibility makes it possible for the SAT solver to produce satisfying assignments even if a design bug affects the correctness of the memory accesses. Let \( bwe_{p,b} \) denote the bus write enable signal which enables time-frame \( p \) to write to bus \( b \) (i.e. to \( BD_b \) and \( BA_b \)). The following constraints, expressible
in $O(kB)$ clauses, describe the functionality of the $bwe_{p,b}$’s:

$$\bigwedge_{p=1}^{k} \bigwedge_{b=1}^{B} [bwe_{p,b} \rightarrow (WA_p = BA_b) \land (WD_p = BD_b)]$$  

(3.2)

Setting $bwe_{p,b} = 1$ allows time-frame $p$ to write its address and data values to bus $b$. This is the first step in forwarding data from time-frame $p$ using the bus variables.

The next step involves a time-frame $q$ reading from the same bus. Let $bre_{q,b}$ be the bus read enable signal that allows time-frame $q$ to read from the data and address buses $BD_b$ and $BA_b$. The following constraints, expressible in $O(kB)$ clauses, describe the functionality of the $bre_{q,b}$’s:

$$\bigwedge_{q=1}^{k} \bigwedge_{b=1}^{B} [bre_{q,b} \rightarrow (RA_q = BA_b) \land (RD_q = BD_b)]$$  

(3.3)

Setting $bre_{q,b} = 1$ allows data to be forwarded from bus $b$ to time-frame $q$. Therefore, setting $bwe_{p,b} = 1$ and $bre_{q,b} = 1$ for some $p, q$ and $b$ creates a data forwarding path $p \leadsto q$.

If the simulated data forwarding path $l(q) \leadsto q$ is wrong because of a design bug, the SAT solver will attempt to use one of the unused buses to forward the data in a way that satisfies the design debugging problem constraints. This is done by setting $se_q = 0$, which bypasses the simulated forwarding and instead setting $bwe_{p,b} = 1$ and $bre_{q,b} = 1$ for some appropriate $b$ and $p$, in order to enable time-frame $q$ to read from time-frame $p$.

### 3.3.3 Enforcing Memory Semantics

Additional constraints are needed in order to ensure that enable signals for forwarding are set correctly, and that the memory semantics are preserved.

To ensure that each time-frame $q$ can only have one source to read from, the number of sources that are able to forward to it are constrained as follows:

$$\bigwedge_{q=1}^{k} \left[ \left( \sum_{b=1}^{B} bre_{q,b} + se_q \right) = re_q \right]$$  

(3.4)

Equation 3.4 captures the idea that if $re_q = 1$, only one of the forwarding sources, either a simulated forwarding path or a path using one of the buses, can be enabled. If $re_q = 0$, they are all disabled and no forwarding occurs to time-frame $q$. 


Chapter 3. A Succinct Memory Model for Automated Design Debugging

Constraining the sum of Boolean variables to a certain cardinality can be achieved in different ways as shown in [17]. Our implementation uses bit-wise hardware adders, which are built using a linear number of clauses in the number of added bits [37]. For Equation 3.4, \( k \) adders of \( B \) bits each are needed, resulting in a total of \( O(kB) \) clauses.

Next, the memory model must ensure that only one value is written to a single bus, using the following constraints:

\[
\bigwedge_{b=1}^{B} \left[ \sum_{p=1}^{k} bwe_{p,b} = 1 \right]
\]  

Equation 3.5 is also implemented using adders amounting to a total of \( O(kB) \) clauses. We must also make sure that a time-frame \( p \) does not forward its data if \( we_p = 0 \), with the following constraints, expressible in \( O(kB) \) clauses:

\[
\bigwedge_{p=1}^{k} \bigwedge_{b=1}^{B} \left[ we_p \rightarrow bwe_{p,b} \right] \\
\bigwedge_{q=1}^{k} \left[ we_{i(q)} \rightarrow se_q \right]
\]  

Equation 3.6

A final set of constraints is added to satisfy the causality of the memory semantics. That is, a write must occur to an address before a read from that address can take place. To ensure that this constraint is met, the model uses an intermediate bus read enable variable \( PE_{p,b} \) between each time-frame \( p \) and bus \( b \), as follows:

\[
\bigwedge_{p=1}^{k} \bigwedge_{b=1}^{B} \left[ bwe_{p,b} \rightarrow PE_{p,b} \right] \\
\bigwedge_{p=2}^{k} \bigwedge_{b=1}^{B} \left[ PE_{p,b} \rightarrow PE_{p-1,b} \right] \\
\bigwedge_{p=1}^{k} \bigwedge_{b=1}^{B} \left[ PE_{p,b} \rightarrow bre_{p,b} \right]
\]  

If time-frame \( p \) writes to bus \( b \), these constraints force the SAT solver to disable all reads from bus \( b \) by time-frames \( q \) such that \( q \leq p \). This ensures the causality of memory writes and reads using \( O(kB) \) clauses.

**Example 9** Figure 3.1 shows how the first two types of constraints can be added to an unrolled circuit. In this example, data is forwarded in the simulated error trace from time-frames 1 to 3.
and from time-frames 3 to 4. These connections are added to the SAT instance and are enabled via $se_3$ and $se_4$. We also have one set of bus variables ($B = 1$). Every time-frame $p$ can either read from or write to the bus depending on the enable signals $bre_{p,1}$ and $bwe_{p,1}$. Clearly, only one time-frame should write to the bus and only one source (either simulated forwarding or bus forwarding) should be enabled for each time-frame. This idea is captured with the additional constraints (not shown in the figure), which ensure that the memory semantics are preserved.

It should be noted that this construction allows for additional solutions (i.e. potential error locations) to be found, beyond what the explicit memory representation would find. The reason is that certain data forwarding types are permitted which are not allowed by the memory semantics. A special case can occur when a time-frame $q$ reads from time-frame $p$ but the most recent write is at time-frame $m$ such that $p < m < q$. The method will still find all solutions with up to $B$ changes to the memory accesses, but might return extra solutions that are not potential error locations.

The number of clauses produced by this construction grows according to $O(kB)$, where $B$ is the number of buses used and $k$ is the number of unrollings. In many cases, $B << k$, resulting in a significant decrease in the size of the problem compared with previous memory models.
### 3.4 Modelling Initial Memory Configurations

The above construction disallows reads from the initial state of memory (i.e. before the first time-frame), which is a valid assumption if the trace begins from a reset state. However, this may not always be the case since debugging a circuit from a reset state may require the use of an excessively long trace. Instead, a suffix of the trace is commonly used to reduce the problem size.

Let $M_{\text{initial}}$ be an initial state of memory comprising of $I$ pairs of addresses and data, of the form $(IA_i, ID_i)$, where $1 \leq i \leq I$. We wish to allow time-frames to read values from $M_{\text{initial}}$ via the forwarding methods introduced in Section 3.3. To that end, the two forwarding methods from our debugging memory model, namely forwarding according to the simulated values and forwarding using bus variables are used.

