Practical Considerations for Post-Silicon Debug using BackSpace

by

Marcel Gort

B.Sc., The University of Western Ontario, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2009

© Marcel Gort 2009

Abstract

With the ever-increasing complexity of integrated circuits, the elimination of all design errors before fabrication is becoming more difficult. This increases the need to find design errors in chips after fabrication. This task, termed post-silicon debug, can be made easier if it is possible to obtain a trace of states that leads to a known state. BackSpace, a proposal for a new debug infrastructure which provides such a trace has been recently presented. BackSpace combines formal analysis with on-chip instrumentation. In this thesis, we show that BackSpace can be made practical by modifying the architecture and debug flow to address the area overhead, and also by addressing on-chip realities such as non-determinism and signal propagation delay. Additionally, this thesis describes a proof-of-concept implementation of a complex processor instrumented with the debug architecture and shows that BackSpace can produce traces hundreds of cycles long. Our results indicate that the area overhead of the breakpoint circuit, a main component of the debug architecture, can be reduced to 5% for our prototype, while still allowing the debug flow to create state-accurate traces.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables viii

List of Figures ix

Acknowledgements xii

Introduction 1

0.1 Motivation 1

0.2 BackSpace 2

0.3 Contributions 3

0.3.1 Debug Flow 3

0.3.2 Area Model 4

0.3.3 Prototype 4

0.3.4 Bit Selection for Prototype 4

0.4 Thesis Organization 5

1 Background and Previous Works 6

1.1 Previous Work 6

1.1.1 Software-Based Debug 7

1.1.2 Using Existing Test Features 7

1.1.3 In-Circuit Emulation 8

1.1.4 On-Chip Emulation 8

1.1.4.1 Width Compression 9

1.1.4.2 Depth Compression 9

1.1.4.3 Post-Processing 10

1.1.4.4 Breakpointing 11

1.1.4.5 Industry Debug Methods 12

1.1.4.6 Reconfigurable Debug Architectures 12

1.2 BackSpace 13

1.2.1 Debug Flow using BackSpace 13

1.2.2 On-Chip Hardware 16

1.2.3 Practical Considerations 17

2 Practical Debug Flow 18

2.1 Deterministic Architecture 19

2.2 Non-deterministic Architecture 21

2.2.1 About Non-Determinism 21

2.2.2 Full Breakpoint 21

2.2.2.1 Architecture 22

2.3 Partial Matches 23

2.3.1 Spatial and Temporal Mismatches 24

2.4 Coping with Non-Determinism by Scanning Out 24

2.5 One Counter Method with Scan Out 26

2.6 Two Counter Method with Scan Out 28

2.7 Delayed Breakpoints 35

2.7.1 Pipelining the Breakpoint Circuit 36

2.7.2 Debug Flow with Delayed Scan Out 37

2.7.3 Repeating the Same Mistake 42

2.7.4 Blind Spot Problem . . . 42
2.7.5 Blind Spot Solution . . . 43
2.8 Selecting Bits . . . 44
2.9 Chapter Summary . . . 44

3 Area Considerations . . . 52
3.1 Area Model Calibration . . . 52
3.2 Breakpoint Circuitry . . . 53
3.2.1 Full Breakpoint . . . 53
3.2.2 Partial Breakpoint . . . 54
3.2.3 Two Partial Breakpoints . . . 55
3.3 Signature Creation . . . 55
3.3.1 Hard-Wired Bits . . . 56
3.3.2 Concentrator . . . 56
3.3.3 Coarse-Grained Concentrator . . . 58
3.3.4 Hash Function . . . 59
3.4 Signature Collection . . . 62
3.5 Tradeoffs . . . 63
3.5.1 Area as a Function of S_{width} . . . 64
3.5.2 Signature Creation Circuit Comparison with Differing S_{width} Values . 65
3.5.3 Area Contribution of Each Component . . . 66
3.5.4 Breakpoint Circuit Area . . . 67
3.5.5 Area Overhead of Prototype . . . 68
3.6 Chapter Summary . . . 69

4 Prototype and Breakpoint Bit Selection . . . 73
4.1 Motivation for Prototype . . . 73
4.2 OpenRISC 1200 . . . 74

4.3 Debugging Architecture Insertion 75
4.4 System Operation 77
4.4.1 System Hardware 77
4.4.2 Communication between BackSpace and the Hardware 80
4.5 On-Chip Non-Determinism 81
4.6 Preliminary Results 81
4.7 Partial Breakpoint Bit Choice 82
4.7.1 Choosing Bits Randomly 83
4.7.1.1 Modeling Random Bit Choice with Real Data 84
4.7.2 Choosing Bits Intelligently 85
4.7.2.1 Characterizing Spurious Runs 86
4.7.2.2 Gathering Differing Bit Data 86
4.7.2.3 Results 88
4.8 Chapter Summary 90

5 Conclusion 92
5.1 Summary 92
5.2 Contributions 93
5.3 Limitations and Future Work 94
5.3.1 Reducing Area Overhead 94
5.3.2 Automating Breakpoint Bit Selection 95
5.3.3 Studying and Coping with Non-Determinism 96
5.3.4 Multiple-Bug Scenarios 96

Bibliography 98

Appendices

A Usage of Software Registers 103

B Bit Order Tables 104

List of Tables

3.1 Area per Bit Values in 0.18um for Architecture Components 53

A.1 Uses of Software Registers 103

B.1 Hamming Bits 105

B.2 Weighted Breakpoint Bits 106

List of Figures

1.1 Debug Flow using BackSpace 14

1.2 Debugging Architecture 16

2.1 Debug Flow using a Cycle Counter as the Breakpoint Bits 19

2.2 Breakpoint Architecture using a 64-Bit Cycle Counter 20

2.3 Debug Flow using Full Breakpoint 22

2.4 Breakpoint Circuit 23

2.5 Spatial and Temporal False Matches 24

2.6 Partial Breakpoint Circuit 26

2.7 Debug Flow using One Partial Breakpoint Counter 27

2.8 The Inability to Differentiate Temporal and Spatial False Matches can Result in Skipping the Correct Match 28

2.9 Overall Debug Flow using Two Partial Breakpoints 30

2.10 Flowchart Detailing the Operation of Mode 1 31

2.11 Flowchart Detailing the Operation of Mode 2 32

2.12 Flowchart Detailing the Operation of Mode 3 33

2.13 Two Partial Breakpoints Circuit 35

2.14 The Comparator Portion of a Breakpoint Circuit with One Pipelining Stage 36

2.15 Relation between Defined States in a Trace Generated by BackSpace and a Run On-Chip 38

2.16 Converging Paths with Different Breakpoint State but the Same Frozen State 39

2.17 Modified Version of Mode 1 Taking Delayed Scan-Out into Account 40

2.18 Modified Version of Mode 2 Taking Delayed Scan-Out into Account 41
2.19 BackSpace Trace and On-Chip Run Connecting to Form a Valid Path from Reset to Crash 42
2.20 Debug Flow Used to Deal with BackSpace Going down Unreachable Paths . 43
2.21 Mode 1 for Debug Flow Coping with Unreachable Paths 46
2.22 Mode 2 for Debug Flow Coping with Unreachable Paths 47
2.23 Frozen State Occurring after Latest State in Trace 47
2.24 Flowchart Detailing the Blind Operation of Mode 1 48
2.25 Flowchart Detailing the Blind Operation of Mode 2 49
2.26 Flowchart Detailing the Blind Operation of Mode 3 50
2.27 Overall Debug Flow using Two Partial Breakpoints with a Blind Zone 51

3.1 Using a Concentrator as the Signature Creation Circuit 56
3.2 Concentrator Construction 57
3.3 Coarse Grained Concentrator with $k = 4$, $m = 1$, and $n = 2$ 59
3.4 Concentrator Area for Different Set Sizes using N_{mon}=10000, S_{width}=0.4 and C_{cycles}=1 60
3.5 Hash Function Matrix and Corresponding Circuit 61
3.6 Construction of 5-Input XOR Gate using 4 2-Input XOR Gates 62
3.7 Signature Creation Circuit Area for $Nmon$ of 10000 65
3.8 S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.2 . . 67
3.9 S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.4 . . 68
3.10 S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.6 . . 69
3.11 S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.8 . . 70
3.12 S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 1.0 . . 71
3.13 Area Contribution of Architecture Components using $N_{mon} = 10000$, $S_{width}/N_{mon} = 0.25$ and a Concentrator with a Coarseness of 16 71
3.14 Breakpoint Area vs $bp_{fraction}$ 72

3.15 Prototype Area Overhead Contribution vs $bp_{fraction}$ 72

4.1 Breakpoint Masking for Partial Breakpoints 76
4.2 Loading of Target State Register using Software Registers 79
4.3 System Level View of Debugging Methodology 80
4.4 Waveform Viewer Representation of Trace Produced by BackSpace 82
4.5 Probability of a Spurious False Match vs. $bp_{fraction}$ 85
4.6 Probability of False Match vs. Number of Breakpoint Bits, Prioritizing Bits using Probability of Bits Differing 89
4.7 Probability of False Match vs. Number of Breakpoint Bits, Prioritizing Bits using Correlation-Based Weight Function 90
4.8 Average Probability of False Match vs. Number of Breakpoint Bits for Both Prioritizing Methods 91

Acknowledgements

I would like to thank my research supervisor, Dr. Steve Wilton, for his support throughout my degree. I'm indebted to him for the opportunity to study under his supervision, as well as for his continual commitment to his students and his strong technical guidance.

I would also like to thank Dr. Alan Hu and Dr. Tor Aamodt, who, in addition to being on my supervisory committee, provided guidance and valuable perspective throughout the course of my research by way of courses and meetings.

I would like to thank Flavio De Paula, with whom I constantly collaborated and brainstormed. His friendly manner and his willingness to discuss ideas in an open way were conducive to making progress even when stress levels were high.

Additionally, I would like to thank Johnny Kuan for his technical contributions, including his idea for a two-breakpoint system, which influenced the direction of my research.

I cannot overstate how much I appreciate the collaborative atmosphere of the SoC group at UBC. The whiteboard sessions and ideas exchanged over coffee never failed to provide fresh perspectives. I would especially like to thank Scott Chin, Joydip Das, Eddie Hung, Brad Quinton, Alastair Smith, Julien Lamoureux, Cindy Mark, Andrew Lam and Suart Dueck, who were all members of Dr. Steve Wilton's research group at various times during my Master's.

Finally, I would like to thank my partner, Lauren, for being so supportive throughout the course of my Master's research. She was the main reason why I was able to remain sane when working consecutive long days. Additionally, her own academic ambition is strong motivation for me to succeed.

This thesis comprises one part of a two-part collaborative project. Some of the research

was done independently, while some of it was done in collaboration with Flavio De Paula. The research for Chapters 2 and 3 was performed independently from Flavio's research.

Flavio made several contributions to Chapter 4. He was responsible for the pre-image computation software running on the PC connected to the prototype. Additionally, he created a mailbox communication protocol that is used to enable communication between the PC and the FPGA prototype. Flavio wrote the code that uses the mailbox communication protocol on the PC side, but not on the prototype side.

Although I was responsible for implementing the OpenRISC 1200 in hardware, Flavio eased the process by first successfully simulating its operation in software. Additionally, Flavio cross-compiled the code for the benchmarks which ran on the OpenRISC 1200.

Two papers and two demonstrations have resulted from our work on BackSpace. The two papers both contain material written by Flavio de Paula, Dr. Alan Hu, Dr. Steve Wilton, and myself. The two presentations were collaborative efforts between Flavio and myself.

Introduction

0.1 Motivation

Integrated circuits are becoming increasingly complex. Chip designers are going to ever greater lengths to ensure that such devices operate as expected when they come back from fabrication. Despite the increase in effort, some bugs still make it onto silicon. This is not surprising; in the design of the Intel Pentium 4, researchers report that their simulation effort required 6000 processors, each running for two years, and that their effective simulation coverage was less than one minute of real time operation. Out of a total of 2^{10441} distinct processor configurations, only 2^{37} were covered in verification.[6]

Clearly, existing pre-silicon verification methods are not sufficient to uncover all the possible design errors that may exist in the device. Nonetheless, it is critically important to find bugs before products are shipped. Failing to do so can be costly. In a well known example, a bug in the floating-point division unit of the Pentium processor forced a complete recall of the product and cost Intel $475 Million [26].

The only way to find these bugs is to test manufactured chips. A real chip can run orders of magnitude faster than a simulated chip. Additionally, it can be attached to other integrated circuits in a larger system. This allows for much larger functional coverage.

Although achieving adequate functional coverage on manufactured chips is easier than achieving the same coverage on simulated chips, diagnosing the cause of these errors is much harder.

This process is termed *post-silicon debug*. Time-to-market pressure means that design errors must be discovered and diagnosed quickly and it is essential that as many errors as

possible are uncovered before the chip is subject to another lengthy and multi-million dollar manufacturing spin. Yet, finding these bugs is very challenging.

The fundamental problem is a lack of post-fabrication observability. The transistors on an I.C. must be observed using external pins, of which there are very few. This problem is exacerbated by the ever increasing number of transistors on an integrated circuit. The Intel Itanium 2 processor, for example, has up to 1.72 billion transistors but only 611 contacts with which to infer the behaviour of these transistors [9][34]. The magnitude of the problem is evident when considering that 35% of I.C. development time occurs after initial device fabrication, and this proportion continues to grow [1].

The lack of observability means that the success of the post-silicon debug process is dependent on the educated guess-work of a knowledgeable validation engineer. This variability in the amount of time required to diagnose bugs post-silicon motivates the need for a more systematic approach.

0.2 BackSpace

BackSpace is an off-chip algorithm that uses formal analysis techniques to compute a target state's possible predecessor states. BackSpace also uses on-chip information obtained during a run that leads to this target state to prune the set of predecessor states to a small set of states. It can eliminate the predecessor states whose flip flop values are not consistent with on-chip trace buffer information.

The reason for computing predecessor states is to create a trace of states that leads to a target state, which is critical for debugging. With BackSpace, both the information contained in the RTL and in on-chip trace buffers are combined to improve observability, and by extension, post-silicon debugging.

0.3 Contributions

Before beginning the work contained in this thesis, a basic on-chip architecture for BackSpace had already been conceived. That architecture consists of three components: a *Signature Collection Circuit*, a *Signature Creation Circuit*, and a *Breakpoint Circuit*. Designed as a starting point, this architecture is meant to provide the best observability and controllability for BackSpace, without considering practicality. This thesis follows up on this work and is about "practical considerations" for BackSpace.

0.3.1 Debug Flow

One of the main issues with BackSpace is the large area overhead associated with the Breakpoint Circuit. Because the full breakpoint circuit has to be able to differentiate between any two arbitrary states (state-accurate), it monitors every flip flop in the circuit under debug. This necessitates storing the target copy of each one of these flip flops, thereby doubling the total number of flip flops on the chip.

The first main contribution of this thesis is a method to make the debug methodology practical by addressing the potential area overhead while also coping with chip realities such as non-determinism and signal propagation delay. This is done with the development of a partial breakpoint circuit, which reduces the number of bits on which a breakpoint is triggered. The partial breakpoint circuit still allows the debug flow to be state-accurate but incurs a much smaller area overhead. A new debug flow was created to use this partial breakpoint circuit while also allowing for realistic multi-cycle breakpoint generation and distribution. Additionally, the debug flow is capable of coping with non-determinism on chip.

0.3.2 Area Model

The second main contribution of this thesis is an area model estimating the standard cell area of the three main components of the debugging architecture. The model estimates area based on several architecture parameters, which can be varied based on the target application. A range of parameters were examined to gain insight into the trade-off between the effectiveness of each of these debug component variations and their associated area-overhead.

0.3.3 Prototype

The third main contribution of this thesis is a hardware prototype which the BackSpace Algorithm can control. An open source 32-bit processor called the OpenRISC 1200 is instrumented at the gate level with our debugging architecture. We then incorporate the processor into a system-level Power-PC based design in order to ease communication with the Host PC running BackSpace. The system-level design is programmed onto an FPGA connected to the Host PC via a PCI bus and two serial ports. Traces of hundreds of cycles in length are built backwards from different simulated Crash States using BackSpace on the hardware prototype.

0.3.4 Bit Selection for Prototype

The fourth main contribution of this thesis is a demonstration that it is possible to choose a small set of bits to use with the partial breakpoint circuit. We develop an algorithm to prioritize the OpenRISC's 3007 state bits based on how effective those bits are as breakpoint bits. The algorithm uses a weight function based on experimental data gathered from the prototype. Using this algorithm, we find an effective set of 46 breakpoint bits which leads to a 5% area overhead for our prototype.

0.4 Thesis Organization

This thesis is organized as follows. Chapter 2 presents an overview of background topics and related works, including a description of the BackSpace algorithm. Chapter 3 describes a debug flow which addresses practical concerns with the BackSpace debug methodology. Chapter 4 presents an area model used to estimate the standard cell area for each component of the debug architecture. Chapter 5 describes an FPGA prototype of our debug architecture implemented on the OpenRISC 1200 32-bit processor. Additionally, this chapter shows how a small effective set of breakpoint bits can be chosen for the partial breakpoint circuit on the prototype. A summary of the thesis along with limitations of the debug architecture and suggestions for possible future work are presented in Chapter 6.

Chapter 1

Background and Previous Works

In this chapter, we present an overview of techniques used to debug integrated circuits after they have been fabricated. We begin with a discussion of academic work and continue with a review of debugging techniques that are used in industry. Additionally, a brief explanation of the BackSpace algorithm, a part of Flavio De Paula's PhD research, is provided. The nature of BackSpace motivates many of the architecture decisions presented in this thesis.

1.1 Previous Work

The goal of a silicon-debug solution is to help a validation engineer debug problems on a chip after it is fabricated. This can be accomplished by increasing the speed at which a debugging engineer is able to find bugs and by making the post-silicon debug process more predictable. These benefits come at the cost of either the on-chip area/pin overhead incurred, or the off-chip test equipment necessary to debug the chip. There are a range of currently used silicon debug solutions, operating at different points in the trade off between cost and debugging effectiveness.

Post-silicon debug solutions can be categorized as follows: software-based with basic hardware support, using device test features, in-circuit emulation, and on-chip emulation [14]. Although on-chip emulation requires the most on-chip area of these methods, it has many advantages, especially in more complicated SoC designs. Without the dedicated on-chip debug logic provided by on-chip emulation, it is very difficult to get adequate observability onto complicated SoCs. For this reason, most new debugging solutions are based on on-chip emulation [12] [32] [33]. Because the debugging approach presented in this thesis is

also based on on-chip emulation, we will provide detailed background on current on-chip emulation techniques and provide a less detailed description of the other three types of silicon debug techniques.