An address and data entry $(IA_i, ID_i) \in M_{\text{initial}}$ is allowed to be forwarded to some time-frame $q$ if that behavior occurs during simulation. Let $m(q)$ be the initial state location that time-frame $q$ reads from during simulation. Note that $m(q)$ is unique because time-frame $q$ can only read from one address. Now the following constraints to Equation 3.1 to include data and addresses read from the initial state can be added:

$$\bigwedge_{1 \leq q \leq k} \left[ \exists m(q) \left[ se_q \rightarrow (IA_{m(q)} = RA_q) \land (ID_{m(q)} = RD_q) \right] \right] \quad (3.8)$$

Similarly, each entry in $M_{\text{initial}}$ can now be a source for any time-frame to read from using the bus variables. We must allow the initial state to write to the bus variables in a similar manner to Equation 3.2 for each $(IA_i, ID_i)$ pair in $M_{\text{initial}}$. We extend the notation for $bwe_{p,b}$’s to let $bwe_{0,b}^i$ denote the bus write enable signal that allows the initial memory location $(IA_i, ID_i)$ to write to bus $b$. Using these variables, the following constraints need to be added to Equation 3.2:

$$\bigwedge_{i=1}^{I} \bigwedge_{b=1}^{B} \left[ bwe_{0,b}^i \rightarrow (IA_i = BA_b) \land (ID_i = BD_b) \right] \quad (3.9)$$

In addition, it is necessary to ensure that the bus variables are given the choice to connect
Figure 3.2: Modelling an initial memory configuration

to the initial state of memory by modifying Equation 3.5 as follows:

$$\bigwedge_{b=1}^{B} \left[ \left( \sum_{p=1}^{k} bwe_{p,b} + \sum_{i=1}^{I} bwe_{i,b} \right) = 1 \right]$$

(3.10)

This ensures that each bus is written to either by the initial state of memory or by a time-frame in the trace.

Every address in memory does not need to be included in $M_{\text{initial}}$, but rather only values that are known to be valid. This information is easily obtained by a standard simulator. It should be noted that using similar types of constraints as described in this section, the crossbar model in [20] can be extended to deal with initial memory configurations as well.

**Example 10** Figure 3.2 extends Example 9 to handle initial memory states. The trace length has been cut in half by taking the last two time-frames from Example 9 (the time-frames are re-indexed). The memory state after the second time-frame in Example 9 now becomes $M_{\text{initial}}$, which has one entry, $(IA_1, ID_1)$. $(IA_1, ID_1)$ is connected to time-frame 1 and enabled via $se_1$ because during simulation this time-frame would have read from $M_{\text{initial}}$. In addition, the model must also connect $(IA_1, ID_1)$ to the bus variables to allow it to forward data to any future time-frame. This is enabled using $bwe_{1,0,1}$, which is constrained in Equation 3.10 (not shown in the figure).
3.5 Modelling Final Memory Configurations

In certain situations, a cycle accurate expected output trace is not available. Instead, only an expected final state of memory can be used for comparison. For example, in verification environments today, it is common for RTL-level implementations to be compared to non-cycle, pin accurate high-level behavioral models, e.g. a C program. If an error is found during verification, the trace ends when the state of memory differs between the design and the high-level model and no intermediate pin mapping values are available. This results in the need to constrain the final state of memory to the expected memory configuration given by the high-level model.

The presented memory model for design debugging allows a natural extension to handle these types of non-cycle accurate output traces. In this problem, an erroneous design, an input trace, as well as the expected final state of memory $M_{final}$ are given. $M_{final}$ is a set of $F$ pairs $(FA_f, FD_f)$, where $FD_f$ is the data at address $FA_f$, for $1 \leq f \leq F$. The idea behind this formulation is to ensure that every pair $(FA_f, FD_f) \in M_{final}$ has at least one source which forwards data to it, coming from either a simulation forwarding or a bus forwarding.

Let $h(f)$ denote the most recent time-frame $p$ such that there exists a forwarding path
between time-frame \( p \) and \((FA_f, FD_f)\) in the simulation of the design in the error trace. We extend the notation for \( se_q \)'s to let \( se_{k+1}^f \) denote the enable signal allowing a forwarding path from time-frame \( h(f) \) to the final memory location \((FA_f, FD_f)\). Now, the following constraints to Equation 3.11 to include data and addresses written to the final memory state can be added:

\[
\bigwedge_{1 \leq f \leq F} \left[ se_{k+1}^f \rightarrow (WA_{h(f)} = FA_f) \land (WD_{h(f)} = FD_f) \right]
\] (3.11)

Setting \( se_{k+1}^f = 1 \) creates a forwarding path from time-frame \( h(f) \) to \((FA_f, FD_f)\). Note that \( h(f) \) may not exist for every \( f \) because the expected final memory state might have more entries than the simulated final memory state. Also, the definition of \( h(f) \) along with Equation 3.11 can be trivially extended to allow paths from initial memory state entries to final memory state entries. Such a forwarding path indicates that there have been no memory writes to that address during the trace.

The other possible data forwarding sources to the final state are the buses. For each pair \((FA_f, FD_f)\), the notation is extended for \( bre_{q,b} \) to let the enable signal \( bre_{k+1,b}^f \) allow a connection to each set of bus variables, \( BA_b \) and \( BD_b \). This enable signal is constrained in the same way as in Equation 3.3:

\[
\bigwedge_{f=1}^{F} \bigwedge_{b=1}^{B} [bre_{k+1,b}^f \rightarrow (FA_f = BA_b) \land (FD_f = BD_b)]
\] (3.12)

When \( bre_{k+1,b}^f = 1 \), data is forwarded from the bus to \((FA_f, FD_f)\).

Finally, it is necessary to ensure that only one time-frame (or the initial state of memory) forwards data to each pair \((FA_f, FD_f)\). A similar constraint to Equation 3.4 is used to ensure this, as follows:

\[
\bigwedge_{f=1}^{F} \left[ \sum_{b=1}^{B} bre_{k+1,b}^f + se_{k+1}^f = 1 \right]
\] (3.13)

**Example 11** Consider Figure 3.3, an extension of Example 10, using the final state of memory \( M_{final} \) to constrain the problem. \( M_{final} \) has two entries \((FA_1, FD_1)\) and \((FA_2, FD_2)\). During simulation, \( FA_1 \) is written to by time-frame 1 and therefore a connection is made between them, which is enabled using \( se_1^1 \). A connection is also made to the bus variables in a similar manner with \( bre_1^1 \). Notice that the entry \((FA_2, FD_2)\) has no corresponding simulated forwarding path.
This implies that during simulation no time-frame writes a value to this memory location. The only way to satisfy this entry is for a time-frame to write to the bus and for \( b_{3,1}^2 \) to be enabled.

### 3.6 Summary

In this chapter a novel memory model for design debugging is presented. The proposed memory model results in a parameterizable SAT encoding that grows linearly with the error trace length. In addition, the described method is extended to handle initial memory configurations as well as non-cycle accurate design debugging with an expected final memory state.
Chapter 4

Scalable VLSI Debugging with Interpolation

4.1 Introduction

Modern automated debugging tools use time-frame expansion which replicates the combinational component of a design for the number of clock cycles in an error trace. However, with the growing size and complexity of modern designs, error traces generated from functional verification tools can be thousands of clock cycles long [9]. This places a large memory requirement for the debugging tool and limits their ability to analyze large designs with large error traces.

Many other fields of CAD also have large computational requirements either due to memory or runtime requirements. For example, ideally a detailed route could be explored for each placement, however this would exhaust the computational resources that are currently available for modern designs. To cope with these resource issues, parts of the problem are approximated. This is a common theme in CAD where intractable problems are simplified so they can be solved within these resources constraints. A similar theme is applied in this chapter.

This chapter proposes a scalable SAT-based debugging algorithm that divides the debugging problem into multiple parts. It accomplishes this by using interpolants to reduce the number of simultaneous time-frames that need to be stored in memory. The algorithm analyzes windows
of time-frames along the length of the error trace beginning with a suffix window (as described in Section 2.2) and iteratively moves the window backwards until a prefix window of the error trace is analyzed. The interpolants are used to over-approximate constraints for a suffix of the error trace that is not modeled in the current window, ensuring that the erroneous behavior is properly constrained. Additionally, this method is shown to be complete and a technique using multiple interpolants is presented to improve its resolution.

The sections in this chapter are organized as follows. Section 4.2 describes some related work in debugging and verification. In Sections 4.3 and 4.4, a description of how to generate debugging instances for a suffix window and prefix window of an error trace is shown. Using these two ideas, a complete scalable algorithm for debugging is described in Section 4.5 which partitions the original problem into smaller debugging instances. Finally, Section 4.6 shows how to improve resolution by using multiple interpolants.

4.2 Related Work

Bounded Model Debugging (BMD) was introduced in [35] and presents an iterative method for debugging an error trace. It starts by debugging a suffix window of the error trace. It then iteratively increases the length of the suffix until the entire error trace is analyzed. However, in some cases the full error trace need not be analyzed by using a suspect on the initial state constraints. This suspect variable on the initial state ensures that if there exists a fix before the current analyzed suffix then the initial state suspect variable will be found. Results are promising as in many cases the full error trace does not need to be analyzed. However, the method will potentially expand the suffix to the entire error trace resulting in no net savings in memory and an increase in run-time.

UNSAT cores have been previously used in SAT-based debugging to deal with multiple design errors. In [39], the authors intersect multiple UNSAT cores to reduce the number of potential suspects that need to be analyzed. Since any suspect needs to be able to resolve an UNSAT core, it must be able to minimally resolve the intersection of multiple UNSAT cores. However, one drawback of this method is that it is notoriously difficult to obtain multiple
Interpolants have been widely used in other CAD applications. It proved very effective in
the model checking domain by allowing a SAT-based unbounded model checking algorithm [30].
This was a big step forward because previously the state of the art model checkers were based
on symbolic techniques which are known to have problems scaling for certain types of circuits.
Since then, it’s popularization has extended to the synthesis domain [31] where it is used to
calculate a dependency function.

4.3 Suffix Window Debugging

Debugging a suffix of an error trace can be achieved by applying the original SAT-based debug-
ging scheme given in Equation 2.6. By using a suffix, only errors that are both excited within
this window and propagate to primary outputs can be found. The following lemma describes a
useful characteristic of suspects found in a suffix debugging instance.

Lemma 1 Any suspect found in a debugging instance, $\text{Debug}_p^k$, for a suffix of an error trace,
$\mathcal{V}_p^k$, will be found as a suspect to the debugging instance, $\text{Debug}_0^k$, for the entire error trace,$\mathcal{V}_0^k$.

Proof: Let $M(E)$ be an assignment to the suspect variables in $E$ such that $\text{Debug}_p^k \wedge M(E)$
is satisfiable. We wish to prove the lemma which can be written as:

$$\text{Debug}_p^k \wedge M(E) \text{ is SAT } \rightarrow \text{Debug}_0^k \wedge M(E) \text{ is SAT}$$

From Equation 2.6, it can be seen that that $\text{Debug}_0^{p-1} \wedge \text{Debug}_p^k$ and $\text{Debug}_0^k \wedge S^p(s^p)$ generate
the same clauses. $\text{Debug}_0^{p-1}$ is SAT regardless of the error trace because the error has not been
observed yet, so there is no mismatch in primary outputs. $\text{Debug}_0^{p-1} \wedge S^p(s^p)$ is SAT when no
suspect variables are active because the instance $\text{Debug}_0^{p-1}$ amounts to simulating the circuit
for the first $p$ cycles of the error trace generating the same values as $S^p(s^p)$. Finally, $\text{Debug}_0^{p-1} \wedge
S^p(s^p) \wedge M(E)$ is SAT because each active suspect variable allows the corresponding component
to become an arbitrary non-deterministic function, which will not change the satisfiability of
an instance if it was already satisfiable.
Therefore, if $\text{Debug}_p^k \land M(E)$ is SAT then $\text{Debug}_0^{p-1} \land \text{Debug}_p^k \land M(E)$ is SAT, since the only common variables are $s^p$ and $E$ which are fully assigned. As a result, $\text{Debug}_0^k \land S^p(s^p) \land M(E)$ is SAT implying that $\text{Debug}_0^k \land M(E)$ is SAT as required.

Lemma 1 guarantees that suspects found in the suffix are suspects that will be found in the entire error trace. However, if the error is excited before the current suffix, then there is no guarantee that the error will be found in $\text{Debug}_p^k$. Even though analyzing a suffix of an error trace may not result in a complete algorithm, valuable information can be extracted from the resulting UNSAT core as stated in the following theorem.

**Theorem 1** Let $U$ be an UNSAT core generated after blocking all satisfying assignments to suspects for $\text{Debug}_p^k$. If $U \cap S^p(s^p) = \emptyset$ then the suspects found in $\text{Debug}_p^k$ will be exactly the suspects found in the entire debugging instance, $\text{Debug}_0^k$.

**Proof:** From Lemma 1 any suspect found in $\text{Debug}_p^k$ is a suspect found in the entire debugging instance, $\text{Debug}_0^k$.

Now we prove by contradiction that any suspect found in $\text{Debug}_0^k$ will be found in $\text{Debug}_p^k$. Assume towards a contradiction that, $M(E)$ is an assignment to the suspect variables such that $\text{Debug}_0^k \land M(E)$ is SAT and $\text{Debug}_p^k \land M(E)$ is UNSAT. And let $U$ be the UNSAT core derived after blocking all satisfying assignments to suspects for $\text{Debug}_p^k$, which contains no clauses in $S^p(s^p)$.

Since $\text{Debug}_p^k \land M(E)$ is UNSAT, $M(E)$ is not blocked by any of the blocking clauses to $\text{Debug}_p^k$, which we denote by $\text{blocking\_clauses}_p^k$. This means that $\text{Debug}_0^k \land \text{blocking\_clauses}_p^k$ is satisfiable. However we know that in terms of clauses

$$U \subseteq (\text{Debug}_p^k \land \text{blocking\_clauses}_p^k - S^p(s^p)) \subseteq \text{Debug}_0^k \land \text{blocking\_clauses}_p^k$$

since $U$ does not contain any clauses from $S^p(s^p)$. However, $\text{Debug}_0^k \land \text{blocking\_clauses}_p^k \land M(E)$ is satisfiable, so $U \land M(E)$ is satisfiable. But $U$ is an UNSAT core, which is a contradiction. So it must be the case that $\text{Debug}_p^k \land M(E)$ is SAT.

Theorem 1 gives a condition to omit a prefix debugging analysis with very little additional computation beyond extracting the UNSAT core from the suffix debugging instance. However,
in the case where Theorem 1 is not valid, the prefix of the error trace must be analyzed to get a complete set of suspects. The following example illustrates a case where Theorem 1 cannot be applied because there are clauses in the UNSAT core from the initial state predicate.

**Example 12** A suffix debugging instance derived from Example 8 is shown in Figure 4.1. The suffix, \( V_1 \), is used to produce a suffix debugging instance Debug\(^1\) with \( N = 1 \). The clauses for the suffix debugging instance are shown to the right of the circuit diagram. This instance is unsatisfiable. The following is an UNSAT core from the instance:

\[
(s_0^1)(x_1^1)(y_1^1)(y_2^1) \\
(s_0^1 + y_1^1 + e_1)(x_1^1 + y_1^1 + e_1) \\
(s_0^1 + x_1^1 + y_1^1 + e_1) \\
(x_2^1 + y_2^1 + e_2)(x_2^1 + y_2^1 + e_2) \\
(e_1 + e_2)(\overline{e_1} + \overline{e_2})
\]

Using Theorem 1, it can be seen that Debug\(^1\) does not result in the complete set of suspects to Debug\(^0\) because the UNSAT core contains the clause \( s_0^1 \subseteq S^1(s^1) \), so the prefix of the error trace still needs to be analyzed.