1.1.1 Software-Based Debug

Software-based debug methods rely either on the insertion of debug instructions such as print statements, entering a debug mode (using a software debugger such as gdb), or using an interrupt handler [3]. This approach is limited in that a system must run long enough to allow the processor to begin executing debug instructions before reaching the bug. Additionally, access points are limited to memory-mapped resources visible by the processor [14], which is inadequate for complicated I.C.s that consist of more than a processor. This method is also very intrusive, and has the potential to alter the behaviour of the underlying system so that the bug is no longer reproducible [27].

1.1.2 Using Existing Test Features

It is typical to find design for test (DFT) features on most modern I.C.s to assist in their manufacturing test [44] [13]. Using a test mode, flip flops that are part of scan chains can be frozen, allowing their values to be read-out serially (scanned out). This can provide very good observability into the state of a chip when it was stopped. Failing elements of a scanned out state can be identified using latch divergence analysis [10] or failure propagation tracing [8] techniques. Since scan chains are generally already present on modern I.C.s, using them for debug makes good use of already committed silicon area. Although the observability provided by a scan chain is generally very good for a single state, it does not provide the ability to observe how states change over multiple cycles. The ability to single step can help alleviate this problem, but like the software based method, has the potential to alter the behaviour of the underlying system.

1.1.3 In-Circuit Emulation

In circuit emulation consists of creating a more easily debuggable version of an integrated circuit called a bond-out chip. A bond-out chip provides additional observability and more I/O than the production version of the same integrated circuit [38]. The cost incurred by the additional area and pin overhead is not as important because it will not be on the final version of the integrated circuit. Unfortunately, because the bond-out chip can be significantly different than the production version, there is no guarantee that these two version of a design share the same bugs. A debugging engineer may end up debugging timing bugs on the bond-out version that will not exist on the production version of the chip. Worse still, bugs that do not occur on the bond-out version of the chip but do occur on the production version may be missed. Another problem with a bond-out chip is that the behaviour of the chip is being observed through the I/O, which is slow compared to the chip. As a result, a lower clock speed needs to be used for the chip, which can affect its behaviour, again potentially resulting in missed bugs [27].

1.1.4 On-Chip Emulation

The on-chip emulation approach to silicon debugging seeks to capture relevant data on-chip, rather than make it immediately available at the chip outputs [19]. The start or end point of the data capture is typically controlled using some trigger logic, called a breakpoint. Once the data is successfully captured, it can be safely off-loaded from the chip for post-mortem analysis [14]. The main benefit of this method is that debugging the chip does not necessarily alter its behaviour. Assuming the debugging hardware can run at the chip's full speed, consecutive cycles of data can be captured without slowing the chip down, avoiding a potential change in its behaviour. This provides insight into the chip's state leading up to a bug.

Debugging techniques that fall within the on-chip emulation category share a common shortcoming in area overhead. Because any debugging hardware that is added to the chip

may remain in the production version of the chip, it is important to minimize and fully utilize this hardware. The most direct way of doing this is to reduce the amount of data that needs to be captured. There are several methods for reducing the amount of data that must be stored in the on-chip trace buffer. These methods can be split into depth compression or width compression methods.

1.1.4.1 Width Compression

Width compression techniques aim to reduce the number of bits captured at each cycle of the trace while maintaining the amount of useful information provided by those captured bits. Alternatively, the number of bits captured can remain the same if the useful information provided by these bits increases. The net effect is either a reduction in the size of the trace buffer or an increase in the amount of useful information that can be captured. In [5], several methods of lossless compression are compared to determine their suitability for on-chip use. It is found that dictionary encoding can provide a reduction in the size of the trace buffer for very long traces ($11.37 * 2^{20}$ 16KB samples). This highlights a key challenge in width compression: the compression hardware must generally be amortized over many cycles to see a reduction in the total amount of on-chip area. The IBM Cell Broadband Engine also uses a form of width compression. [33]

1.1.4.2 Depth Compression

The size of the observation window provided by a debug solution is the number of cycles of trace information that can be captured in a trace buffer. Depth compression aims to reduce the number of cycles of information stored in the trace buffer while increasing the size of the observation window.

In [4], the authors attempt to collect only the cycles which are buggy, thereby avoiding the capture of error-free states, which leads to a much larger observation window. The first debug session is run in a compressed mode, using a lossy width compression method

to increase the size of the observation window. The compressed data is then compared to corresponding error-free simulated data off-chip to determine which cycles are error-free. The second debug session is run storing only the cycles which were shown to have errors.

A similar approach is used in [43] with an extra initial pass to determine the size of the observation window that a given trace buffer can support. During the initial run, a parity check is performed on every cycle and compared against error-free simulated values off-chip. The number of erroneous parity values provides an indication of the average error rate on-chip. Using this error rate and the size of the on-chip trace buffer, the number of cycles that will elapse before the trace buffer is full of erroneous states is computed. Using this size, the start point of the observation window is set. The second pass identifies suspect states using multiple-input shift register (MISR) registers and cycling register signatures. The third pass collects these suspect states in the trace buffer.

1.1.4.3 Post-Processing

Strongly tied to the compression of trace information is the processing of that information off-chip. In [15], an algorithm is presented which can infer the values of combinational nodes from state information captured on-chip. Because debugging is easier at the RTL level, the gate-level nodes are then mapped to their RTL counterparts. The authors also use knowledge of the algorithm to find the set of signals which, if captured on chip, expand to provide the maximum amount of inferred observable nodes.

A similar approach is presented in [17], though the algorithm is capable of inferring more than combinational nodes. This is done by forward propagating and backward justifying captured state bits, which reveals information about other state bits. In addition to this technique, they also propose a trace signal selection method, where the signals that are stored result in greater success for the state restoration algorithm.

1.1.4.4 Breakpointing

The method of determining when to start and stop the trace collection is also an area of active research. In addition to breakpoint techniques that aim to improve hardware debugging, there are a wide range of breakpoint solutions that aim to improve software debugging. Many processors have built-in debug modules that help users debug their applications. Existing breakpointing capabilities generally consist of observing chip signals, where a breakpoint is produced based on a certain boolean combination of these signals. Breakpoints can also be produced after a sequence of events or after a certain number of events occur.

The MIPS architecture [36] offers optional hardware breakpoint modules that can be configured to breakpoint when an instruction is executed anywhere in the virtual address space. Additionally, a breakpoint can be triggered on a data transaction to a software variable. The debug module also offers the ability to mask the breakpoint address or data value to offer finer control.

There has also been work on automatic generation of breakpoint circuits. In [16], a method of automatically configuring a breakpoint responsible for starting and/or stopping the collection of signals is presented. The trigger criteria is based upon a predetermined set of trigger signals which can be combined to produce a signal that will trigger data collection. If a designer wants to trigger data collection on a signal which is not part of the available predetermined signals, he/she must determine a combination of those trigger signals which infer the unavailable signal. The paper proposes a method of automatically finding a combination of the available trigger signals that offer the same breakpointing functionality as using the unavailable signal as a trigger signal.

In [39], a breakpoint description language (bdl) is presented, which can be used to help describe existing breakpoint techniques. The designer is responsible for specifying the conditions under which a breakpoint should be possible using bdl but the authors present software that will convert that description into a breakpoint module that has those capabilities.

To the best of our knowledge, there is no published work which provides the ability to

breakpoint a chip on any arbitrary state. This can make it difficult to breakpoint on specific condition that lead to an erroneous state.

1.1.4.5 Industry Debug Methods

Current on-chip emulation based debugging techniques used in industry typically rely on well-placed trace buffers and coarse grained breakpointing capabilities. Debugging techniques that rely on scan chains are also used to enhance observability onto the chip.

The multi-core AMD Opteron processor uses HyperTransport trace capture buffers (HTTB) to capture relevant intra-node and inter-node HyperTransport communication traffic. The HTTB is controlled by a set of debug registers which provide the ability to breakpoint on memory transactions such as instruction and data addresses, as well as IO values. [12]

The IBM Cell Broadband Engine uses a trace logic analyzer (TLA) to capture on-chip signals from one or more of the nine processing elements. An on-chip compression algorithm is also included to compress input data to the trace buffer so that more data can be observed. Up to 128 signals can be captured in the trace buffer every cycle. The breakpoint capabilities of the Cell allow for up to 4 event and trigger signals, which can be combined and/or counted to create a breakpoint condition [33].

1.1.4.6 Reconfigurable Debug Architectures

Debugging solutions that use some on-chip reconfigurability result in greater flexibility. This flexibility means that the decision of which signals to monitor can be made after fabrication, when information about the nature of the bug is available.

In [30], a debugging framework is presented which makes use of a programmable access network to route a set of signals to a central programmable logic core (PLC). Because the PLC is reconfigurable, custom trigger logic can be implemented after fabrication. A concentrator is used as an access network and an interface buffer is used to bridge the speed gap between the PLC and the rest of the I.C. A debugging engineer can implement

whatever debugging circuitry is necessary on the PLC in addition to choosing the set of signals connected to the PLC.

In [2], a reconfigurable distributed debug architecture is presented. It consists of a signal probe network, a debug monitor and a tracer. The debug architecture is implemented in a distributed reconfigurable fabric, which eases the routing of signals because they must not be routed as far. The debug monitor can programmably set triggers to start or stop the trace buffer and also supports assertions.

Although reconfigurable debug solutions offer greater flexibility, the signals which can be observed are still limited. Additionally, reconfigurable logic results in greater area overhead.

1.2 BackSpace

The BackSpace debug flow and architecture provides additional visibility into a malfunctioning integrated circuit [23]. Given a "crash state", the BackSpace flow can be used to obtain a *history of states* that led up to the crash state. This history of states would help a debug engineer understand the cause of observed incorrect behaviour. Theoretically, this history can be arbitrarily long.

The BackSpace flow, as originally described in [23], is summarized in this section. The remainder of this thesis examines practical considerations related to BackSpace, and presents refinements to this flow dictated by these practical considerations.

1.2.1 Debug Flow using BackSpace

Figure 1.1 shows the debug flow. We assume a crash state is known, either by scanning out the state after a crash, or through some other means.

This crash state is fed to an off-chip algorithm, which computes a set of potential *predecessor states* (also called the *pre-image states*) that could have occurred one or more cycles before the crash state. This off-chip algorithm, termed *BackSpace* in Figure 1.1, is the key

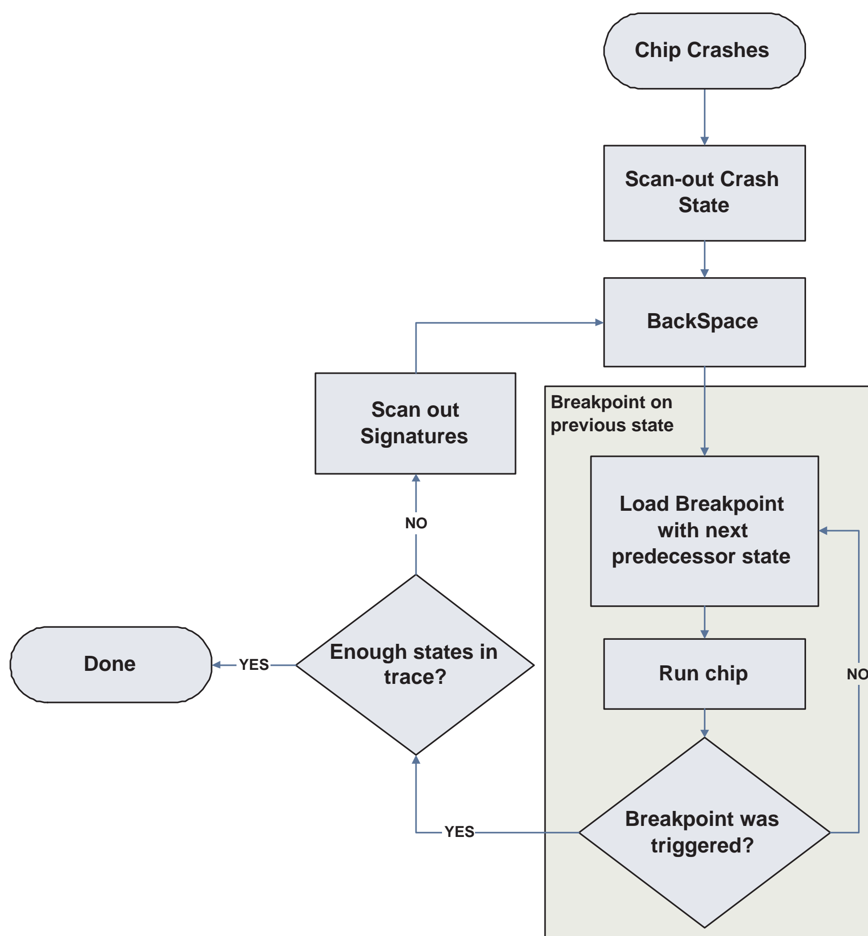

Figure 1.1: Debug Flow using BackSpace

enabling technology of our flow. The algorithm uses formal verification techniques to compute the pre-image state(s) of the crash state. Essentially, the algorithm "works backwards" through a state diagram. Since there is often more than one way to get to a given state, there will likely be more than one possible pre-image state. In extreme cases, such as a reset state, there may be many states in the pre-image, however in practice, the number of pre-image states is usually small. If the number of pre-image states is very small, it is possible to iterate through the off-chip algorithm calculating predecessor states that occurred n cycles before the crash state, where $n > 1$. However, in practice, for $n > 1$, the number of potential predecessor states grows very quickly.

Each potential predecessor state is then considered individually. The first potential prede-

cessor state is loaded into an on-chip breakpoint circuit, and the integrated circuit is re-run. If the chip passes through the potential predecessor state, the chip crashes, and we know that this potential predecessor state is, indeed, the correct predecessor state, so it is added to the history. If the chip does not pass through the potential predecessor state (indicated by a time-out circuit), the next potential predecessor state is tried, and the process repeated. Ideally, exactly one of the predecessor states will cause the breakpoint to be triggered. This predecessor state can then be used as input to the off-chip BackSpace algorithm, and the entire process repeated.

In practice, external events and non-determinism in the execution flow may mean that the execution times out, even though the correct predecessor state is loaded into the breakpoint register. This may happen if the execution path "misses" the target state. A simple solution is to re-run the chip until it does crash on one of the predecessor states. We will revisit the issue of non-determinism extensively in Chapter 3.

For this approach to be feasible, the number of predecessor states found by each invocation of the off-chip BackSpace algorithm should be small. We can "prune" the set of potential predecessor states by gathering additional information, called a *signature* during the execution of the chip, and using this information to rule out some of the potential predecessors. For example, if there are two predecessor states for a given state, and a single well-selected signature bit is gathered, this bit may allow us to rule out one of the two states. If there are more potential predecessor states, more bits may be required. The number of bits that make up the signature, and the way in which the signature is created, allows for a tradeoff between the area overhead of our approach and the number of runs required during each iteration of the debug flow. In Chapter 4, we present several alternative signature collection structures, and discuss their suitability and area overhead.

1.2.2 On-Chip Hardware

Figure 1.2 shows the on-chip hardware implied in the preceding discussion. Circuitry is needed to collect the signature information and use it to create and store a signature that can be used by the off-chip algorithm to prune the set of predecessor states. In addition, a programmable breakpoint circuit that can cause the chip to halt at any arbitrary breakpoint is required. In Chapter 4 of this thesis, we consider the area overhead of this on-chip circuitry in detail.

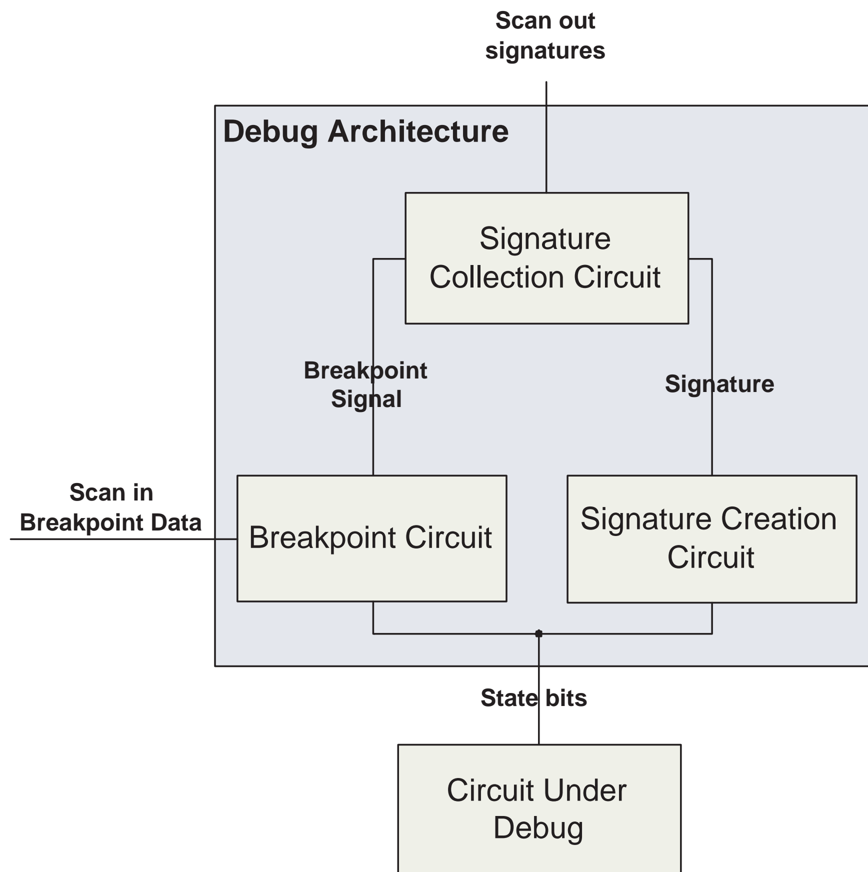

Figure 1.2: Debugging Architecture

1.2.3 Practical Considerations

This thesis focuses on practical considerations related to the BackSpace flow. There are a number of practical considerations not considered in the original proposal that will be addressed. These include:

1. The area overhead required for the on-chip circuitry is critical. We will show that the area required by the full breakpoint circuit is unreasonable. In the next chapter, we examine ways of reducing the size of the breakpoint circuitry. The breakpoint strategy is intimately related to the issue of non-determinism in the execution path of the chip, so non-determinism will be an important part of the discussion.

2. Even with the modified breakpoint strategies, the area overhead is still significant. In Chapter 4, we will model this area in detail, and show how different architectural parameters affect the overall area overhead. In doing so, we will also present several signature collection circuits.

3. Finally, to make our proposal concrete, we have developed a proof-of-concept implementation of our flow. The proof-of-concept will be described in Chapter 5.