**Example 13** A suffix debugging instance derived from the circuit in Example 8 is shown in Figure 4.2 with a different suffix error trace given below:

\[
V_1^1 = \langle s_0^1, \langle x_1^1 \land x_2^1 \rangle, \langle y_1^1 \land y_2^1 \rangle \rangle
\]
The suffix, $V_1$, is used to produce a suffix debugging instance $\text{Debug}_1$ with $N = 1$. The clauses for the suffix debugging instance are shown to the right of the circuit diagram. This instance is unsatisfiable. The following is an UNSAT core from the instance after the solution $e_2 = 1$ was found:

$$(s_0^1)(x_1^1)(y_1^1)(y_2^1)$$
$$(s_0^1 + y_1^1 + e_1)(x_1^1 + y_1^1 + e_1)$$
$$(s_0^1 + x_1^1 + y_1^1 + e_1)$$
$$(x_2^1 + y_2^1 + e_2)(x_2^1 + y_2^1 + e_2)$$
$$(e_1 + e_2)(\overline{e_1} + \overline{e_2})$$

Using Theorem 1, it can be seen that $\text{Debug}_1$ results in the complete set of suspects to $\text{Debug}_0$ because the UNSAT core does not contain any initial state clauses.

### 4.4 Prefix Window Debugging

Debugging a prefix of an error trace can be formulated in two parts. The first part uses the conventional SAT-based formulation (Equation 2.6) using a prefix of the error trace. The second part is an interpolant approximating time-frames for the corresponding suffix of the error trace.

Recall that the erroneous behavior is only observed in the last time-frame. If only a prefix of the error trace is modelled then the instance will not be properly constrained with the erroneous behavior because the prefix debugging instance does not model the time-frame with
the erroneous behavior. To avoid this situation, the interpolant is used as an over-approximation for the constraints that model the corresponding suffix. This ensures that the prefix debugging instance is properly constrained.

The interpolant can be generated by using an UNSAT core of the solved suffix debugging instance. To generate the interpolant, a partition of $\text{Debug}_p^k \land \text{blocking clauses}$ is defined by partitioning the clauses into two sets $A$ and $B$. Set $A$ represents the clauses modelling the enhanced transition function from $p$ to $k$ along with the primary input and output predicates from the error trace. Set $B$ represents the initial state predicate, the error cardinality constraints and the blocking clauses. The clauses forming $\text{Debug}_p^k \land \text{blocking clauses}$ can be separated into $A$ and $B$ as follows:

\[
A = \bigwedge_{i=p}^k X^i(x_i) \land Y^i(y_i) \land T(s^i, s^{i+1}, x^i, y^i, E)
\]

\[
B = S^p(s^p) \land \Phi_N(E) \land \text{blocking clauses} \quad (4.1)
\]

The common variables of $A$ and $B$ are the state variables $s^p$ and the suspect variables $E$. Using this partition, an interpolant for the suffix, denoted $P_p^k$, can be generated from the resolution graph using the algorithm from [30].

$P_p^k$ can be interpreted as an over-approximation of the suffix debugging instance. $P_p^k$ will involve a subset of state and suspect variables that are directly related to the erroneous behavior observed at the primary outputs. The benefit of $P_p^k$ is that it retains only the useful information that causes the erroneous behavior instead of modelling all the time-frames for the suffix of the error trace. In cases where the interpolant gets too large, the original clauses can be used in place of the interpolant, bounding the size of the constraints used to model the erroneous behavior. However experimental results show that in most cases, the interpolant is much smaller than the instance it was generated from, confirming the efficacy of using interpolants for debugging.

**Example 14** Figure 4.3 shows the resulting resolution graph on the left and interpolant on the right from the UNSAT core in Example 12. Notice how many of the root nodes of the resolution graph generate constants values in the interpolant. This is a common occurrence and generally leads to a small interpolant relative to the UNSAT core that it was derived from.
\((x^1_2) \quad (\overline{x^1_2} + y^1_2 + e_2)\)

\((\overline{y^1_2} + e_2) \quad (y^1_2)\)

\((\overline{e_1} + \overline{e_2}) \quad (e_2)\)

\((\overline{e_1}) \quad (s^1_0 + \overline{y^1_1} + e_1)\)

\((y^1_1) \quad (s^1_0 + \overline{y^1_1})\)

\((s^1_0) \quad (\overline{s^1_0})\)

\((0)\)

Figure 4.3: Resolution graph and interpolant
Figure 4.4: Prefix window debugging with an interpolant

Using $P^k_p$, $\text{Debug}^p_{0}^{-1}$ can be constrained with the cause for the erroneous behavior. The interpolant ensures that any suspect found in the prefix debugging instance resolves the erroneous behavior from the UNSAT core found in the suffix. The debugging instance for a prefix of an error trace with an interpolant, which is denoted as $\text{DebugItp}^p_{0}^{-1}$, can be written as follows:

$$\text{DebugItp}^p_{0}^{-1} = \text{Debug}^p_{0}^{-1} \land P^k_p$$

(4.2)

$\text{DebugItp}^p_{0}^{-1}$ will be UNSAT when no suspect variables are active because $\text{Debug}^p_{0}^{-1}$ will be equivalent to simulating the design for clock cycles 0 to $p - 1$ and will implicitly generate the initial state predicate $S^p$ which is known to be UNSAT with $P^k_p$. The next example builds from previous ones to show how a prefix debugging instance can be created.

**Example 15** Figure 4.4 shows how the interpolant generated in Example 14 can be used to debug a prefix of an error trace. Notice that the interpolant is significantly smaller once the constants have been propagated through the gates. In Figure 4.4, activating suspect variable, $e_2$, leads to the only satisfying assignment. This is consistent with the solution found in Example 8.

The interpolant constrains the prefix debugging instance but it is an over-approximation. In other words, it will not miss suspects, as stated in the next theorem.
Theorem 2 Any suspect found in Debug$^k_0$ will be found in DebugItp$^{p-1}_0$.

Proof: By definition, Debug$^k_0 = Debug^{p-1}_0 \land A$, where $A$ is defined in Equation 4.1. So any satisfying assignment to Debug$^k_0$ will satisfy Debug$^{p-1}_0$ and $A$. But $A \rightarrow P^k_p$, so it also satisfies $P^k_p$ satisfying DebugItp$^{p-1}_0$.

Theory 2 guarantees that solving the prefix debugging instance will result in a complete method where no suspects will be missed. However, it does not guarantee that spurious suspects will not be found. $P^k_p$ is used as an over-approximation for the suffix, so it does not provide as many constraints as explicitly modelling time-frames $p$ to $k$. This results in DebugItp$^{p-1}_0$ possibly returning suspects that will not be found when debugging the entire error trace. Section 4.6 aims to reduce these extra suspects and improve the resolution.

4.5 Scalable Debugging Algorithm

By using suffix and prefix debugging instances, it is possible to further divide the debugging problem into smaller windows that model no more than a user-defined number of time-frames. Algorithm 4.1 presents pseudo-code for a scalable debugging algorithm that divides an error trace of length $k$ into $\lceil k/step \rceil$ windows, where $step$ is a user-defined parameter that specifies the maximum number of simultaneous time-frames to be modelled.

The algorithm iteratively analyzes windows of the error trace, starting with a suffix (lines 8-19). In each iteration, it begins by analyzing the current window of the error trace and finds all suspects shown on line 11. It then proceeds to generate an UNSAT core from the same instance and checks whether it contains any variables corresponding to the initial state predicate for the current instance. If it does not have any, it returns the current set of suspects. This is shown on lines 12-15. This condition allows for an early exit from the algorithm which in Theorem 1 was shown to be complete. If an early exit is not taken, it proceeds to generate an interpolant from the UNSAT core. Finally, it removes any suspects found in this iteration for consideration in the next iteration of the loop, pruning the search space for future iterations.

By iteratively analyzing consecutive windows of an error trace, the peak memory usage will be dramatically lowered with potential improvements in run-time. Even though the algorithm
divides the error trace beyond just a suffix and prefix, it still guarantees completeness as shown in Theorem 3 but first we have to prove Lemma 2.

**Lemma 2** Let $k$ be the length of the error trace. Let $A_p^q = \bigwedge_{i=p}^q X^i(x_i) \land Y^i(y_i) \land T(s^i, s^{i+1}, x^i, y^i, E)$, defined in a similar manner to Equation 4.1. Let $P_i$ denote the variable $P$ from Algorithm 4.1 after the $i^{th}$ iteration of the loop in Algorithm 4.1.