Chapter 2

Practical Debug Flow

The BackSpace-based debug flow presented Section 1.2 is impractical due mainly to the area overhead associated with the on-chip architecture. As we will see in Chapter 3, one of the largest area contributors in the debug architecture is the breakpoint circuit, since it must store a copy of the target value of every breakpoint bit. This essentially doubles the number of flip flops on the chip. This chapter explains the changes in architecture, specifically with the breakpoint circuit, required to reduce this area overhead. The changes in methodology that must be made to support the new architecture are also described. Additionally, this methodology must cope with complications related to non-determinism in the execution of the chip.

The chapter is organized as follows. Section 2.1 provides the details of a simple breakpoint architecture that is suitable for a deterministic system. Section 2.2 explains the problems that arise as a result of non-determinism in a system. Additionally, a basic architecture and debug flow is presented that copes with non-determinism . Section 2.3 introduces the notion of a partial breakpoint, where there are fewer breakpoint bits than state bits. Sections 2.4 to 2.6 discuss how temporal and spatial false matches can be avoided when using partial breakpoints. Section 2.7 discusses the implications of a breakpoint signal being delayed, meaning that it takes more than one cycle for it to propagate across the chip. Finally, Section 2.8 discusses potential methods for choosing the partial breakpoint bits.

2.1 Deterministic Architecture

A deterministic is a system which, if given the same input pattern from run to run, runs in exactly the same way on each run. In such a system where the the input pattern is the same from run to run, a state can be fully represented by the number of cycles that have elapsed since the system was reset. This simplifies the breakpoint circuit significantly. To halt the chip at a known state, a cycle counter, which counts the number of cycles that have elapsed since reset, is used. Each cycle, the value in this counter is compared to the target state register, which holds the number of cycles that should occur before a breakpoint is produced. The overall debug flow when using a cycle counter as the breakpoint criteria in a deterministic system is shown in Figure 2.1.

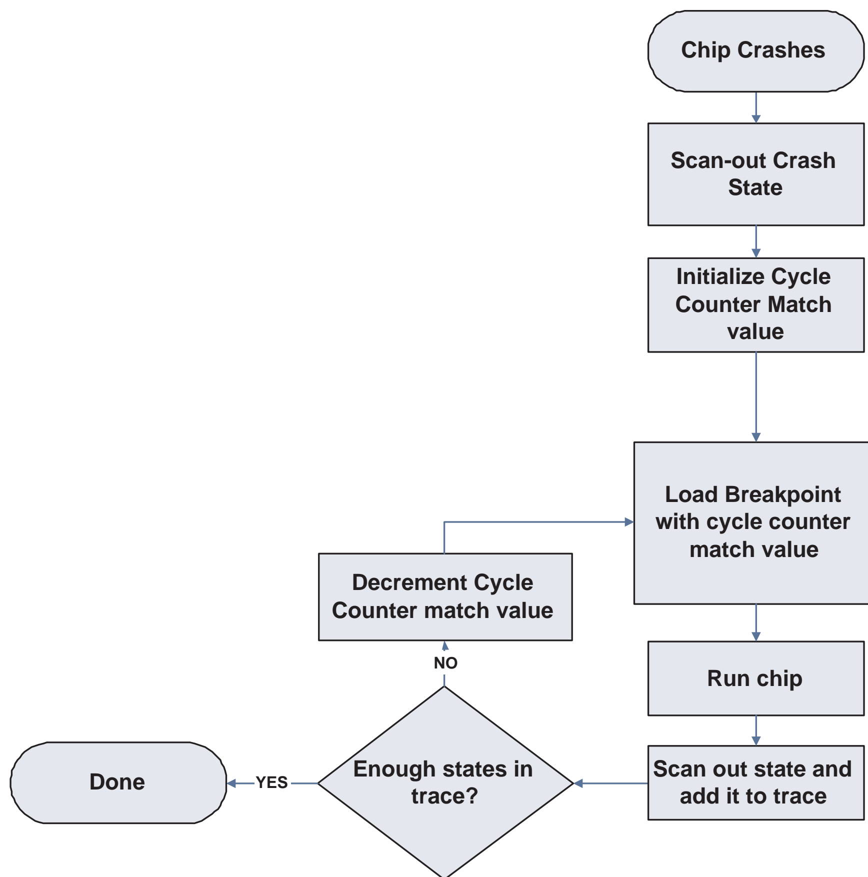

Figure 2.1: Debug Flow using a Cycle Counter as the Breakpoint Bits

In practice, to avoid aliasing of cycle counter values, a 64-bit counter is sufficient. The minimized set of breakpoint bits is simply the 64 bits of this on-chip cycle counter. This architecture is shown in Figure 2.2.

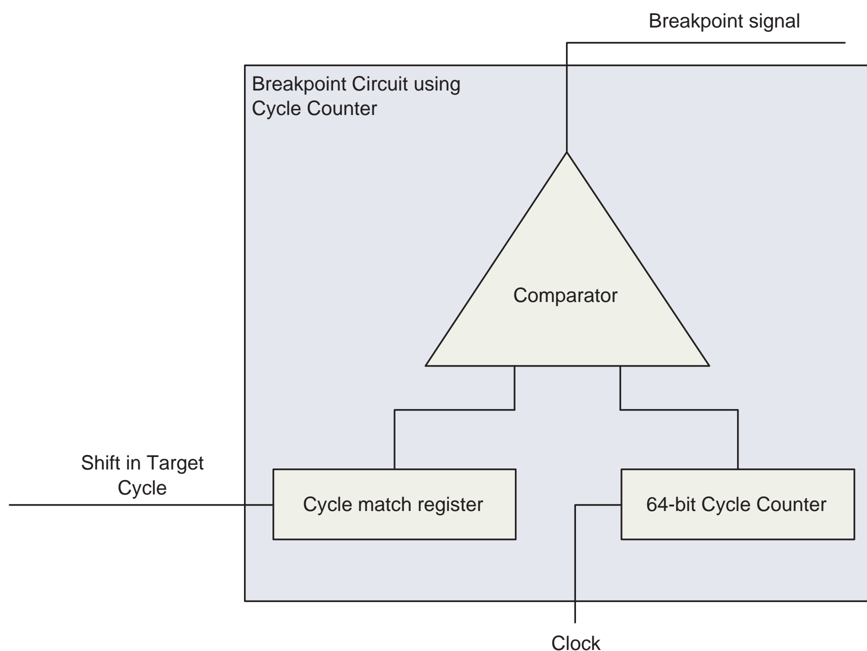

Figure 2.2: Breakpoint Architecture using a 64-Bit Cycle Counter

A downside of this method is that by augmenting the chip's state with a cycle counter, a potential differentiation is made between what were previously identical states. This makes the breakpoint criteria more strict that it should be. For example, if the system being debugged is an Ethernet block, which spends a large portion of its time waiting to send or receive packets, the system could run many cycles in the same idle state. Adding a cycle counter to the system would differentiate each of these idle states. If the Ethernet module enters a buggy state whenever it receives a packet, for example, a breakpoint should be generated at this state. If the breakpoint criteria is the value of a cycle counter, a correct breakpoint will only be generated if an Ethernet packet is received at exactly this cycle. This situation is extremely unlikely, and not what a debugging engineering would want from a debugging solution.

2.2 Non-deterministic Architecture

2.2.1 About Non-Determinism

In a practical solution, the internal signals can be different from run to run, even given the same input pattern on each run. We refer to such a system as *non-deterministic*. One possible source of non-determinism is communication between two or more separate clock domains [21]. A signal may cross the boundary from one clock domain to the next at different times from run to run depending on when the clocks were locked, relative to each other. Another possible source of non-determinism is DRAM, where the latency is variable. Even if the memory accesses occur in the same order, the timing of DRAM refreshes is non-deterministic. There are proposals to force a system to be deterministic [42] [35], but they do not cover all types of systems and require considerable changes to system hardware as well as significant design effort. Our approach is to cope with non-determinism rather than to try and control it.

Non-determinism in a system complicates breakpoint generation in two ways. First, the number of cycles before a desired breakpoint is reached may differ from run to run. Thus, the counter method described in Section 2.1 will not be sufficient. Second, non-determinism may cause the execution to take different paths from run to run, meaning the desired breakpoint may not ever be reached in any given run. In the following, we refer to a run that leads to the desired breakpoint state as a *correct run*, and a run that does not pass through the desired breakpoint state as a *spurious run*.

2.2.2 Full Breakpoint

The most straightforward way of breakpointing in a non-deterministic system is to monitor all state bits in the system, and halt the system when all state bits match the desired breakpoint bits. The architecture required for this solution is described below. As described above, it is possible that in a given run, the desired breakpoint state may not be reached. In

this case, we employ a time-out mechanism to recognize that this has occurred, and simply re-run the chip until the breakpoint is generated. Experience with the prototype described in Chapter 4 suggests that only a small number of spurious runs typically occur before a correct run that passes through the breakpoint state occurs, though this number will vary from system to system. The debug flow for this type of breakpoint architecture is repeated from Chapter 1 in Figure 2.3 below.

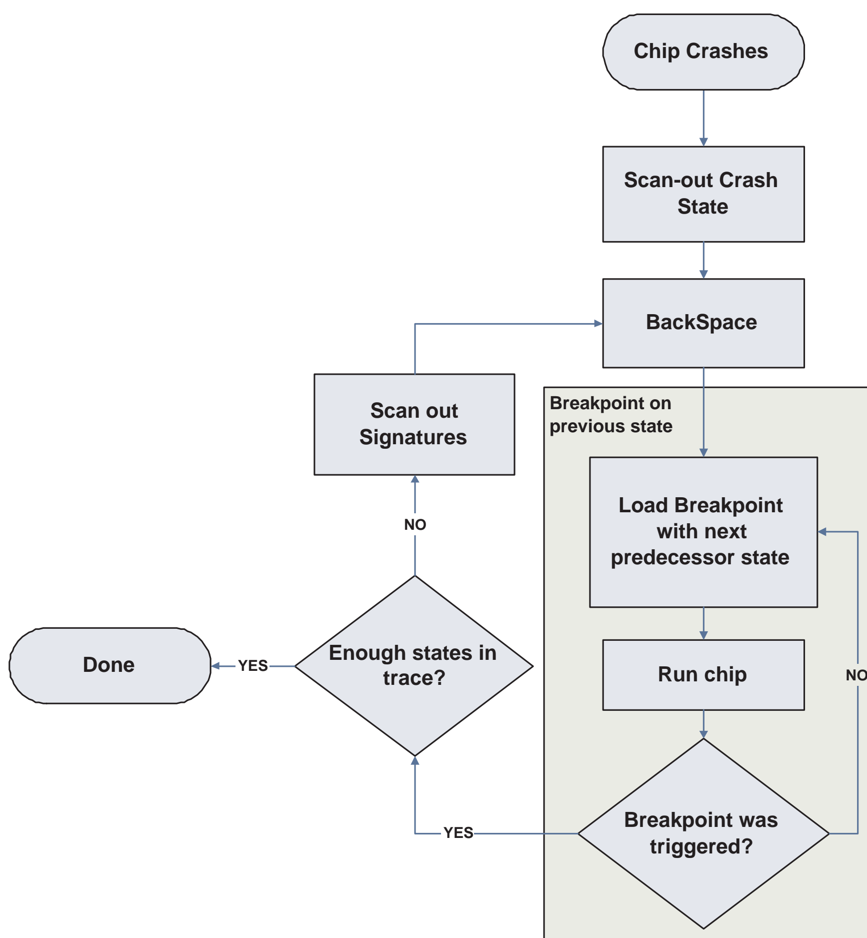

Figure 2.3: Debug Flow using Full Breakpoint

2.2.2.1 Architecture

The full breakpoint circuit is essentially a comparator. It has a target state register, which holds the values of the flip flops that make up the target state. It compares the values

in the target state register with the values of the corresponding flip-flops in the current state and provides a breakpoint signal if they are equal, as shown in Figure 2.4. For a series of n state flip flops denoted by X_0 through X_{n-1} and a series of target state register bits denoted by Y_0 through Y_{n-1}, the breakpoint function can be represented as $\overline{(X_0 \oplus Y_0) + (X_1 \oplus Y_1) + ... + (X_{n-1} \oplus Y_{n-1})}$.

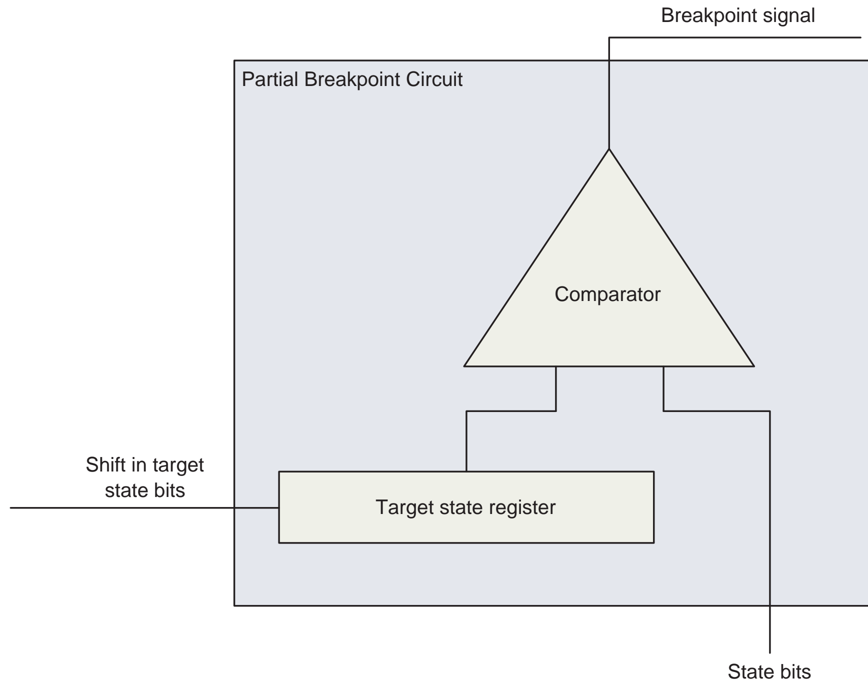

Figure 2.4: Breakpoint Circuit

2.3 Partial Matches

As detailed in Chapter 3, the disadvantage to using a full breakpoint circuit is that the area overhead required to match all state bits is large. The area overhead incurred by the target state register alone doubles the number of flip flops on the chip. It is possible, however, to match on only a subset of the state bits, though this leads to a new problem. Since not all bits are monitored, a match can be generated despite some of the bits that are not monitored not matching. This condition will be referred to as a *false match*. It is important to recognize

false matches and not include them in the overall trace being generated, because doing so would mean that a debugging engineer would waste a great deal of time looking at a trace that does not lead to the crash state.

2.3.1 Spatial and Temporal Mismatches

These false matches can be classified into two types: *temporal* and *spatial.* This is illustrated in Figure 2.5. A temporal false match is a match that occurs earlier than the target state. Since not all bits are monitored, it is possible that a match is generated several times before the correct target state is reached. A spatial false match is a match that occurs on a spurious run. This will happen if a state along the spurious run differs from the target state by only the bits that are not part of the monitored set. Deterministic systems will only experience temporal false matches, while non-deterministic systems may experience both types of false matches.

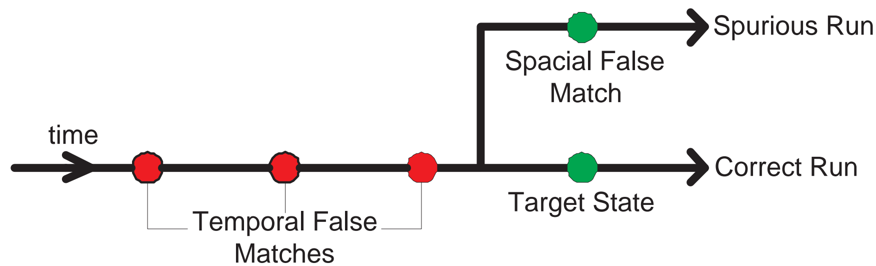

Figure 2.5: Spatial and Temporal False Matches

2.4 Coping with Non-Determinism by Scanning Out

One way of ensuring that a trace includes no false matches is to ensure that each state in the trace is on the correct path. This can be done for each run by comparing the *breakpoint state* (state that caused the breakpoint) with the target state on which a breakpoint was supposed to have been produced. If the on-chip breakpoint state matches the off-chip target state, then

the breakpoint was generated on the correct trace and the BackSpace algorithm can use the information in the signature collection circuit to generate the state's pre-image. If the states do not match, the chip was stopped as a result of a false match, which can be discarded, and the chip can be re-run. The benefit of this is that it is tolerable for the breakpoint circuit to generate false breakpoints on a spurious run, which eases the requirements on the partial breakpoint circuit.

This ability to compare the on-chip breakpoint state with the off-chip target state is made possible by scanning out the state that triggered a breakpoint. If all flip flops are part of a scan chain, the test mode can be enabled to freeze the state that is defined by those flip flops. If a breakpoint signal can, in one cycle, be generated and propagated to all flip flops on a chip, the state which caused the breakpoint can be frozen and saved for later scan-out. This is done by connecting the breakpoint signal to the test mode enable input of every flip flop in a scan chain. The assertion of the breakpoint signal stops the flip flops from changing based on their input values, thereby freezing the current state.

Scanning out the breakpoint state offers the ability to avoid including false matches in the trace. Unfortunately, by stopping the chip to scan out when a temporal false match occurs, the chip never reaches the actual breakpoint. Even if the scan-out is non-destructive, the time taken to scan out and compare the state will change the timing of the system such that the bug may not be reproducible. For this reason, it is necessary to be able to avoid temporal false matches in hardware. The cycle counter method proposed in Section 2.1 would eliminate temporal matches, but as previously mentioned, it is undesirable because it makes the breakpoint criteria overly strict. A solution is required that avoids temporal false matches without overly constraining the breakpoint criteria.

2.5 One Counter Method with Scan Out

Temporal false matches can be recognized using a programmable breakpoint match register and counter. The bits of this counter supplement the partial breakpoint bits, so that a breakpoint occurs only if the partial breakpoint matches *and* the counter matches the value pre-loaded into the match register. Every time the partial breakpoint matches but the counter does not match the pre-loaded value, the counter is incremented by one. In this way, the chip will halt at the n_{th} partial breakpoint match, where n is the value loaded into the match register. The corresponding architecture is shown in Figure 2.6 (note that the flip-flop producing the delayed breakpoint signal is not used in this section, but will be used in the next section).

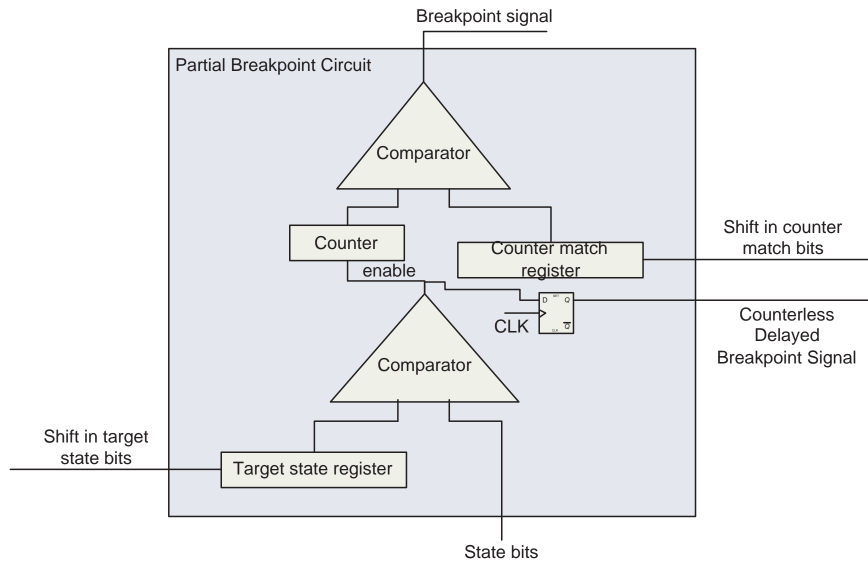

Figure 2.6: Partial Breakpoint Circuit

The challenge with this approach is to know what value to load into the match register. One approach is as follows. The match register is first set to one, and the chip is run. The chip will halt the first time the partial breakpoint matches. When this occurs, the full state is scanned out and compared to the target state. If the states do not match, i.e. if this is a false match), the counter is incremented by one, and the chip is re-run (this time, stopping

on the second partial match). This is repeated until the correct breakpoint state is reached. This flow is summarized in Figure 2.7.

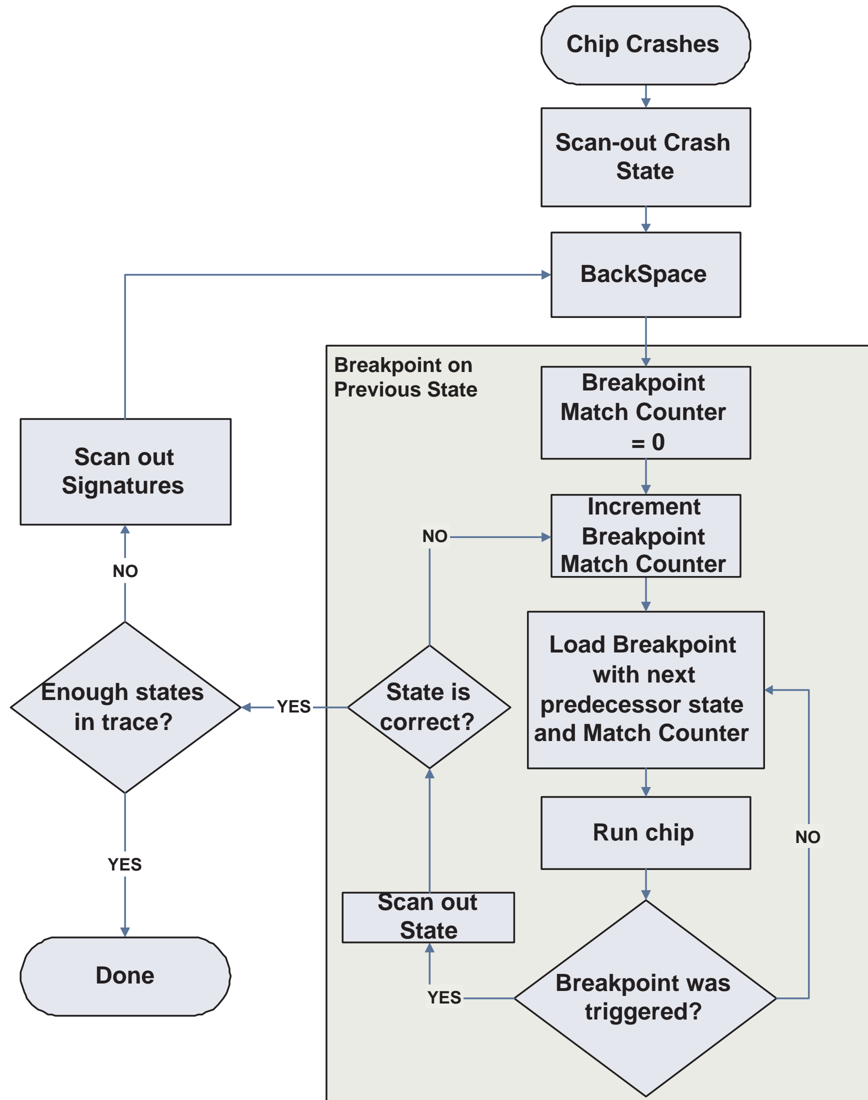

Figure 2.7: Debug Flow using One Partial Breakpoint Counter

There are two main draw-backs to this method. The first is that the extra debugging time incurred by having to run the chip several times for every breakpoint can be problematic. The second arises from the inability to differentiate a temporal false match from a spatial false match off-chip. The breakpoint match counter is incremented every time a false match occurs, whether it is temporal or spatial. In the case of a spatial false match, this results in

the correct match being skipped, as illustrated in figure 2.8. On the first and second runs, when the breakpoint counter is set to values of 1 and 2, respectively, temporal false matches occur. On the third run with the breakpoint counter set to 3, a spatial false match occurs and the breakpoint counter match value is incremented to 4 for the next run. On the fourth run, the actual target state is skipped because the counter match value is too high.

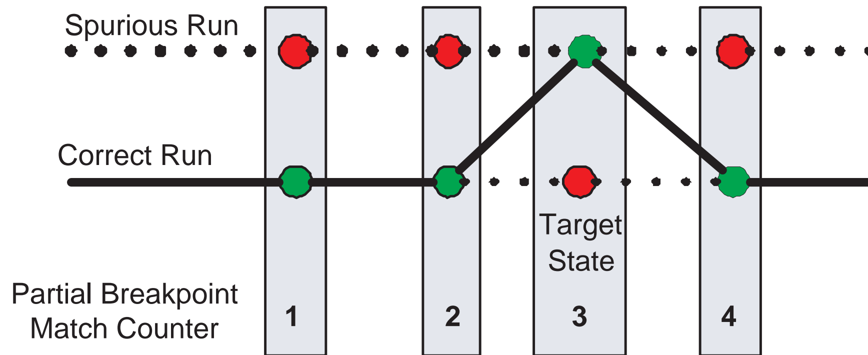

Figure 2.8: The Inability to Differentiate Temporal and Spatial False Matches can Result in Skipping the Correct Match

2.6 Two Counter Method with Scan Out

A modification to the previous breakpoint solution was proposed by Johnny Kuan which avoids the need to scan out the state to check for temporal false matches[18]. The details of the architecture required to implement this concept, however, are contributions of this thesis. A second identical partial breakpoint circuit is used in addition to the first. At any given time, one of the partial breakpoint circuits creates the breakpoint signal based on its programmed target state and its match counter. The other partial breakpoint circuit counts the number of false matches that occur before the breakpoint signal is generated. There are three modes of operation:

1. Breakpoint Circuit A generates the breakpoint signal. Breakpoint circuit B counts the number of partial breakpoint hits until it receives the breakpoint signal.

2. Breakpoint circuit B generates the breakpoint signal. Breakpoint circuit A counts the number of partial breakpoint hits until it receives the breakpoint signal.

3. Both Breakpoint A and Breakpoint B count the number of partial breakpoint hits until they receive an external breakpoint signal.

The two partial breakpoint architecture is shown in Figure 2.13. The value in the two-bit register determines in which of the above three modes the circuit is operating. Breakpoint signal A is ANDed with a one-cycle delayed version of Breakpoint signal B to ensure that Breakpoint B occurs exactly one cycle before Breakpoint A. Breakpoint signal B is ANDed with a one-cycle delayed version of Breakpoint signal A for the same reason.

The operation of the breakpoint circuit is best shown with an example. A flowchart for this example, which details the overall debug flow using two partial breakpoints is provided in Figure 2.9. Flowcharts providing details for each mode are provided in Figures 2.10, 2.11, 2.12.

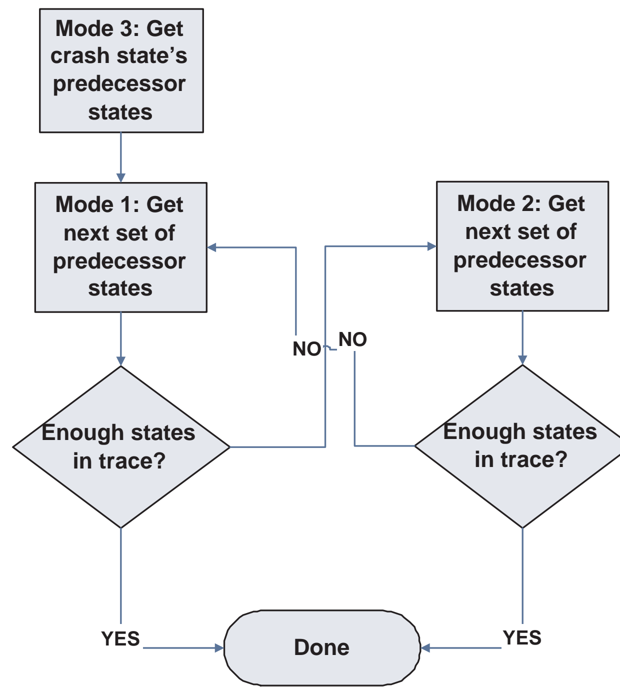

Figure 2.9: Overall Debug Flow using Two Partial Breakpoints

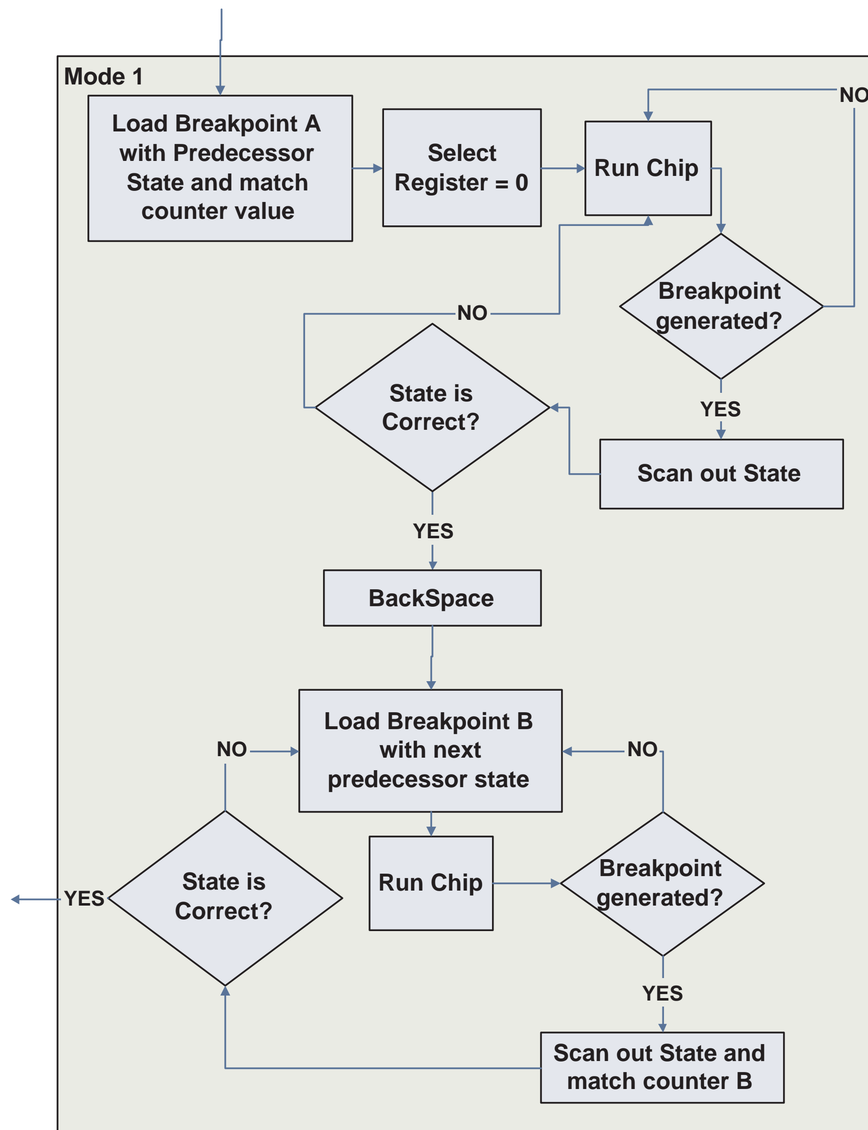

Figure 2.10: Flowchart Detailing the Operation of Mode 1

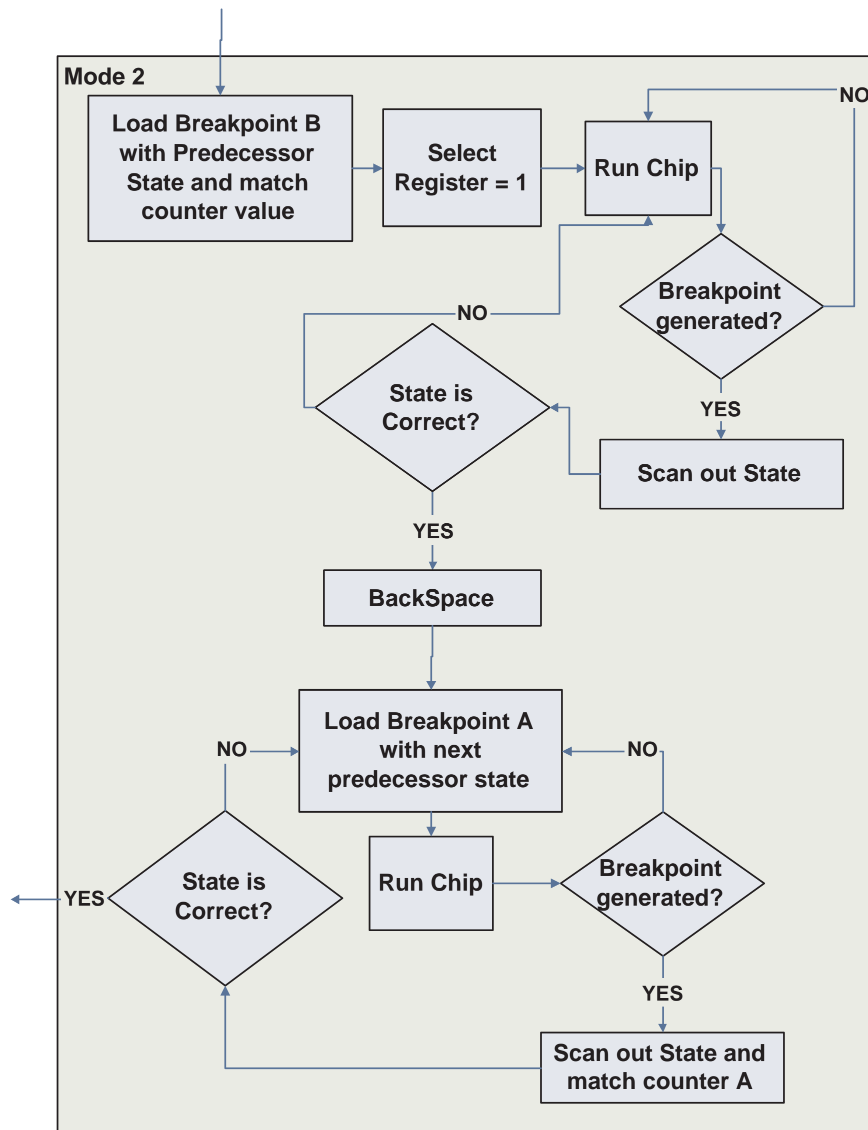

Figure 2.11: Flowchart Detailing the Operation of Mode 2

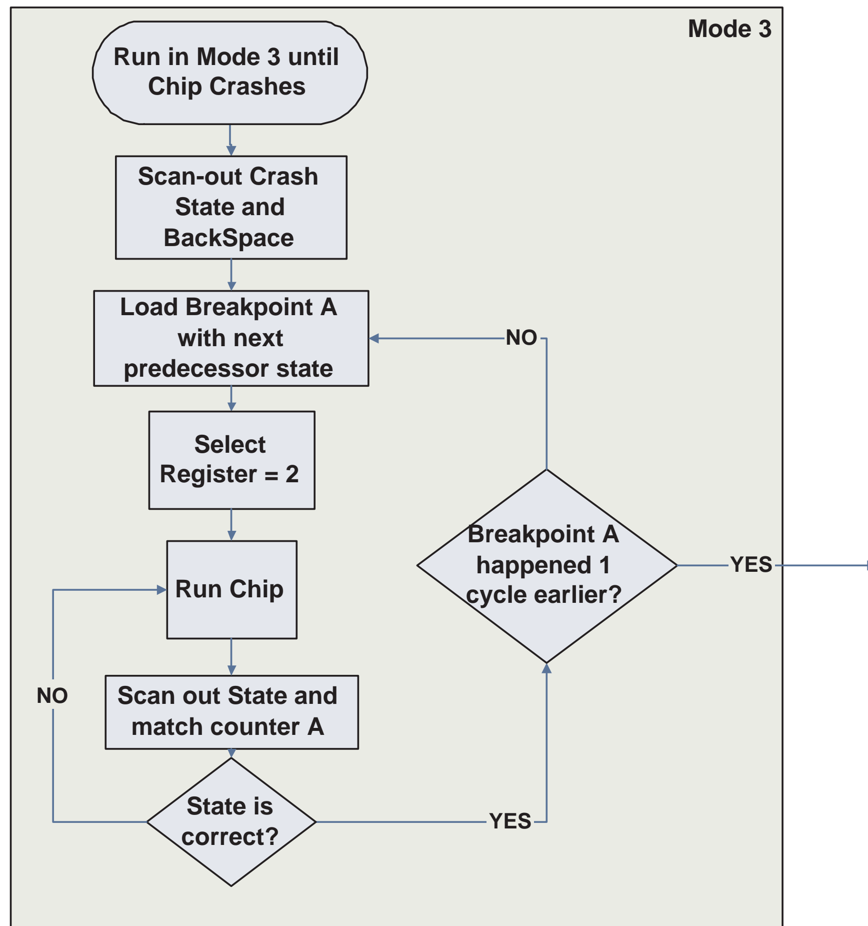

Figure 2.12: Flowchart Detailing the Operation of Mode 3

1. The breakpoint circuit starts in Mode 3 so that neither Breakpoint circuit A or Breakpoint circuit B are generating the breakpoint signal. When the buggy state is first observed off-chip, the full state and the history of signatures are scanned out. The set of candidate predecessor states is then computed using BackSpace.