After the $i^{th}$ iteration of the loop in Algorithm 4.1, $A_{k\text{-}i\text{step}}^{k-1} \rightarrow P_i$.

**Proof:** We will prove this by induction on the loop iteration $i$.

Base case ($i = 1$): In the first iteration of the loop $A_{k\text{-}step}^{k-1} \rightarrow P_1$ by definition according to Equation 4.1. This proves the lemma for $i = 1$.

Inductive Step: Assume that $A_{k\text{-}i\text{step}}^{k-1} \rightarrow P_i$ is true for the $i^{th}$ iteration. We will show the lemma true for the $(i+1)^{th}$ iteration.

By definition we have:

$$A_{k-(i+1)\text{step}}^{k-1} = A_{k-(i+1)\text{step}}^{k-1} \land A_{k-i\text{step}}^{k-1}$$

Using the inductive hypothesis we get:

$$A_{k-(i+1)\text{step}}^{k-1} \rightarrow A_{k-(i+1)\text{step}}^{k-1} \land P_i$$

From the algorithm and Equation 4.1 we get:

$$A_{k-(i+1)\text{step}}^{k-1} \rightarrow P_{i+1}$$

Thus showing the property true for a general case $i$.

**Theorem 3** Algorithm 4.1 is complete.

**Proof:** Consider the last iteration, $i^*$, of the loop.

By definition:

$$\text{Debug}_p^{k-1} = \text{Debug}_p^{k-i^*\text{step}-1} \land A_{k-i^*\text{step}}^{k-1}$$
Using Lemma 2:

\[ \text{Debug}^{k-1}_p \rightarrow \text{Debug}^{k-i*\text{step}-1}_p \land P_{i^*} \]

This implies that a solution found for \( \text{Debug}^{k-1}_p \) will also be a solution found for \( \text{Debug}^{k-i*\text{step}-1}_p \land P_{i^*} \). If an early exit is taken, then Theorem 1 can be used and all solutions found so far will result in a complete set of solutions. Otherwise when \( p = 0 \), \( \text{Debug}^{k-i*\text{step}-1}_0 \land P_{i^*} \) will result in a complete set of solutions.

Although Algorithm 4.1 is complete, there is a trade-off between the \text{step} parameter and the final resolution. Each successive interpolant generated will potentially be a weaker constraint than the previous one. By setting \text{step} to a small value, too many suspects can be returned. One way to cope with this is to provide a ranking of the suspects to the user so they can concentrate their effort on the most likely suspect. Algorithm 4.1 implicitly gives a useful ranking of suspects. More confidence can be given to suspects found in earlier iterations because a stronger constraint is used for the approximation of the suffix. In the case of the first iteration, all suspects found in the suffix will be found when debugging the entire error trace, as stated in Lemma 1.

### 4.6 Improving Resolution using Multiple Interpolants

As mentioned in the previous section, Algorithm 4.1 is complete but can return spurious suspects that may not be the cause of the error. The spurious suspects are due to the over-approximation nature of using interpolants. However, by using multiple UNSAT cores to generate multiple interpolants, a stronger constraint can be made thus potentially improving the resolution and reducing the amount of spurious solutions. The intuition is that with multiple interpolants, the approximation will more closely match the original constraints potentially reducing the number of spurious suspects that are found.

Using this fact, Algorithm 4.1 can be improved (line 12) by extracting multiple UNSAT cores to generate multiple interpolants. However, extracting multiple UNSAT cores can be an expensive process in general [25]. Algorithm 4.2 presents pseudo-code for a fast procedure for finding multiple UNSAT cores specifically for use in Algorithm 4.1.
The algorithm begins with an UNSAT instance and finds an UNSAT core (line 4). If the UNSAT core doesn’t contain any clauses involving the initial state predicate, it exits and returns all UNSAT cores found so far (line 6-8). Otherwise, it randomly removes a subset of clauses from the initial state predicate that were involved in the current UNSAT core and is sent to the SAT solver again (line 10). This process is repeated until the instance is found to be satisfiable.

The size of the subset of initial state predicate clauses removed is a parameter to the algorithm. A smaller subset will leave more constraints in the problem having a higher chance of generating another UNSAT core but potentially taking more time and memory. By limiting the size of the subset and the number of cores found, the user can effectively trade-off run-time and memory for improved resolution.

4.7 Summary

In this chapter, a scalable design debugging algorithm using interpolants is proposed. It partitions the problem into a sequence of smaller sub-problems that are easier to solve. Interpolants are used to reduce the number of simultaneous time-frames examined in the error trace by replacing sets of original clauses with a succinct approximation. The method is proven to be complete and an additional technique is presented to improve the quality of the debugging results using multiple interpolants.
Algorithm 4.1 Debugging with interpolants
1:  \( \text{step} := \text{maximum number of time-frames} \)

2:  \textbf{procedure} \( \text{DEBUGINTERPOLANT}(\text{step}) \)

3:  \( N := \text{error cardinality} \)

4:  \( E := \text{set of potential suspect variables} \)

5:  \( k := \text{length of error trace} \)

6:  \( \text{sol}utions := \text{suspects found by algorithm} \)

7:  \( \text{sol}utions \leftarrow \emptyset, P \leftarrow 1 \)

8:  \textbf{while} \( k \geq 0 \) \textbf{do}

9:  \( p \leftarrow \max(k - \text{step}, 0) \)

10:  \( \text{inst} \leftarrow \text{Debug}^{k-1}_{p}(N, E) \land P \)

11:  \( \text{sol}utions \leftarrow \text{sol}utions \cup \text{SOLVEALL}(\text{inst}) \)

12:  \( U \leftarrow \text{EXTRACTUNSATCore}(\text{inst}) \)

13:  \textbf{if} \( U \cap S^{p}(s^{p}) = \emptyset \) \textbf{then}

14:    \( \text{return sol}utions \)

15:  \textbf{end if}

16:  \( P \leftarrow \text{GENERATEINTERPOLANT}(U) \)

17:  \( \text{E} \leftarrow \text{E} - \text{sol}utions \)

18:  \( k \leftarrow k - \text{step} \)

19:  \textbf{end while}

20:  \( \text{return sol}utions \)

21:  \textbf{end procedure}
Algorithm 4.2 Extracting multiple UNSAT cores

1: procedure $\text{ExtractMultipleCores}(\text{instance})$

2:     $\text{Cores} \leftarrow \emptyset$

3:     while $\text{instance}$ is UNSAT do

4:         $U \leftarrow \text{ExtractCore}(\text{instance})$

5:         $\text{Cores} \leftarrow \text{Cores} \cup \{U\}$

6:     if $U \cap S^p(s^p) = \emptyset$ then

7:         return $\text{Cores}$

8:     end if

9:         $\text{to\_remove} \leftarrow \text{select\_clauses}(S^p(s^p) \cap U)$

10:        $\text{instance} \leftarrow \text{instance} - \text{to\_remove}$

11:     end while

12:     return $\text{Cores}$

13: end procedure
Chapter 5

Experimental Results

5.1 Introduction

This chapter presents the experimental results for the contributions presented in this thesis. Section 5.2 presents results for the embedded memory model for automated design debugging. Section 5.3 presents results for the novel scalable SAT-based design debugging algorithm with interpolants.

5.2 Memory Model Results

This section presents experimental results for the proposed memory modelling technique in a SAT-based automated design debugging framework. In addition to the memory model presented in Chapter 3, the memory model in [20] is implemented and extended to handle arbitrary initial memory configurations in a design debugging context. The memory model presented in [20] involves connecting each time-frame $p$ with all time-frames $q > p$ using a forwarding path. Since this resembles a crossbar, for simplicity, we will refer to this memory model as the crossbar model. The results for the explicit memory representation which models memory as an array of flip-flops are also included. A SAT-based design debugging framework similar to [3] is used to produce the SAT instances. The SAT-solver MINISAT-v2.0 [16] is deployed on a Pentium Core 2, 2.4 GHz workstation with 4GB of memory.