2. With the breakpoint circuit still in Mode 3, Breakpoint circuit A is programmed with one of the partial previous states in the pre-image. At this point, it is unknown how many partial breakpoint hits would occur before the real breakpoint was reached so the values to put in the target match counter register is not known. During the second run of the chip, breakpoint circuit A counts the number of breakpoint hits that occur

before the buggy state is observed. This value is stored off-chip. The state of the chip is then scanned out to ensure that it corresponds with the crash state. The flip flop holding the value of breakpoint signal A delayed by one cycle is scanned out to check whether the candidate occurred one cycle before the crash state. If it did, the next step is started. Otherwise, the process is repeated with the next state in the pre-image.

3. The first partial breakpoint state and counter values are then programmed into breakpoint circuit A. The chip is put into Mode 1 and the chip is be re-run. In Mode 1, breakpoint circuit A is used along with the associated match counter value to generate the breakpoint signal. The breakpoint state is scanned out to ensure that it is on the correct path.

4. The signatures of the previous state can be scanned out and the candidate predecessor states can be computed using BackSpace. Again, the partial breakpoint required to breakpoint on this previous state is known, but the number of times that it must be hit to reach the real breakpoint is not known. The chip must be re-run to get this number.

5. One of the partial breakpoint states is then programmed into breakpoint circuit B. The chip is re-run, counting the number of false matches of breakpoint B's target state register. If breakpoint A did not generated a breakpoint signal, the chip is re-run with the next candidate predecessor state. Once breakpoint A generates a breakpoint signal, breakpoint B's target match counter register can be scanned out and stored off chip.

6. With this information, the chip can be put into Mode 2, making breakpoint B the active one. The chip can then again be re-run, this time using breakpoint circuit B along with the associated match counter value to generate the breakpoint signal. This process can continue, alternating back and forth between breakpoint A and B.

There are two main benefits to this method. The first is that only two runs are required for any breakpoint. The second is that temporal false matches are detected on chip so that the breakpoint state must only be scanned out to detect spatial false matches. The downside is that the circuit requires twice as much area as a single partial breakpoint, assuming that the number of breakpoint bits is the same in both cases.

Figure 2.13: Two Partial Breakpoints Circuit

2.7 Delayed Breakpoints

The two breakpoint solution provided in the previous section requires the breakpoint signal to be generated and distributed to all flip flops in one cycle. Unfortunately, this is very difficult to do without slowing down the chip. The generation of the breakpoint signal requires the comparison of many flip flop outputs, which are scattered across the chip. These outputs must first be routed to a central location. Meeting timing requirements for this step alone is not trivial. The routed signals must then be compared against the target state register outputs. In a realistic implementation, the comparator could be distributed

to reduce the amount of routing required, but the signals would still need to eventually be routed to a central location. The propagation of the breakpoint signal is also difficult to do in one cycle. Solving this step is similar to designing a clock distribution network, which requires significant design effort. Slowing down the clock can make one-cycle breakpoint signal generation and propagation possible, but it is undesirable because it can alter the behaviour of the chip. The behaviour of the chip during normal operation and during debugging should be identical to facilitate the reproducibility of bugs.

2.7.1 Pipelining the Breakpoint Circuit

The circuitry that generates and propagates the breakpoint signal can be pipelined to ease timing requirements. Figure 2.14 shows the comparator portion of the breakpoint circuitry with one pipeline stage so that the breakpoint signal is generated in two cycles instead of one. The propagation of the breakpoint signal to the flip flops on the chip can be pipelined in the same way.

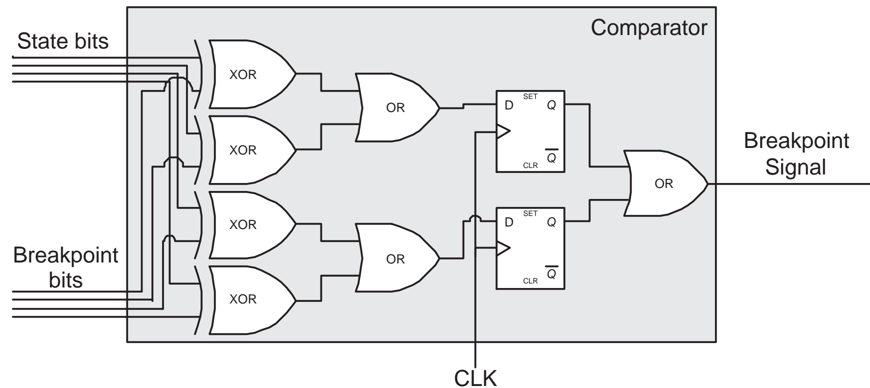

Figure 2.14: The Comparator Portion of a Breakpoint Circuit with One Pipelining Stage

The result of pipelining the breakpoint circuit is that the breakpoint signal will stop the flip flops some number of cycles after the breakpoint is triggered. This means that it is not possible to scan out the state that triggered the breakpoint. The solution to this problem is addressed in the next section.

2.7.2 Debug Flow with Delayed Scan Out

Pipelining the breakpoint circuit as described in Section 2.7.1 causes a problem with the Two Counter flow in Section 2.6. The Two Counter flow requires that, when the chip stops because of a partial breakpoint match, the state is scanned out and compared to the target state. If the breakpoint circuit is pipelined, then by the time the circuit stops, the state has progressed several cycles beyond the matching state. In this subsection, we describe a modification to the flow that works even if the breakpoint signal is delayed by several cycles. We first describe the basic flow in this subsection, and then two refinements in the next two subsections that make the flow practical.

Before explaining the flow, we first present several definitions. In the following, we make the distinction between the states that the chip goes through (we will refer to these as *execution states*) and the states that are stored in the trace history during the previous iterations of BackSpace (which we will refer to as *stored states*). In the Two Counter flow, each time a run successfully matches the state in the breakpoint register, the state is added to the set of stored states (the set of stored states is the eventual output to the BackSpace flow). In this section, we differentiate between the following two execution states:

Breakpoint State This is the state of the chip that actually triggers the breakpoint signal.

Frozen State This is the state of the chip that is frozen as a result of an active breakpoint signal.

In the Two Counter flow presented in Section 2.6, the Frozen State is the same as the Breakpoint State, but if the breakpoint circuit is pipelined, the Frozen State occurs $Delay_{pipeline}$ cycles after the Breakpoint State, where $Delay_{pipeline}$ is the delay of the pipelined circuit.

We also differentiate between two stored states in the trace history:

Frontier State This is the state from which the breakpoint bits are derived.

Test State This is a state that was added to the trace history $Delay_{pipeline}$ cycles before the Frontier State (ie. if the Frontier State is used as a breakpoint in iteration i of the BackSpace flow, then the Test State is the state that was used as a breakpoint in iteration $i - Delay_{pipeline}$ of the BackSpace flow).

Figure 2.15 illustrates the relationship between execution state and stored states. The circles on the left side of the figure represent the execution states, while the boxes on the right side of the figure represent the stored state. The Breakpoint, Frozen, Frontier States and Test States are labeled.

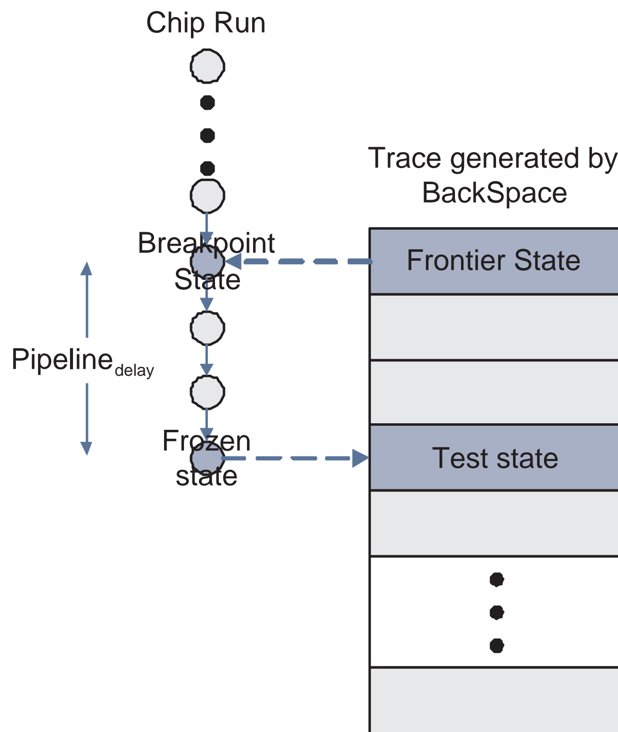

Figure 2.15: Relation between Defined States in a Trace Generated by BackSpace and a Run On-Chip

Given these definitions, the flow is as follows. As in the previous flow, the list of candidate predecessor states produced by BackSpace is considered one at a time. The candidate predecessor state being considered (the Frontier State) is loaded into the breakpoint register, and the chip is run. The Frontier State is also added to the trace history, and tagged as

unconfirmed. When a match occurs, the breakpoint signal is generated $Delay_{pipeline}$ cycles after the state that caused the match (the Breakpoint State). By this time, the execution has progressed $Delay_{pipeline}$ cycles, meaning that the state when the chip stops is no longer the Breakpoint State, but instead has progressed to the Frozen State. This Frozen State is then scanned out, and compared to the Test State. If a match occurs, the Test State in the trace history is tagged *confirmed*, and the iteration is complete. If a match does not occur, a new candidate predecessor state is selected, and the process repeated.

A key observation in the above algorithm is that if the Frozen State matches the Test State, then the Breakpoint State *likely* matches the Frontier State. There is a possibility, however, that the execution does not pass through the Breakpoint State but does pass through the Test State, *and* that a false match is generated exactly $Delay_{pipeline}$ cycles before the Test State. In Figure 2.16, Path 1 is the run that occurs on chip, Path 2 is the run that would need to occur to pass through the Frontier State, and $Delay_{pipeline} = 3$. A false match occurs so that a breakpoint is generated on Path 1 exactly $Delay_{pipeline}$ cycles before the Frozen State. In this scenario, the Frozen State matches the Test State but the Frontier State does not match the Breakpoint State. When this occurs, the Frontier state is still added to the trace history, yet it should not be there.

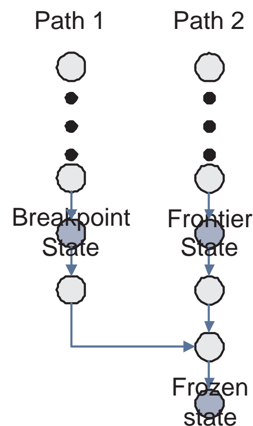

Figure 2.16: Converging Paths with Different Breakpoint State but the Same Frozen State

This situation can be detected if, when BackSpacing from a Frontier State, none of the

candidate predecessor states lead to a match after a certain number of chip runs. This would mean that the Frontier State is on an un-reachable or a difficult to reach path. If this occurs, the flow reverts to the most recently *confirmed* state, and tries again. Flowcharts for the modified flow are shown in Figures 2.17 and 2.18.

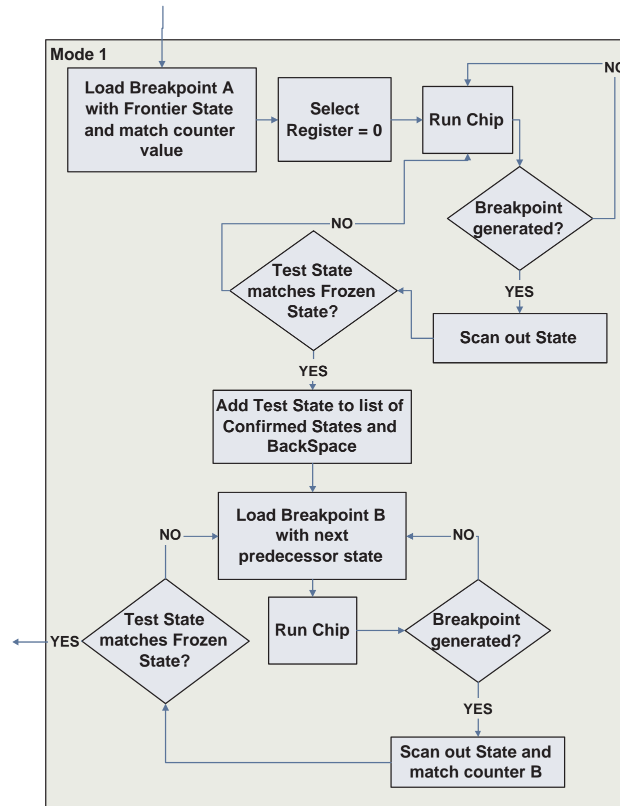

Figure 2.17: Modified Version of Mode 1 Taking Delayed Scan-Out into Account

In this thesis, we do not provide a proof-of-correctness for this flow. However, our intuition behind the correctness of this flow is as follows. This methodology works because all the states labeled *confirmed* can be relied upon to be part of a single trace that is reachable and leads to the Crash State. This property of the confirmed states can be justified by further exploring the situation in which each Test State is labeled as *confirmed*.

The first Test State that is produced is generated by computing the pre-image of the Crash State. The fact that the Test State is a predecessor to the Crash State means that it is possible to reach the Crash State from this Test State. The next Test State that

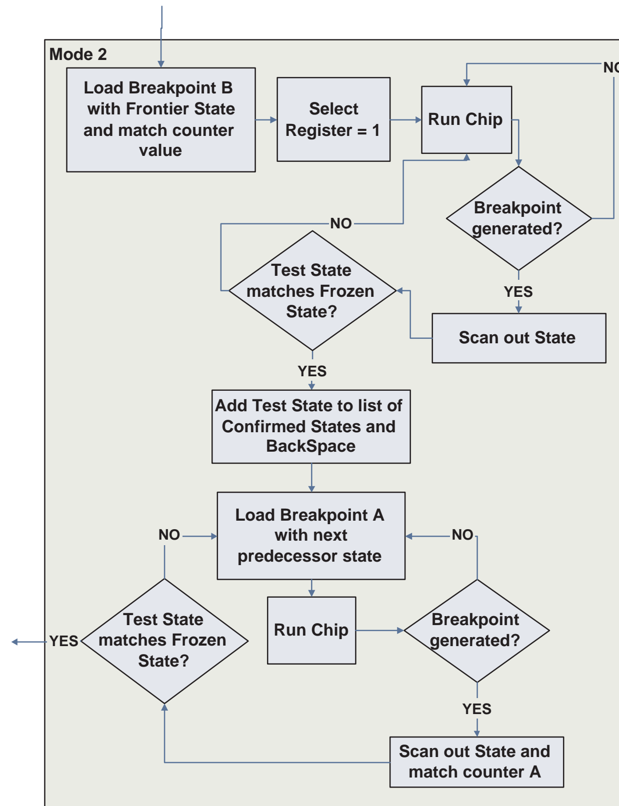

Figure 2.18: Modified version of Mode 2 taking delayed scan-out into account

is generated by BackSpace is generated by computing the pre-image of the previous Test State. This means that this more recent Test State can also reach the Crash State. All subsequent Test States are generated in the same way, and can therefore lead to the Crash State. In fact, any Test State begins a valid path that leads through the other states in the trace previously generated by BackSpace and ends at the Crash State.

The Frozen State is always a reachable state because it is scanned out from the chip itself, which shows that it was reached. If the Frozen State matches the Test State, the first part of the trace leading from Reset to the Frozen State connects with the second part of the trace leading from the Test State to the Crash State to form one continuous path from Reset to the Crash State, that passes through all confirmed states, as shown in Figure 2.19.

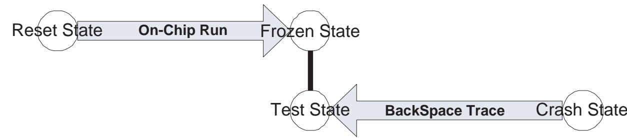

Figure 2.19: BackSpace Trace and On-Chip Run Connecting to Form a Valid Path from Reset to Crash

2.7.3 Repeating the Same Mistake

The methodology provided in the previous section provides a mechanism to recover from BackSpace going down an unreachable path, but there is nothing to prevent BackSpace from continuously making the same mistake. If the candidate predecessor states are selected in the same order and the on-chip runs also occur in the same order so that the same false matches occur, BackSpace could repeatedly go down the same unreachable path.

Ensuring that BackSpace does not test the candidate predecessor states in the same order every time will solve this problem. The simplest way of doing this is to randomly choose the order in which the predecessor states are tested on chip. This method, however, precludes further research into choosing this ordering intelligently. Instead, the order in which the candidate predecessor states are tested on-chip can be determined by a weight function. A candidate predecessor state that has led to an unreachable path is negatively weighted to discourage its use. It is not completely eliminated because it is possible that it leads to a reachable path in addition to leading to the unreachable path which was previously traversed.

The overall debug flow used to account for BackSpace going down unreachable paths is provided in Figure 2.20. Modes 1 and 2 have been altered according to Figures 2.21 and 2.22.

2.7.4 Blind Spot Problem

Comparisons between the Frozen State and the Test State can only be made if BackSpace has already generated part of the trace leading to the crash state. Unfortunately, for the first $Delay_{pipeline} - 1$ states directly before the crash state (blind spot states), the frozen

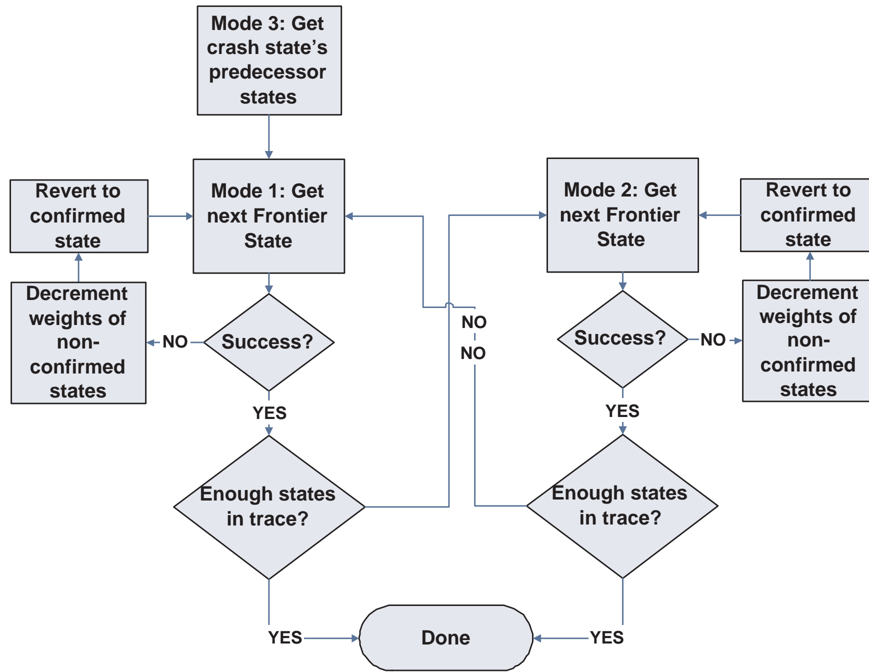

Figure 2.20: Debug Flow Used to Deal with BackSpace Going down Unreachable Paths

state occurs after the crash state, where no trace information is available. Figure 2.23 demonstrates the situation when the breakpoint signal is delayed such that the Frozen State occurs after the latest state in the trace created by BackSpace. Since no trace information is available for blind spot states, it cannot be guaranteed that a breakpoint is generated on a correct path simply by scanning out the Frozen State.