The effectiveness of the proposed memory model is shown for multiple debugging instances
on three designs with large dual-port embedded memories from OpenCores.org [33]. Each instance is generated using a different bug location inserted randomly in the RTL design. The type of bugs inserted were common errors such as a wrong assignment, missing case statement or incorrect operator. The error cardinality is set to 1 in all the debugging instances. The number of bus variables $B$ is initialized to 1 and is incremented until the error is located.

Table 5.1 shows the relevant statistics for each of the three designs. The first three columns show the circuit name followed by the gate and flip-flop (DFF) count in the absence of embedded memories. Columns four and five show the address and data widths of the embedded memory, while the next two columns show the additional gates and flip-flops generated due to the explicit memory representation.

Table 5.2 shows cycle accurate design debugging results for two circuits: fifo and vga. Four different instances of varying trace lengths starting from a reset state are created for fifo (fifo1 to fifo4). For each of the three instances with larger traces, results are shown using a suffix of the trace and making use of the initial memory configuration model described in Section 3.4 (fifo2-init to fifo4-init). Six instances of vga (vga1-init to vga6-init) are also generated using the initial memory configuration extension.

The first two columns in Table 5.2 show the instance name and trace length. Column base shows the memory usage of each instance excluding the memory model size. The next two columns show the run-time and size of the memory model for the explicit memory representation. Columns 6 to 11 show the number of clauses in thousands, followed by the run-time in seconds and size of the memory model in MBs for both the crossbar model and the proposed
memory model for design debugging (which is referred to as the bus model). The last column shows the $B$ value required to solve each instance using our method. The run-times for the bus model are the sum of the run-times of all runs starting from $B = 1$ until the listed value of $B$.

For fifo, the proposed method produces an average run-time reduction of 27% and memory model size reduction of 95% compared to the crossbar model. For vga, the proposed memory model gives an average run-time reduction of 15% and size reduction of 98% compared to the crossbar model. Both of these methods significantly outperform the explicit memory model both in terms of performance and memory model size. In fact, the explicit memory representation runs out of memory for many of the instances.

Looking across the second and fourth rows of Table 5.2, one can notice that the explicit memory representation runs out of memory when the full trace length is used but is able to solve the instance using a suffix of the trace. Comparing fifo2 and fifo2-init, reducing the trace length results in a reduction in memory usage from 478MB to 150MB for the crossbar model, and from 7MB to 3MB for the bus model.

Figure 5.1 plots the performance results on a log-log scale for fifo and vga instances for the crossbar and bus model. Notice that most of the points lie below the 45 degree line indicating that the run-times of the proposed method are generally better. On the points that lie above the line, the performance is marginally better showing that the proposed memory model can achieve similar performance results with a large reduction in the memory model size.

Figure 5.2 shows the size of each memory model (in black) relative to the size of the entire instance for fifo3, fifo2-init and vga6-init. Here, exp, cro and bus respectively stand for explicit memory model, crossbar model and bus model. It can be seen that the explicit model takes a large portion of the problem size for all three instances, while the crossbar model size is more dependent on the trace length. The bus model consistently shows a small relative memory footprint.

From Table 5.2, it can be seen that in most instances $B = 1$. This can be explained in two ways. The first is that the error has very little effect on the address lines so only one time-frame is affected by the change, thus only one set of bus variables is needed. The second is that an
Chapter 5. Experimental Results

Figure 5.1: Cycle accurate performance results

Figure 5.2: Normalized memory size for fifo3, fifo2-init and vga6-init
error that affects an address line propagates to the outputs relatively quickly. As a result, it does not manifest itself across multiple time frames because the trace would end at that point. In practice, this to be a common occurrence, thus confirming the effectiveness of our memory model.

Figure 5.3 shows how the memory usage of the crossbar model and the debugging memory model vary with the number of unrollings $k$ for the fifo design. Note that the data used in this figure is not shown in Table 5.2. As predicted, the crossbar model grows quadratically with $k$, while the bus model is linear for a fixed value of $B$. It can also be seen that the memory usage grows linearly when increasing $B$.

Table 5.3 shows non-cycle accurate design debugging results on mrisc. It compares the proposed method to the explicit memory representation. mrisc is a microprocessor design whose outputs are not explicitly generated at each clock cycle. Instead, to verify the correctness of the design, the final state of the embedded memory (a register file) is compared to a high-
Table 5.2: Cycle accurate design debugging results for **fifo** and **vga**

<table>
<thead>
<tr>
<th>Instance Info</th>
<th>Explicit</th>
<th>Crossbar</th>
<th>Bus Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># clks</td>
<td>base (MB)</td>
<td>time (sec)</td>
</tr>
<tr>
<td>instance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo1</td>
<td>28</td>
<td>21</td>
<td>11.3</td>
</tr>
<tr>
<td>fifo2</td>
<td>307</td>
<td>167</td>
<td>MEMOUT</td>
</tr>
<tr>
<td>fifo3</td>
<td>227</td>
<td>128</td>
<td>242.1</td>
</tr>
<tr>
<td>fifo4</td>
<td>595</td>
<td>330</td>
<td>MEMOUT</td>
</tr>
<tr>
<td>fifo2-init</td>
<td>155</td>
<td>87</td>
<td>142.6</td>
</tr>
<tr>
<td>fifo3-init</td>
<td>110</td>
<td>65</td>
<td>69.9</td>
</tr>
<tr>
<td>fifo4-init</td>
<td>308</td>
<td>173</td>
<td>MEMOUT</td>
</tr>
<tr>
<td>vga1-init</td>
<td>148</td>
<td>2478</td>
<td>MEMOUT</td>
</tr>
<tr>
<td>vga2-init</td>
<td>70</td>
<td>1216</td>
<td>810</td>
</tr>
<tr>
<td>vga3-init</td>
<td>130</td>
<td>2210</td>
<td>MEMOUT</td>
</tr>
<tr>
<td>vga4-init</td>
<td>58</td>
<td>1010</td>
<td>496.4</td>
</tr>
<tr>
<td>vga5-init</td>
<td>103</td>
<td>1746</td>
<td>MEMOUT</td>
</tr>
<tr>
<td>vga6-init</td>
<td>82</td>
<td>1401</td>
<td>163.7</td>
</tr>
</tbody>
</table>
Table 5.3: Non-cycle accurate design debugging results on \textit{mrisc}

<table>
<thead>
<tr>
<th>Instance Info</th>
<th>Explicit</th>
<th>Bus Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>instance</td>
<td># clks</td>
</tr>
<tr>
<td>mrisc1</td>
<td>19</td>
<td>87</td>
</tr>
<tr>
<td>mrisc2</td>
<td>22</td>
<td>96</td>
</tr>
<tr>
<td>mrisc3</td>
<td>19</td>
<td>87</td>
</tr>
<tr>
<td>mrisc4</td>
<td>16</td>
<td>74</td>
</tr>
<tr>
<td>mrisc5</td>
<td>31</td>
<td>124</td>
</tr>
</tbody>
</table>

level model. The first three columns of Table 5.3 indicate the instance name, trace length and memory usage excluding the memory model size. The next four columns show the run-time in seconds and size of memory model in MBs for both the explicit memory representation and the proposed memory model. The last column shows the value of $B$ needed to find the design bug using our method. Again, the run-times for the debugging memory model are a sum of each run from $B = 1$ to the value of $B$ shown in the table.

For \textit{mrisc1}, the bus model results in a significant reduction in the memory model size from 199MB in the explicit model to 6MB. The run-time is about 17 seconds for both models in this instance. The explicit state model performs better on some instances such as \textit{mrisc3}, but our method is faster in others. This is an acceptable trade-off in run-time for a 98% reduction in memory usage. Furthermore, for designs with larger memories, the size of the explicit memory representation will affect the run-time to a larger degree, ultimately degrading its performance.

Notice that in Table 5.3, the values of $B$ to find the bug are overall higher than in those in Table 5.2. The reason is that several memory locations in the final state of memory are not written to during the simulation of the erroneous design. Thus, more bus variables need to be added to ensure that each location in the final state of memory has a valid forwarding.
Table 5.4: Debugging with interpolants circuit statistics

<table>
<thead>
<tr>
<th>instance</th>
<th># gates</th>
<th>total suspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac97</td>
<td>25,314</td>
<td>1,086</td>
</tr>
<tr>
<td>divider</td>
<td>5,799</td>
<td>1,092</td>
</tr>
<tr>
<td>fdct</td>
<td>377,849</td>
<td>4,568</td>
</tr>
<tr>
<td>fpu</td>
<td>81,303</td>
<td>939</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>46,425</td>
<td>2,451</td>
</tr>
<tr>
<td>mrisc</td>
<td>18,034</td>
<td>631</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>11,380</td>
<td>1,623</td>
</tr>
<tr>
<td>spi</td>
<td>2,103</td>
<td>223</td>
</tr>
<tr>
<td>vga</td>
<td>154,213</td>
<td>1,337</td>
</tr>
<tr>
<td>wb</td>
<td>3,552</td>
<td>407</td>
</tr>
</tbody>
</table>

5.3 Debugging with Interpolation Results

This section presents experimental results for the proposed scalable SAT-based debugging algorithm as well as the algorithm to generate multiple interpolants. The results are compared to the SAT-based debugging work in [37] for the entire error trace which will be denoted as orig in this section. MiniSAT-v1.14 [16] with proof logging is used to solve the SAT instances and as well as generate the UNSAT cores. Experiments were run on a Pentium Core 2, 2.4 GHz workstation with 8GB of memory with a timeout of 7200 seconds.

The effectiveness of our algorithm on large designs from OpenCores.org [33] is shown. Instances are generated by inserting a common RTL error such as a wrong assignment, missing case statement or incorrect operator. The error trace for each instance is generated by simulating the erroneous circuit through its testbench. Each suspect corresponds to a location in the RTL that can be corrected to satisfy the error trace.

Table 5.4 shows information for each of the designs used in the experimental results. The three columns show the instance name, the gate count of the design and the total number of
Table 5.5: Debugging with interpolants results

<table>
<thead>
<tr>
<th>Instance Info</th>
<th>Orig</th>
<th>Interpolant, r=2</th>
<th>Interpolant, r=3</th>
<th>Interpolant, r=4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>mem</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s)</td>
<td>(MB)</td>
<td>(s)</td>
</tr>
<tr>
<td>ac971</td>
<td>675</td>
<td>588</td>
<td>6,040</td>
<td>34</td>
</tr>
<tr>
<td>ac972</td>
<td>300</td>
<td>314</td>
<td>2,674</td>
<td>41</td>
</tr>
<tr>
<td>divider1</td>
<td>40</td>
<td>10</td>
<td>180</td>
<td>32</td>
</tr>
<tr>
<td>divider2</td>
<td>40</td>
<td>5</td>
<td>188</td>
<td>21</td>
</tr>
<tr>
<td>fdct1</td>
<td>40</td>
<td>TIMEOUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdct2</td>
<td>40</td>
<td>TIMEOUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpu2</td>
<td>312</td>
<td>MEMOUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpu5</td>
<td>300</td>
<td>MEMOUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem_ctrl1</td>
<td>100</td>
<td>174</td>
<td>2,901</td>
<td>12</td>
</tr>
<tr>
<td>mem_ctrl2</td>
<td>100</td>
<td>94</td>
<td>3,012</td>
<td>6</td>
</tr>
<tr>
<td>mrisc1</td>
<td>42</td>
<td>31</td>
<td>546</td>
<td>61</td>
</tr>
<tr>
<td>rsdecoder2</td>
<td>196</td>
<td>20</td>
<td>393</td>
<td>47</td>
</tr>
<tr>
<td>spi1</td>
<td>576</td>
<td>270</td>
<td>871</td>
<td>27</td>
</tr>
<tr>
<td>vga1</td>
<td>40</td>
<td>203</td>
<td>5,150</td>
<td>9</td>
</tr>
<tr>
<td>vga2</td>
<td>40</td>
<td>383</td>
<td>5,187</td>
<td>30</td>
</tr>
<tr>
<td>wb1</td>
<td>132</td>
<td>6</td>
<td>240</td>
<td>8</td>
</tr>
<tr>
<td>wb2</td>
<td>132</td>
<td>5</td>
<td>233</td>
<td>5</td>
</tr>
</tbody>
</table>
potential suspects.

Table 5.5 presents the results for the proposed debugging algorithm with interpolants. Four different sets of experiments are shown in this table. The first set of experiments in columns 3-5 correspond to running SAT-based debugging on the entire error trace (orig). The other three sets of experiments in column 7-14 correspond to debugging with interpolants varying the number of iterations \( r = \lceil k/\text{step} \rceil \) of the loop in Algorithm 4.1, ranging from 2 to 4. Each run uses one interpolant.

The first two columns in Table 5.5 show the instance name, number of clock cycles in the error trace. The next 12 columns show the run-time, peak memory and number of suspects returned for the four sets of experiments. For run-time and peak memory, the column with the lowest value is emphasized in bold.

For \( r = 2 \), the proposed algorithm shows on average a 24% decrease in run-time and 34% decrease in peak memory compared to orig, while increasing the number of suspects returned relative the total number of suspects on average by only 1%. With \( r = 3 \), the decrease in run-time is 26%, peak memory 48% and relative increase in suspects is 3%. \( r = 4 \) shows a similar trend by decreasing run-time by 23%, peak memory by 57%, but the relative increase in suspects is only 2% on average.

Figure 5.4 plots the run-time results from Table 5.5 from two different views. Figure 5.4(a) shows performance results of debugging with interpolants against orig on a log-log scale. Most points lie below the 45 degree line indicating faster runs on average. However, for several instances orig runs faster. In addition, \textit{fdct1} and \textit{fdct2} timed-out with orig while debugging with interpolants were able to successfully solve these instances.

Taking a closer look at how run-time varies with the number of windows, \( r \), Figure 5.4(b) shows how relative run-times of several designs vary with an increased \( r \). The run-times are normalized to the orig instance indicated by \( r = 1 \). While most instances, show a reduction in run-time with larger \( r \), \textit{vga2} shows an increase. This can be attributed to the fact that the run-time of a debugging instance does not necessarily scale linearly with the problem size. However, most instances show a decrease in run-time as \( r \) is increased.
Chapter 5. Experimental Results

Figure 5.4: Performance results for debugging with interpolants
Figure 5.5 shows the benefit of using interpolants with respect to peak memory. Figure 5.5(a) shows the memory of using interpolants against orig on a log-log scale. All instances are below the 45 degree line indicating that they consistently require less memory.

Looking more carefully at the relative memory usage for several instances, Figure 5.5(b) shows that the memory does not necessarily decrease inversely with $r$. *spi1* has a small relative increase from $r = 2$ to 3 but a much bigger relative decrease from $r = 3$ to 4. The reason for this is that the interpolant, which is not necessarily linear in size with the debugging instance, contributes to the peak memory. However, *ac972* shows a case where it does follow an inverse relation with $r$.

One would expect the early exit (line 14 from Algorithm 4.1) to contribute to this inverse relation. However, only the *mem_ctrl* and *wb* instances as well as *fpu2* used the early exit condition. This shows that the decrease in memory is due to the interpolant being significantly smaller than the instance it was generated from.