2.7.5 Blind Spot Solution

One approach to this problem is to operate blindly for the blind spot states, without checking the scanned out state off-chip to ensure that it is on the correct path. Blind versions of modes 1,2, and 3 are shown in Figures 2.24, 2.25, and 2.26, respectively. When BackSpace has added $Delay_{pipeline}$ states to the trace, it can be determined whether the breakpoint state is on the correct path by scanning out the frozen state, which now corresponds to the crash state. If it is on the wrong path, the process is repeated, re-generating a trace for the blind spot states. The number of times that this process must be repeated depends on the

effectiveness of the breakpoint circuit in differentiating a correct run from a spurious run. The overall debug process including the blind zone is shown in Figure 2.27.

Since it is not possible to check off-chip that a breakpoint state is on the correct path when in the blind zone, it is the responsibility of the breakpoint circuit to avoid false breakpoints. The choice of the breakpoint bits determines how effective the breakpoint circuit is at differentiating between a correct path and a spurious path, which determines the number of false breakpoints that are generated.

2.8 Selecting Bits

The architecture and methodology described in this chapter has provided a means to use a subset of all state bits as breakpoints bits. We have not addressed how to choose these bits but we know that the best breakpoint bits are the bits that best differentiate between any correct target state and the spatial false matches that are possible at that point in time. Intuitively, using more breakpoint bits leads to fewer spatial false matches but also has higher area overhead.

In Chapter 4, we will find the set of bits that provides the best compromise between area overhead and reduced number of spatial false matches for our prototype. We run experiments on the prototype to find this set of bits. Although this empirical method is effective for our prototype, it would be useful to have a way of automatically selecting bits that does not rely on having a cycle-accurate simulator or a prototype. This methodology would likely be based on structural information found in the gate-level HDL. As this is outside the scope of this thesis, we discuss the potential methodology further as future work in Section 5.3.

2.9 Chapter Summary

This chapter presented a more practical debugging architecture and flow which works with the BackSpace Algorithm. We focused on reducing the area of the full breakpoint circuit

because it contributed significantly to the overall area overhead of the debug architecture. The modified breakpoint architecture and debug flow that was presented allows for partial breakpoints in realistic systems, where sources of non-determinism exist and where a breakpoint signal cannot be easily propagated across the chip in one cycle. The area savings enjoyed as a result of these architecture changes are detailed in Chapter 3.