From Table 5.5, it can be seen that the number of suspects found generally increases as the $r$ increases. This was explained in Section 4.5 due to potentially weaker interpolants being generated in later iterations of Algorithm 4.1. However, *spi1* and *vga2* show a case where the number of suspects actually decrease with increasing $r$. With $r = 3$ *spi1* generated 79 suspects while at $r = 4$ it generated 38. Similarly *vga2* had 128 at $r = 3$ and 115 at $r = 4$. These results suggest that although the interpolant is more likely to get weaker with increased $r$, how well it constrains the erroneous behavior can vary a great deal depending on the UNSAT core that was used.

Figure 5.6 shows the results from using multiple interpolants with $r = 4$ to constrain the debugging problem. The instances shown in this figure are ones where the number of suspects increased by a large amount over orig. For *spi1*, *vga1* and *vga2*, using multiple interpolants improved the quality of the debugging results by reducing the number of suspects. *vga1* shows a dramatic improvement in the number suspects returned where the interpolants help the suspects converge to the same value as orig. However, in some cases such as *divider2* and *mrisc*, multiple interpolants did not help constrain the problem further. These results show that the effectiveness of multiple interpolants is highly dependent on the debugging instance,
Figure 5.5: Memory results for debugging with interpolants
where in some cases it can dramatically improve the resolution, while in others it does not help.

Table 5.6 shows the experimental results from Figure 5.6. For most of the instances, the memory requirement is close to the ones in Table 5.5. An outlier in this case is \texttt{spi1} which used more memory. This is caused by the large size of the UNSAT proof for this particular instance. As seen in Table 5.5, having a smaller \textit{r} does not necessarily translate into smaller memory usage. However, the dramatic decrease in the number of solutions shows that this trade-off may be acceptable in some cases such as \texttt{vga1} and \texttt{vga2}. 

Figure 5.6: Solutions using multiple interpolants
### Table 5.6: Debugging with multiple interpolant results

<table>
<thead>
<tr>
<th></th>
<th>time (sec)</th>
<th>mem (MB)</th>
<th>solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>divider2</td>
<td>7</td>
<td>98</td>
<td>38</td>
</tr>
<tr>
<td>spi1</td>
<td>491</td>
<td>1467</td>
<td>37</td>
</tr>
<tr>
<td>mrisc1</td>
<td>54</td>
<td>270</td>
<td>93</td>
</tr>
<tr>
<td>vga1</td>
<td>775</td>
<td>2055</td>
<td>9</td>
</tr>
<tr>
<td>vga2</td>
<td>2193</td>
<td>2290</td>
<td>84</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions and Future Work

6.1 Contributions

Debugging today remains one of the few steps in the VLSI design cycle that is still performed manually. Despite the numerous contributions to automated debugging techniques, much work still needs to be done to scale these algorithms to handle industrial strength designs. Two important factors contribute to this scaling problem. One is the size of the designs. The second is the length of the error trace that needs to be analyzed.

The purpose of this thesis is to present two techniques to handle these two problems. The first presents an efficient embedded memory model model for automated design debugging in an effort to handle the growing size of modern designs. The second presents a novel automated debugging algorithm that uses a mathematical technique known as interpolation to help reduce the number of simultaneous clock cycles that need to be analyzed. By addressing two key factors, the size of the design with embedded memories and the number of clock cycles with interpolation, this thesis attempts to take automated debugging techniques one step closer to solving practical industrial problems.

Although automated debugging techniques may not yet be able to handle a full chip designs, there are still practical situations where they can be used. Designs are usually divided into many smaller blocks. These are usually designed and verified before being put together. Common block sizes can be similar to the size of many of the designs used in the experiments in this
thesis. In addition, formal tools typically produce error traces that are tens of clock cycles long instead of thousands. In these niche situations, automated debugging techniques can already be applied with current computational resources. The contributions in this thesis take the limit of what automated techniques can handle a couple steps further which will one day hopefully be able to handle full chips with error traces of thousands of cycles long.

In summary, this work makes the following contributions:

- In Chapter 3, a novel memory model for design debugging is presented. It models memory succinctly by avoiding an explicit representation for each memory bit. The method uses the simulation of the erroneous design to guide the debugging process. This results in a parameterizable formal encoding that grows linearly with the erroneous trace length, significantly reducing the memory requirements of the debugging problem. In addition, the proposed model is extended to handle an arbitrary initial memory configuration, as well as non-cycle accurate output traces where only a final expected memory state is available for comparison.

- In Chapter 4, a scalable SAT-based design debugging algorithm that uses interpolants to over-approximate sets of constraints that model the erroneous behavior. The algorithm partitions the original problem into a sequence of smaller subproblems by using subsections of the error trace that are examined iteratively. This is made possible by using interpolants to properly constrain the erroneous behavior for each subproblem, significantly reducing the number of simultaneous time-frames examined in the error trace. The described method is shown to be complete and an additional technique is presented to improve the quality of the debugging results using multiple interpolants.

6.2 Future Work

The contributions of this thesis rely heavily on many well established formal techniques and technology. The field of formal methods is constantly introducing new and innovative ways to solve CAD problems. To this end, there is much promise in studying these techniques in greater
detail, specifically related to the applications for formal methods described in this thesis. Two areas for future research directions are given. The first relates to modelling embedded memories and the second relates to UNSAT cores and interpolation.

The first future direction relates to a generalization of SAT called Satisfiability Modulo Theories (SMT) [14]. It holds much promise with modelling embedded memories. Instead of a decision problem only allowing predicates over Boolean variables (or theory of Booleans), SMT allows predicates over a suitable set of non-binary values. An example is a decision procedure using the theory of bit-vectors instead of just Boolean. A different type of decision problem is generated for each combination of theories.

Recently, there has been a great deal of research in this field especially in SMT solvers. The field is experiencing a great deal of interest similar to how SAT was not so long ago. If SMT solvers are able to scale, they have the potential to become practical for industrial problems.

A notable theory of interest to modelling embedded memories is the theory of arrays. In this theory, one can access an element of an array with a single constraint i.e. without the addition of many new clauses to model the embedded memory. A popular solver [8] uses under-approximation and refinement techniques to iteratively handle this theory. This solver handles a general SMT problem, however, using the ideas presented in this thesis, perhaps a more efficient debugging instance can be generated in SMT that still retains the efficiency seen in SAT.

The second future direction relates to the underlying techniques in Chapter 4. UNSAT cores and interpolation have been used extensively in model checking applications while there has not been much study of them relating to debugging. One important idea is that the UNSAT core describes how the problem is unsatisfiable i.e. why the error conflicts with the expected value. This information is used in this thesis to over-approximate time-frames in the suffix but it can also have many other applications.

For instance, many industrial designs have assumptions for valid primary input vectors. One important problem is where the designer has forgotten to encode these assumptions formally in
a property. This causes formal tools to produce an error trace that does not accurately reflect
the desired operation of the design. Perhaps UNSAT cores or interpolants can be used here
to help debug the missing assumption. Another interesting area where this could be used is
correction. Since the UNSAT cores give information about what caused the error, this could be useful in generating a new function to replace a candidate error location.

UNSAT cores and interpolants give important information about the debugging problem. They have been used in many other applications but have only begun to be researched in the field of automated debugging. An in depth study into how they relate to errors will produce very interesting results that will surely produce new avenues for future research.

In addition, a specific direction for further research is improving the effectiveness of using the interpolant in the debugging work presented in this thesis. Since the interpolant is an approximation, having a more accurate approximation will yield better results in terms of debugging resolution. A couple of ways to do this include rewriting the UNSAT proof to generate a stronger interpolant and computing interpolants with respect to the middle of a window instead of the beginning. Both of these might yield interesting applications that will further enhance the efficacy of the debugging with interpolants algorithm presented in this thesis.
Bibliography