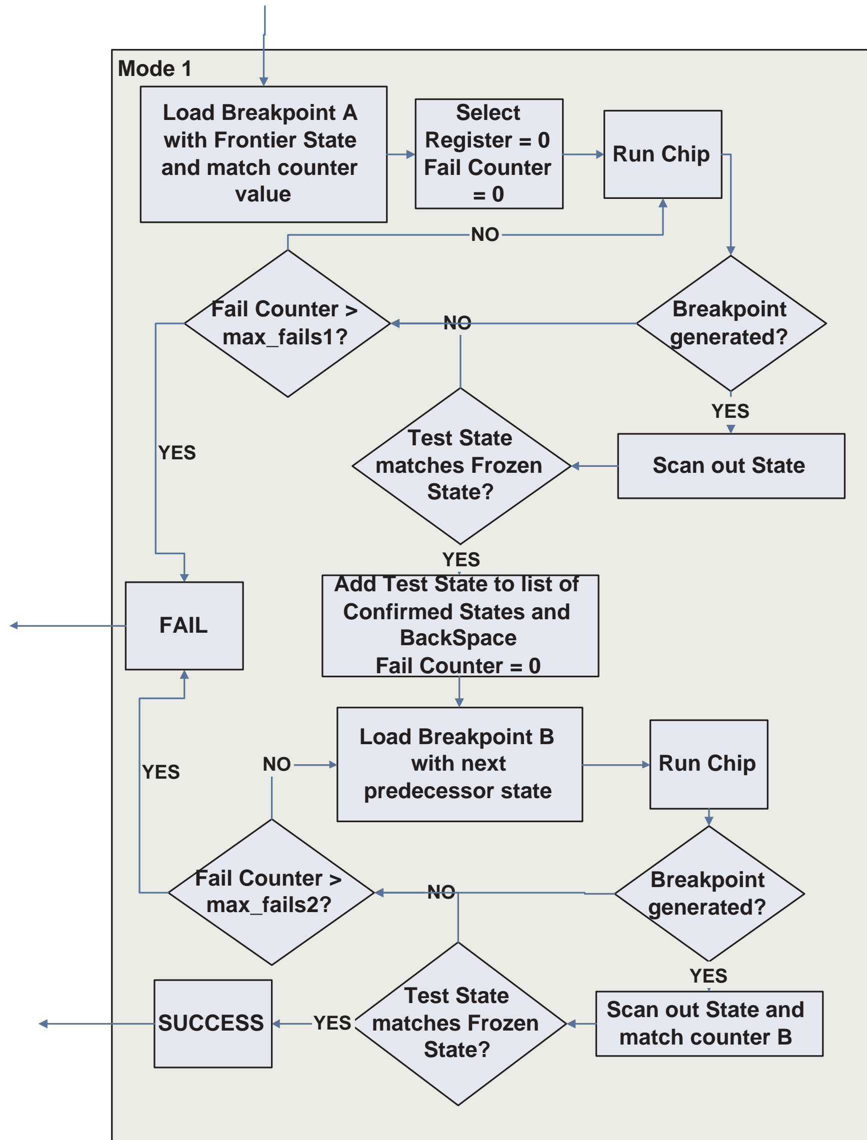

Figure 2.21: Mode 1 for Debug Flow Coping with Unreachable Paths

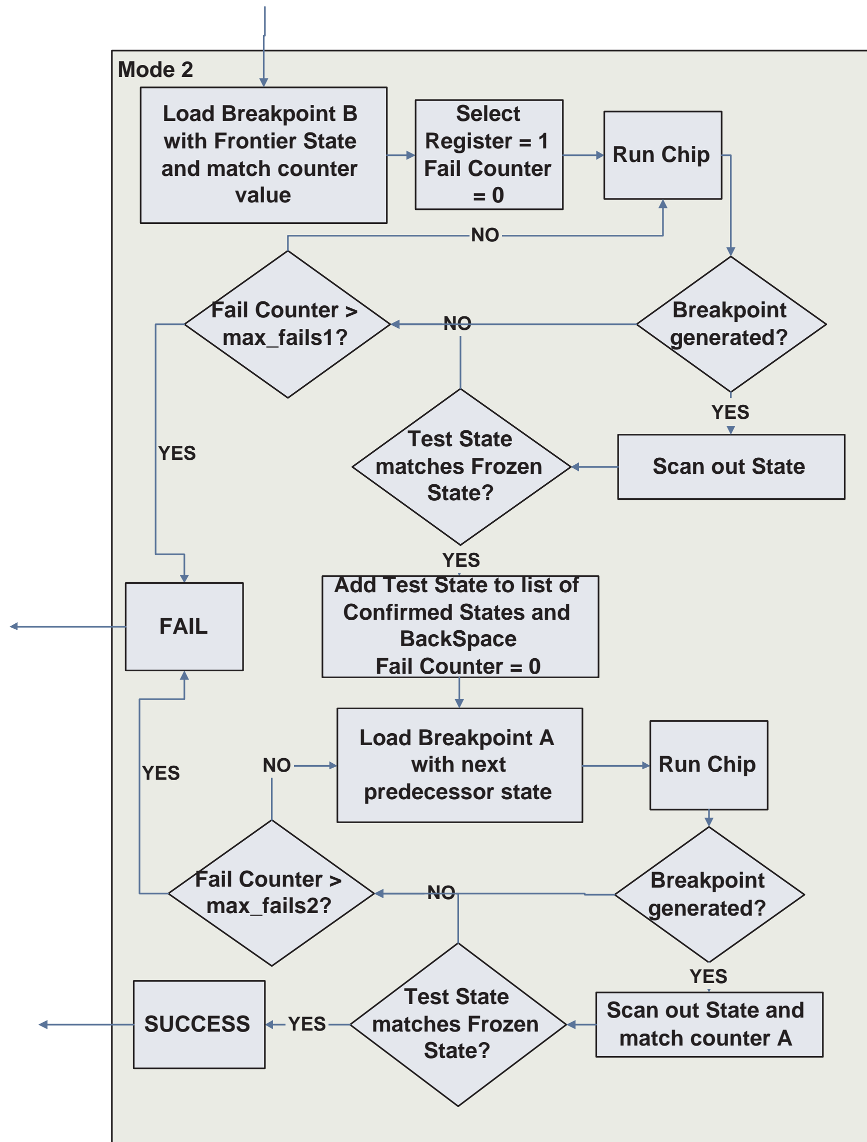

Figure 2.22: Mode 2 for Debug Flow Coping with Unreachable Paths

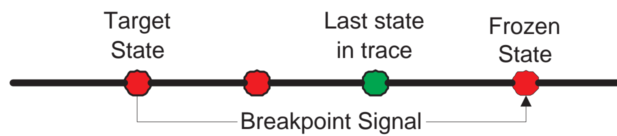

Figure 2.23: Frozen State Occurring after Latest State in Trace

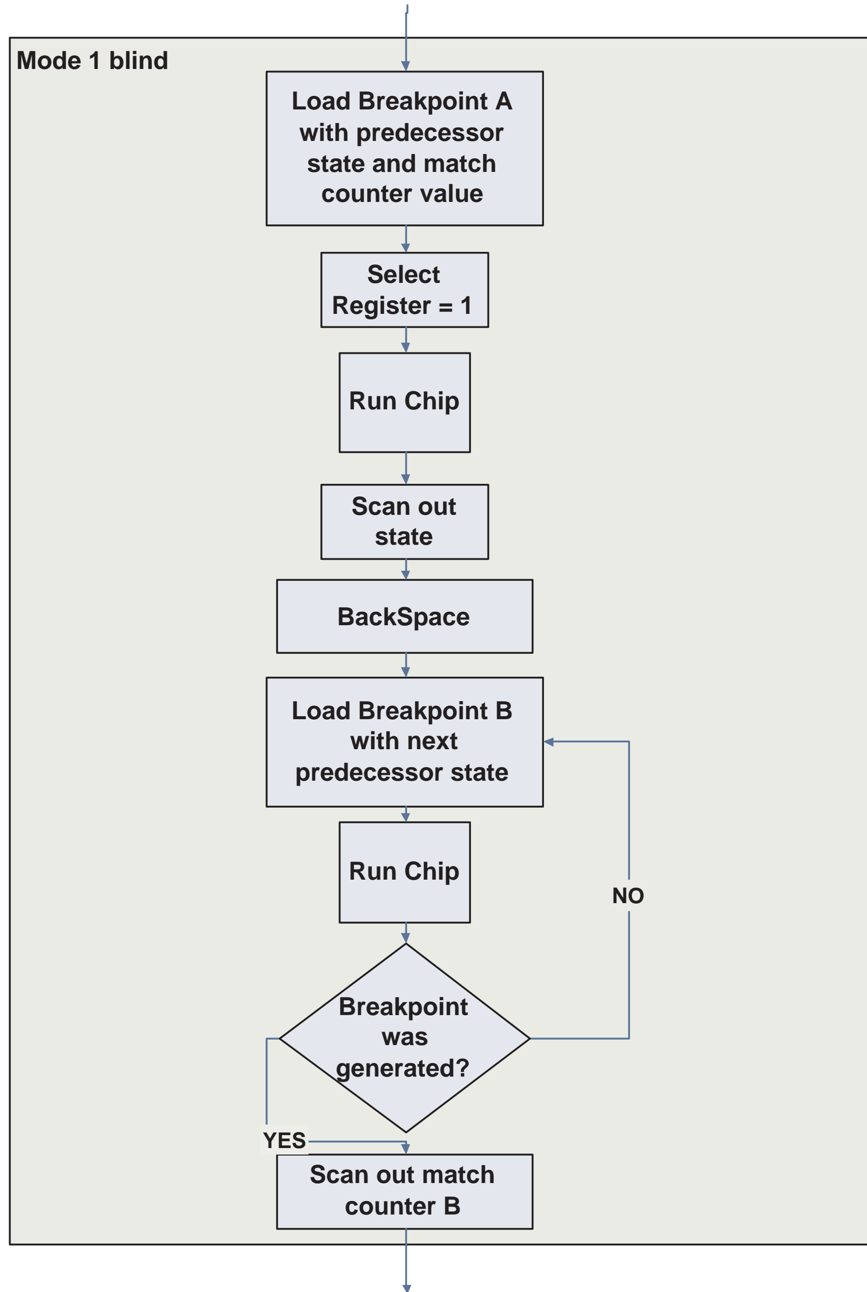

Figure 2.24: Flowchart Detailing the Blind Operation of Mode 1

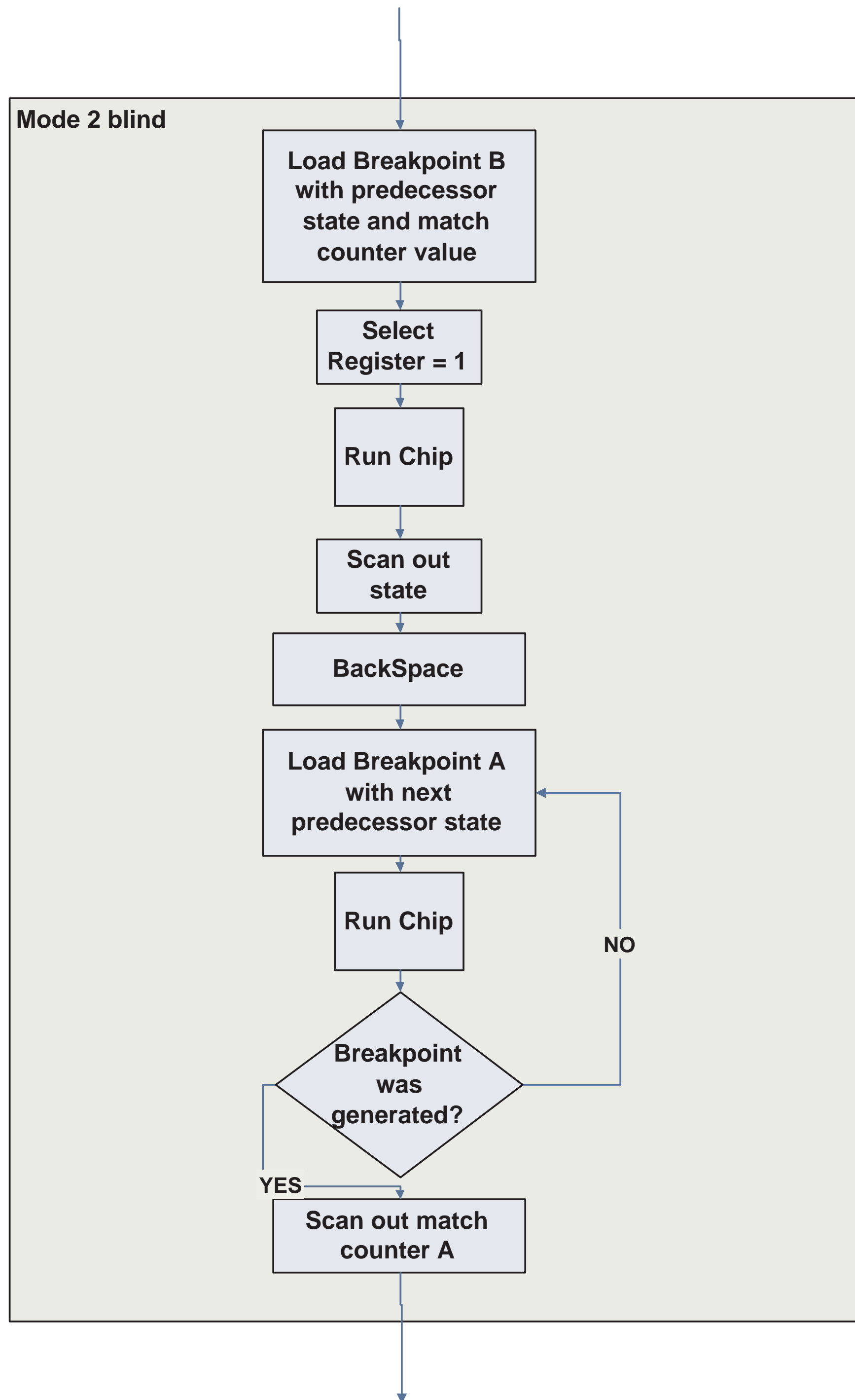

Figure 2.25: Flowchart Detailing the Blind Operation of Mode 2

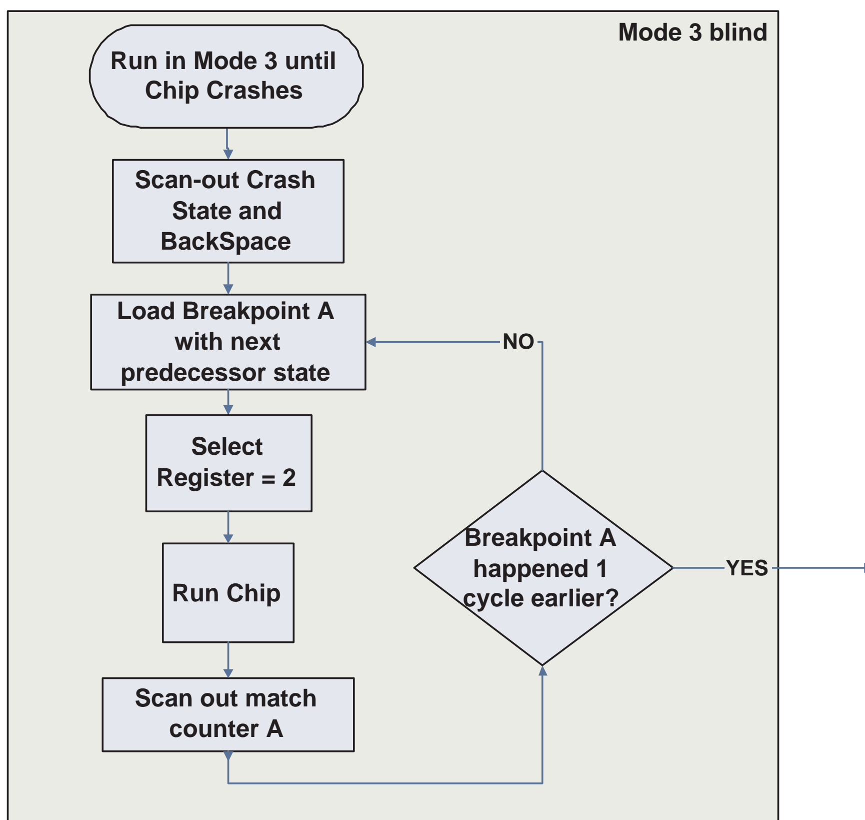

Figure 2.26: Flowchart Detailing the Blind Operation of Mode 3

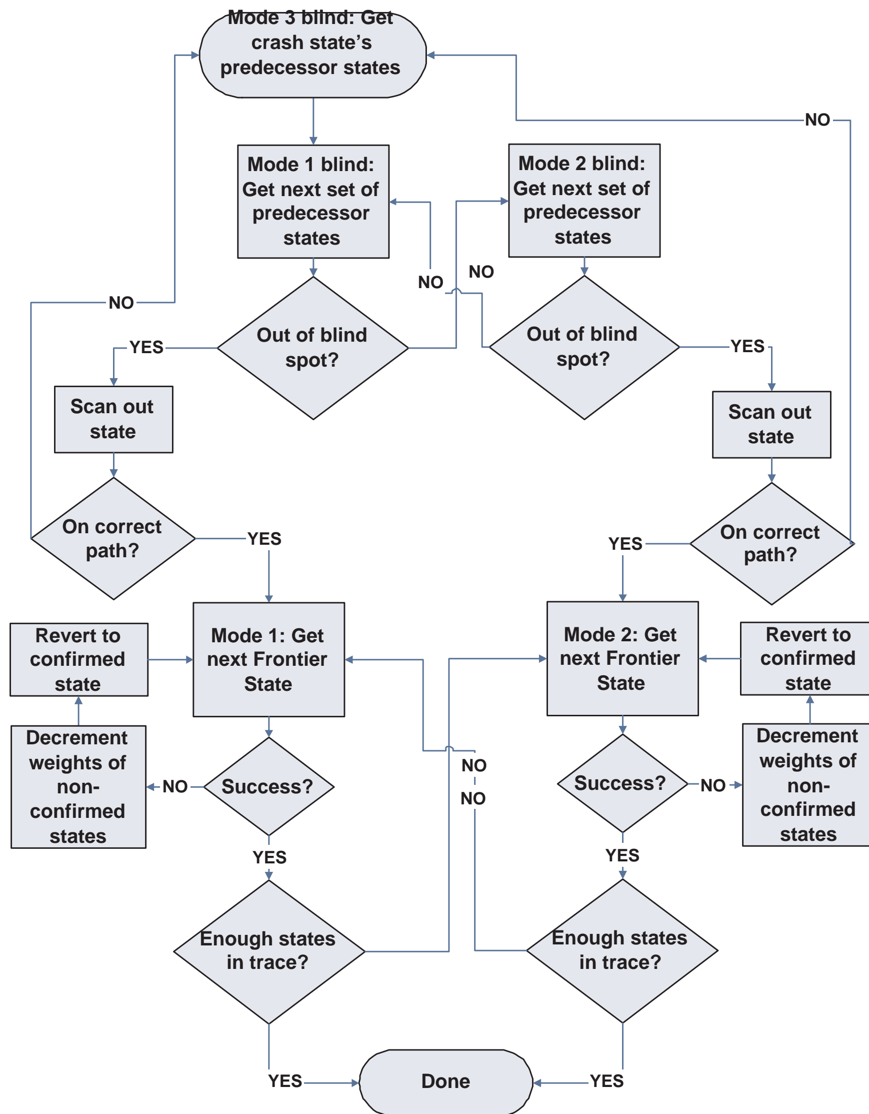

Figure 2.27: Overall Debug Flow using Two Partial Breakpoints with a Blind Zone

Chapter 3

Area Considerations

This chapter presents a parameterized area model used to estimate the area overhead of the debugging architecture. The area incurred by each variation of each component is examined. The methodology used to calibrate the area model is presented in Section 3.1. A detailed area model is provided for the breakpoint circuit in Section 3.2, for the signature creation circuit in Section 3.3, and for the signature collection circuit in Section 3.4. Various tradeoffs associated with the area of each component are examined in Section 3.5.

3.1 Area Model Calibration

The area model consists of a number of equations, which are described in Sections 3.2,3.3, and 3.4. The equations were calibrated with synthesized VHDL models. Each type of circuit, scaled to different bid-widths, was synthesized using Synopsys Design Compiler mapped to the 0.18um TSMC standard cell library. The results were used to create area per bit values, which are shown in Table 3.1, for each type of circuit. These area per bit values were used to calibrate the area model, which was used to quickly get overall area results, sweeping for wide ranges of parameters. The SRAM area was gathered using a model originally developed in [41], but re-calibrated for modern technologies (the results of the model were compared to commercial memories developed from a memory generator and the estimations were found to be accurate). Note that only standard cell area is estimated (does not include routing overhead).

bit area	area (microns)
area_per_comparator_bit	39.449
area_per_gte_comparator_bit	44.626
area_per_reg_bit	73.181
area_per_pointer_logic_bit	116.760
area_per_counter_bit	155.255
xor_area	32.525
ff_area	73.181
mux_area	24.394
and_area	16.262
membitarea	5.515

Table 3.1: Area per Bit Values in 0.18um for Architecture Components

3.2 Breakpoint Circuitry

3.2.1 Full Breakpoint

The area of the breakpoint circuit is composed of the area required for the comparator and the area required for the target state register. Comparators of various sizes were coded in Verilog and synthesized with Synopsys Design Compiler. The area was found to be proportional to the number of inputs to the comparator and so an area_per_comparator_bit value was determined. The area of the comparator can thereby be expressed as:

$$N_{mon} * area_per_comparator_bit \tag{3.1}$$

where N_{mon} is the number of signals being monitored on chip. For the full breakpoint, this is the same as the number of flip flops on the chip. The target state register area can be expressed simply as:

$$N_{mon} * area_per_reg_bit \tag{3.2}$$

where area_per_reg_bit is the same as the standard cell area of a flip flop. All together, the area of the full breakpoint circuit is:

$$N_{mon} * (area_per_reg_bit + area_per_comparator_bit) \tag{3.3}$$

3.2.2 Partial Breakpoint

The area model for the partial breakpoint is similar to the full breakpoint, though N_{mon} is scaled by $bp_{fraction}$, which represents the fraction of all bits which are used to trigger a breakpoint and ranges from 0 to 1. Additionally, a breakpoint match counter and a target match counter register must also be added so that the breakpoint signal can be activated once the counter reaches a certain value. The counter and counter match registers have a fixed width of 32 bits, which is conservative, since it is highly unlikely that the number of false matches exceeds 2^{32}. An extra flip flop is added to store the delayed value of the breakpoint signal, which is used in the two-breakpoint method as described in section 2.6. In addition to the match counter and the target match counter register, a comparator which evaluates whether the match counter value is greater than or equal to the target match counter register must be added. The area associated with the comparison between the counter and the counter match register can be expressed as:

$$32 * (area_per_count_bit + area_per_compare_gte_bit + area_per_reg_bit) \tag{3.4}$$

The comparator used here is different than the comparator used to compare the target state register and the state bits since comparisons are based on a greater than or equal condition rather than only the equal condition. For this reason, area_per_comparator_gte_bit is different than area_per_comparator_bit. The area of the comparator associated with the state bits and the bits in the breakpoint target register, including the flip flop holding the delayed value of the breakpoint signal, can expressed as:

$$N_{mon} * bp_{fraction} * (area_per_compare_bit + area_per_reg_bit) + ff_area \tag{3.5}$$

Adding these two equations together yields the equation for the overall area of the partial breakpoint circuit:

$$(N_{mon} * bp_{fraction} + 33) * area_per_reg_bit + N_{mon} * bp_{fraction} * \\ area_per_compare_bit + 32 * (area_per_count_bit + area_per_compare_gte_bit) \quad (3.6)$$

3.2.3 Two Partial Breakpoints

The area of the two partial breakpoints is essentially twice the area of the single partial breakpoint, though a 3:1 multiplexer (built from two 2:1 multiplexers) and two flip flops must be added to control which partial breakpoint will stop the chip. The area can be expressed as:

$$2 * ((N_{mon} * bp_{fraction} + 33) * area_per_reg_bit + N_{mon} * bp_{fraction} * \\ area_per_comparator_bit + 32 * (area_per_count_bit + \\ area_per_compare_gte_bit) + mux_area + and_area + ff_area) \quad (3.7)$$

3.3 Signature Creation

The signature creation circuit's role is to create a signature from a chip's state. There are several ways to build this circuit. The simplest signature creation circuit would just extract pre-determined bits, while more complex alternatives would involve flexible bit selection (flexible at run-time) or compression to extract key information about the state. In this section, we describe several alternatives for this circuit. In all cases, the goal is to provide an adequate amount of information about the chip's state to the BackSpace algorithm, while consuming the minimum chip area.

3.3.1 Hard-Wired Bits

The simplest variation of the signature creation circuit is to hard-wire a selection of flip flops to the signature collection circuit inputs. This solution provides no compression of state bits and no flexibility to change which bits make up the signature post-fabrication. This lack of flexibility means that the state bits must be chosen carefully. The benefit of hard-wiring bits is that doing so does not require any logic area.

3.3.2 Concentrator

An alternative to hard-wiring the flip flops is to use a concentrator network to access the flip-flops, thereby providing some flexibility post-fabrication, as shown in Figure 3.1. Although the area overhead is larger for this type of circuit as compared to hard-wiring flip flops to the signature collection circuit, the additionally flexibility can be used to change which flip-flops make up the state signature at run-time. This eliminates the need to determine which bits will make up the signature before fabrication. Additionally, whereas the hard-wired signature has to provide enough information to prune any pre-image to very few states, the flexible signature is adequate as long as one of many configurations can provide this information. The flexibility to change the signature potentially means that the number of bits in the signature could be far fewer than the hard-wired signature.

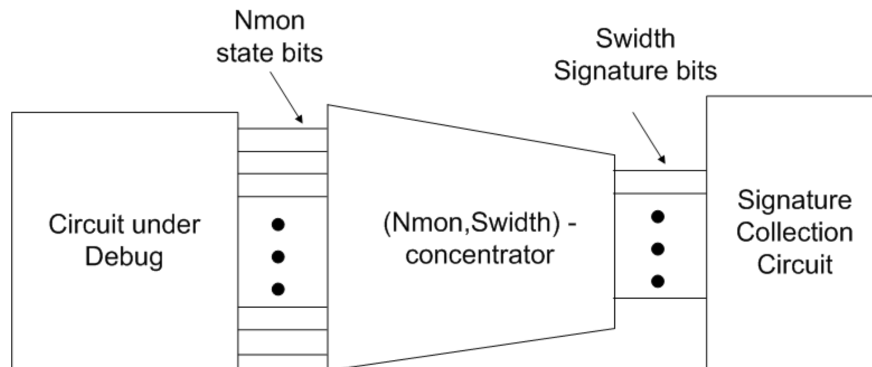

Figure 3.1: Using a Concentrator as the Signature Creation Circuit

Using the same method described in [29], a concentrator with n inputs and m outputs is built recursively from two smaller concentrators, each with $n/2$ inputs and $m/2$ outputs. The first $n-2$ inputs are connected to $(n/2)-1$ crossbars, each with two inputs and two outputs, as shown in Figure 3.2. Each of these is then connected to each of the two smaller concentrators. If the number of inputs/outputs is not divisible by two, the next larger value is used and the unneeded inputs/outputs are ignored. The final two inputs are each directly connected to one of the concentrators. Each of these two smaller concentrators can be built using the same construction. Once the concentrator reaches a size where the number of outputs is less than or equal to 2, a sparse crossbar construction can be used for the concentrator. More details on the use of such a concentrator in a similar application is described in [31].

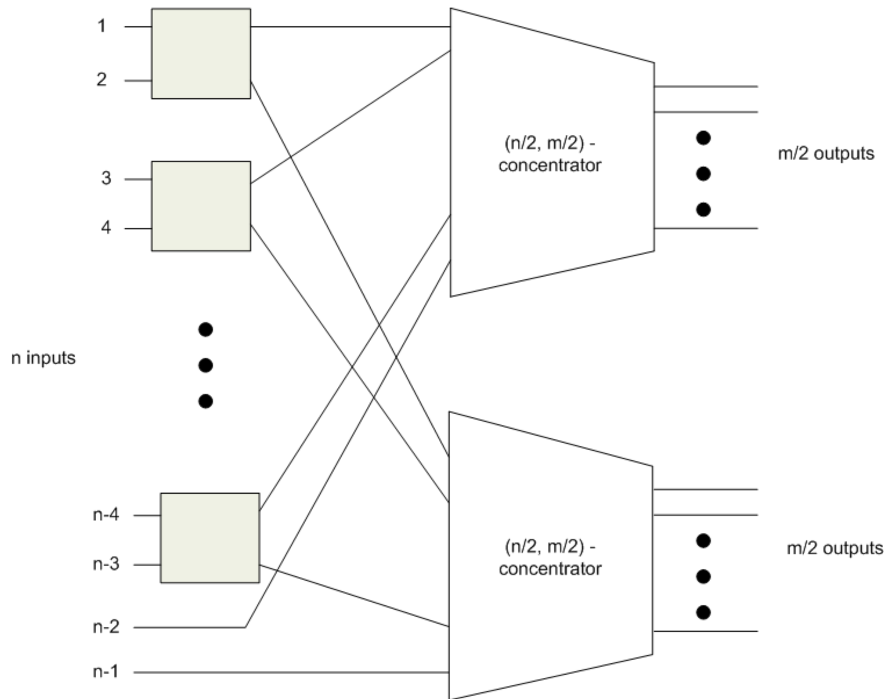

Figure 3.2: Concentrator Construction

The number of multiplexers and flip flops required to build a concentrator was found from [28] and is provided below. Since the select line of each 2:1 multiplexer is programmable, it is assumed that a flip flop will be required to store the configuration of each one. The area

can be represented as:

$$(mux_area + ff_area) * ((n/m) * (m * log(m)) + m * (2 * (n/m) - 1))$$

which simplifies to:

$$(mux_area + ff_area) * (n * log(m) + 2n - m) \tag{3.8}$$

In the equations above, n is the number of inputs to the concentrator and m is the number of outputs. In the overall debugging architecture, m is referred to as S_{width}, since the concentrator is being used to generate a signature.

3.3.3 Coarse-Grained Concentrator

The concentrator can be generalized to make it bus-based rather than bit-based. Given a set of bits of size k (a bus), this allows the flexibility to route any m out of n sets to the m output buses, also of size k. Figure 3.3 shows the construction of a coarse-grained concentrator with $k = 4$, $m = 1$, and $n = 2$. This concentrator allows either set A or set B to be directed to the output but does not allow a combination of bits from sets A and B to be directed to the output together. This generalization removes the guarantee that the concentrator be non-blocking, since it does not allow the separation of the bits within a set. The benefit of this reduction in flexibility is that the total number of flip flops required to programmably route the sets is proportional to the number of sets, rather than the number of total inputs. The reason for this is because the coarse grained concentrator is equivalent to k identically routed concentrators with n inputs and m outputs. The multiplexers of each concentrator can be controlled by the same flip flops.

The area of the coarse grained concentrator is similar to the area for the concentrator in [29] with a scaling factor for the number of flip flops corresponding to the size of the sets k. In the equation below, m corresponds to the number of sets of outputs of size k and

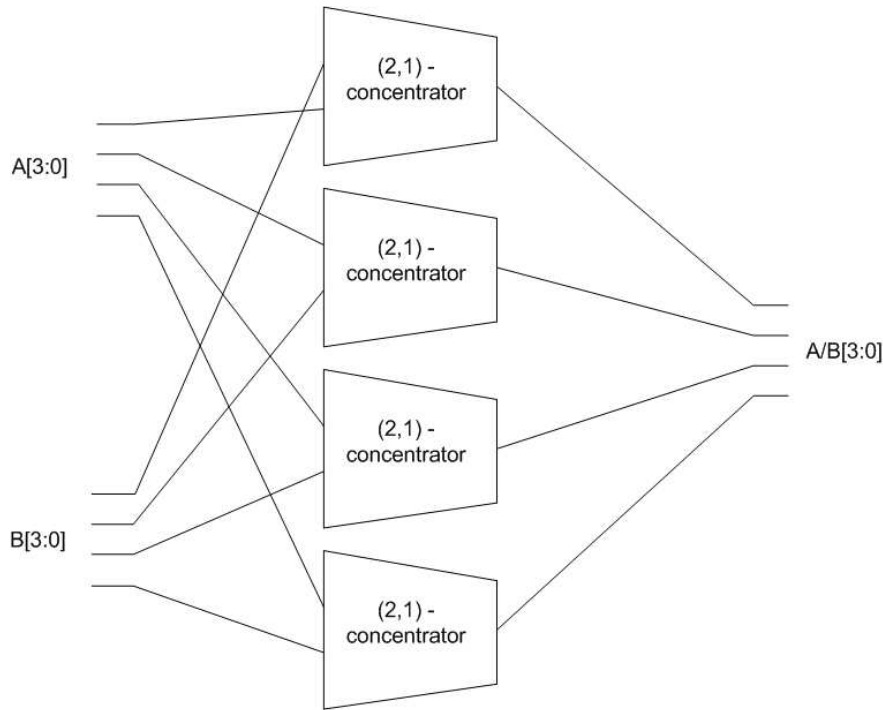

Figure 3.3: Coarse Grained Concentrator with $k = 4$, $m = 1$, and $n = 2$

n corresponds to the number of sets of inputs of size k. The area of the concentrator for different set sizes is shown in Figure 3.4. The figure shows that the benefit of increasing the concentrator coarseness diminishes as the coarseness grows. The area of the multiplexers begins to dominate the area of the flip flops as the set size grows larger than 16.

$$(k * mux_area + ff_area) * (n * log(m) + 2n - m) \quad (3.9)$$

3.3.4 Hash Function

In the previous two variations on the signature creation circuit, certain flip flops are connected to the signature collection circuit without any compression. It is also possible that the signature be a more complicated function of the state bits. A universal hash function can be used to provide information which is useful to BackSpace with far fewer bits than the full state. One of the advantages of a universal hash function is that its configuration is

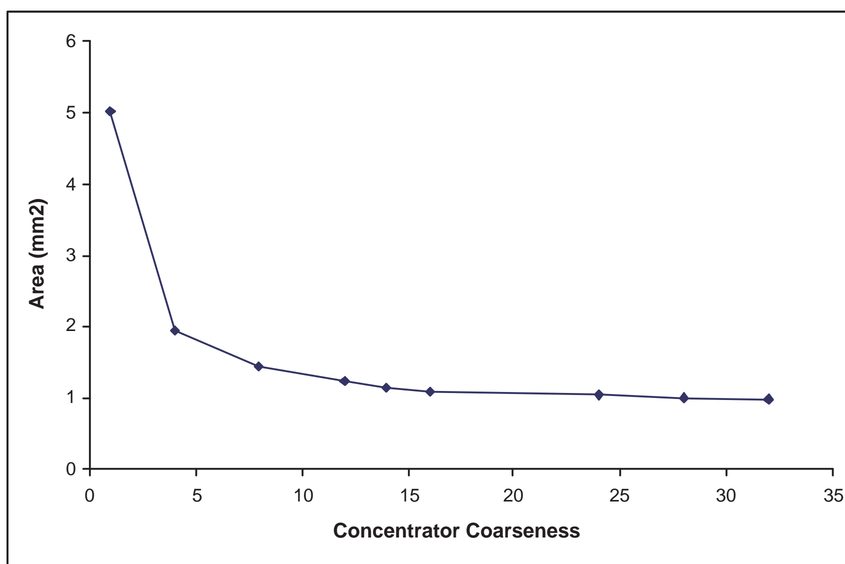

Figure 3.4: Concentrator Area for Different Set Sizes using N_{mon}=10000, S_{width}=0.4 and C_{cycles}=1

randomly chosen, which means that it does not require knowledge of the input function (the chip's state) to be able to provide good compression [7]. This means that the same type of hash function can be used on different designs with similar results. Quantitative benefits of using a hash function are provided below. A version of a universal hash function has been presented as X-Compact in [20], which represents the hash function as a matrix of ones and zeros.

The same representation of a hash function is used for the debugging architecture. Given a hash function with n inputs and m outputs, a representative matrix is created with n columns and m rows. The function representing each output is the exclusive-or of the inputs (columns) that have a one in that output's row. An example hash matrix and its corresponding circuit is shown in Figure 3.5.

Experimental results of using the BackSpace algorithm on a rebuild of a Intel 8051 microcontroller running in a simulator are provided in [23]. The results provide information on the impact of using a 281 bit signature generated by a hash function compared to a 281 bit hard-wired signature. The hash function is given by a hash matrix populated with 1.5%

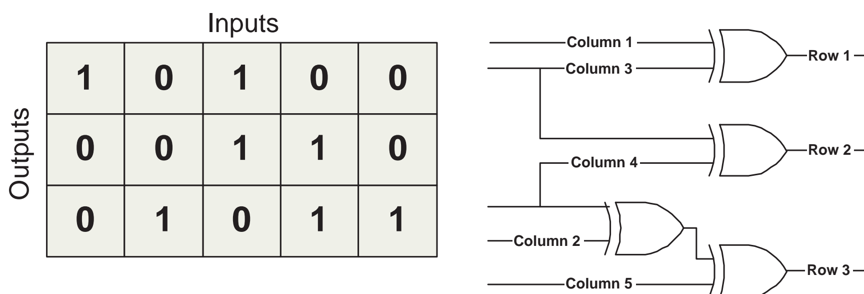

Figure 3.5: Hash Function Matrix and Corresponding Circuit

ones. For ten simulated crash states, the BackSpace algorithm attempted to BackSpace 500 cycles using both the hard-wired signature and the hash-generated signature. For each of the ten 500 cycle histories, the largest pre-image encountered by the BackSpace algorithm was recorded. Using the hash-generated signature resulted in a reduction of 54.21x in the size of these largest pre-images as compared to the hard-wired signature.

Each output (row) in the hash circuit is the exclusive-or (XOR) of one or more of the inputs (columns). The number of 2-input Xor gates needed to create an n-input XOR gate is $n - 1$, as shown in Figure 3.6. The number of XOR gates required to build one output of a hash function given by a representative matrix is therefore the number of ones in that output's row minus one. In fact, the total number of XOR gates required to build a hash function given by a representative matrix is the total number of gates required by all outputs, or the total number of ones minus the number of rows in the matrix. The equation for the area of the hash function is shown below, where m is the number of outputs, n is the number of inputs, xor_area is the standard cell area for a 2-input XOR gate, and $fraction_ones$ is the proportion of the matrix that is populated with ones.

$$xor_area * m * ((n * fraction_ones) - 1) \tag{3.10}$$

A lower fraction of ones in the hash matrix results in a lower area penalty. The hash function must use as low a one fraction as possible without sacrificing its quality. As shown

above, it was found in [23] that a hash function with 1.5% ones causes BackSpace to create significantly smaller pre-images than a hard-wired signature would.

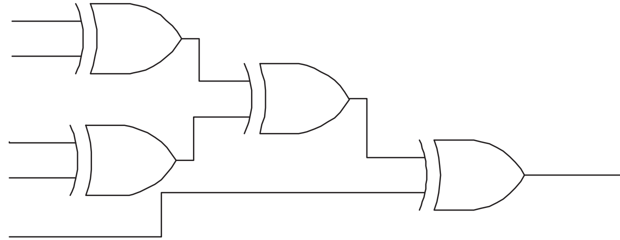

Figure 3.6: Construction of 5-Input XOR Gate using 4 2-Input XOR Gates

3.4 Signature Collection

The signature collection circuit is responsible for recording a history of the most recent signatures before a breakpoint signal is received. These signatures must then be scanned out for use by the BackSpace algorithm, where they are used to prune the generated pre-images.

The signature collection circuit is essentially a First In First Out (FIFO) buffer containing a trace of signatures, with some control logic to stop the buffering at an appropriate time. It is composed of some storage, which can be either flip flops or RAM, and a read/write address register. The address register is incremented each clock cycle as long as the breakpoint signal is not received so that the oldest entry in the FIFO is constantly being overwritten. Additionally, the writing of memory is also enabled as long as a breakpoint is not yet reached. Once the breakpoint is reached, the read/write address register is used to read data in order to unload the contents for analysis.

The area of the signature collection circuit is composed of the SRAM or flip flop area and the address logic area. The SRAM area was calculated using a model originally used in [40], which provides SRAM area in single memory bit equivalents. Since the model was designed for 0.35um technology, it was re-validated against industrial memory blocks in 65nm. It was determined to be accurate for single-port memories. A single memory bit equivalent for

0.18um technology was derived by scaling 65nm data obtained from [28].

The flip flop area of the signature collection circuit storage can be derived much more easily than the SRAM area and can be expressed as:

$$S_{width} * C_{cycles} * ff_area \tag{3.11}$$

where S_{width} is the size of the signature being stored every cycle and C_{cycles} is the number of signatures being stored. The address logic area for both the sram and flip flop configuration can be expressed as:

$$log_2(C_{cycles}) * ff_area \tag{3.12}$$

where $log_2(C_{cycles})$ represents the number of address bits (flip flops) required to address C_{cycles} collection cycles. The total area of the signature collection circuit using flip flops is:

$$S_{width} * C_{cycles} * ff_area + log_2(C_{cycles}) * area_per_pointer_logic_bit \tag{3.13}$$

The signature collection circuit using flip flops is more area efficient than the SRAM when C_{cycles} is less than 3. When C_{cycles} is greater or equal to 3, the area of the sense amps required for SRAM is amortized over enough cycles to make it worthwhile.

3.5 Tradeoffs

Given the number of choices available for each part of the debugging architecture, it is important to quantify the area cost of each configuration of each component. We consider four main parameters.

S_{width} This is the bit width of the signature. This affects the width of the signature collection circuit, as well as the signature creation circuit.

C_{cycles} This is the number of cycles for which signatures are stored in the signature collection

circuit.

N_{mon} This is the total number of bits being monitored. In this thesis, it is assumed to be equal to the number of flip flops in the design.

$bp_{fraction}$ This is the fraction of N_{mon} bits being used to create a breakpoint.

3.5.1 Area as a Function of S_{width}

To understand the impact of the signature size on the area of the debug architecture, the area model was used to generate estimates for a range of S_{width} values. Figure 3.7 provides comparisons of these estimates for the different signature creation schemes explained earlier. The area results are only for the signature creation circuits and assume $N_{mon} = 10000$ and $C_{cycles} = 5$. The hard-wired bits configuration is not included because its logic area is zero. The graph shows that unless S_{width} is extremely small, the hash function appears to be far more costly than either of the concentrators. Additionally, the coarse grained concentrator with a set size of 16 is approximately 5 times smaller than the fine grained concentrator. The final insight is that the concentrator area does not change very much as a function of S_{width}.

For any given S_{width} and C_{cycles}, any signature creation scheme will obviously be higher area than no signature creation scheme (hard-wired bits). The benefit of the concentrator is its flexibility, while the benefit of the hash function is a compressed signature. Both these benefits should allow for a smaller signature that is equally effective as the larger hard-wired signature. Since equally beneficial configurations of hard-wired, hash function, and concentrator signature creation circuits could have different S_{width} values, it is not enough to compare the area of these functions with the same S_{width} value. Instead, it is useful to look at how much smaller the S_{width} of, for example, a hash function must be so that it incurs an equivalent area overhead as hard-wired bits.

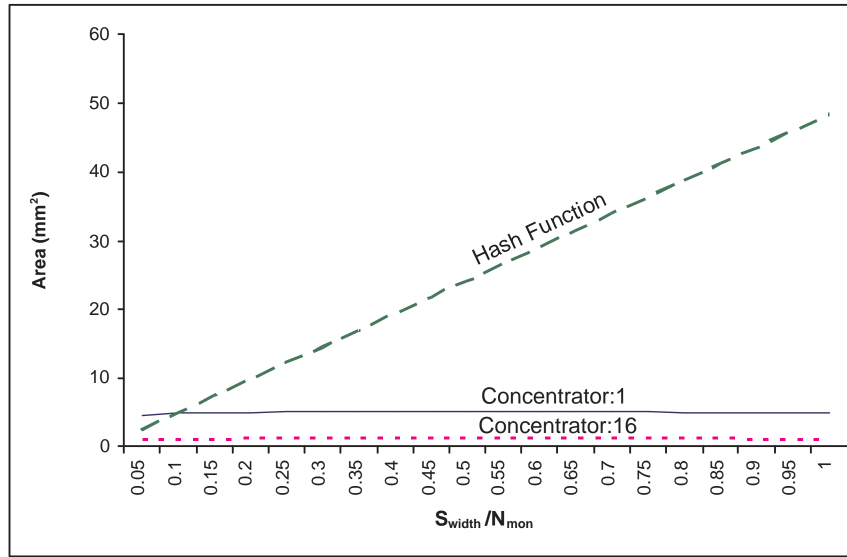

Figure 3.7: Signature Creation Circuit Area for $Nmon$ of 10000

3.5.2 Signature Creation Circuit Comparison with Differing S_{width} Values

The decrease in S_{width} achieved by using a concentrator or a hash function results in a decrease in area for the signature collection circuit proportional to C_{cycles} since the number of bits stored is $C_{cycles} * S_{width}$. Because the choice of signature creation circuit affects the area of the signature collection circuit, it is appropriate to compare the combined areas of both of these circuits. It is interesting to consider the degree to which a concentrator or a hash function must decrease the required size of S_{width} so that the combined area of the signature creation and collections circuits is less than it would be using a hard-wired signature creation scheme.

The area model was again used to compare area values of four competing signature creation configurations. Area estimates were gathered, varying both S_{width} and C_{cycles}. For different values of C_{cycles}, the S_{width} value was recorded which produced an area overhead equal to that produced by a configuration using hard-wired bits.

The combined areas of the signature creation and signature collection circuits for sig-

nature creation circuits is compared against five baseline configurations, each of which uses no signature creation circuit (hard-wired bits), and had S_{width} values of 0.2, 0.4, 0.6, 0.8, and 1. It will be shown later than a hard-wired signature creation scheme with an S_{width} of 0.4 is adequate for the prototype design, but this may not necessarily be the case for all designs. For each baseline and for a range of C_{cycles} values, the S_{width} values required for signature creation circuits to have equal area compared to the relevant baseline are shown in Figures 3.8 to 3.12.

The figures show that a coarse grained concentrator with a set size of 16 could be a better option than using hard bits if the number of cycles of signatures stored in the signature collection circuit is large and if the hard-wired S_{width} is also required to be large. In order for the hash function to be useful, it would need to produce a signature that is composed of very few bits, which provides the same information as a much larger signature composed of hard bits. With a smaller S_{width}, the area overhead of the hash function could be worth it if it resulted in a smaller area overhead for the signature collection circuit, which has to store smaller signatures. The benefit of a smaller S_{width} is multiplied by C_{cycles} since less area is spent on every cycle of history in the collection circuit. From these graphs, it can be seen that unless C_{cycles} is large, it is better to save the area required to implement a complicated signature creation scheme and instead use a wider hard-wired signature at the cost of having to spend a bit more area on a signature collection circuit with a wider S_{width}.

3.5.3 Area Contribution of Each Component

To understand the relative impact that each component has on the overall area of the breakpoint architecture, the contribution of each is plotted in Figure 3.13 as a function of C_{cycles}. A value of $N_{mon} = 10000$, $S_{width} = 0.25$, $bp_{fraction} = 1$, and a concentrator with a coarseness of 16 was used. The figure shows that for low values of C_{cycles}, the signature collection circuit occupies less area than either the breakpoint circuit or the signature creation circuit. For higher values of C_{cycles}, the area contribution of the signature collection circuit becomes more

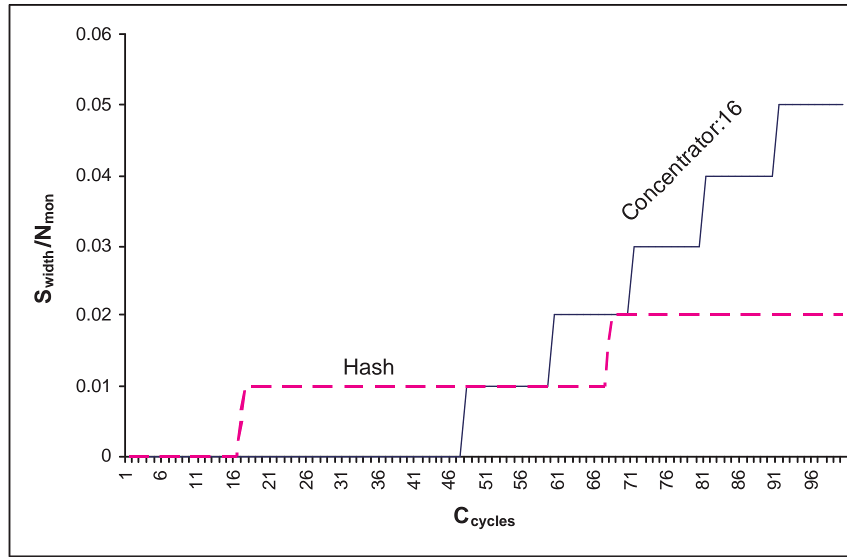

Figure 3.8: S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.2

significant. This helps explain why a signature creation circuit that produces a signature with smaller S_{width} can be worthwhile if C_{cycles} is high, since the reduction in the signature collection circuit is more significant. For a hard-wired signature scheme, the contribution of the signature creation circuit would disappear. However, the contribution of the signature collection circuit would increase since S_{width} would need to increase, due to the lack of flexibility when using hard-wired bits.

3.5.4 Breakpoint Circuit Area

Figure 3.13 shows that the breakpoint circuit occupies a significant fraction of the overall area. The area occupied by the breakpoint circuit is especially significant for low values of C_{cycles}. This observation leads to the possibility of a large area reduction by using a partial breakpoint circuit with a low $bp_{fraction}$ rather than a full breakpoint circuit. Figure 3.14 shows the area for the breakpoint circuit as a function of $bp_{fraction}$. Both the single breakpoint configuration and the two breakpoint configuration are shown. The area decreases linearly as $bp_{fraction}$ decreases.

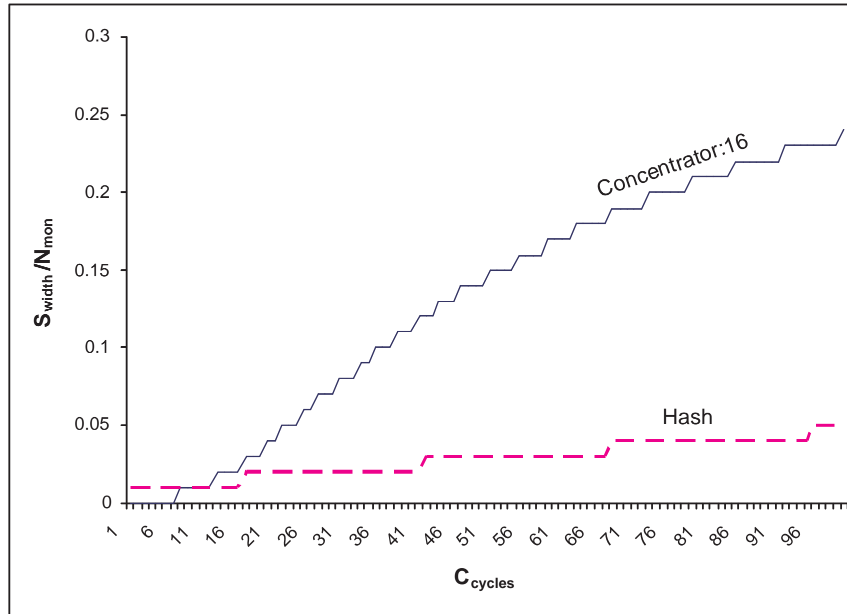

Figure 3.9: S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and baseline with S_{width} of 0.4

As was shown in Figures 3.8 to 3.12, a hard-wired signature is preferable to a flexible signature generated by a concentrator when C_{cycles} is quite low. When using a hard-wired signature, the breakpoint circuit becomes the main area contributor.

3.5.5 Area Overhead of Prototype

The area data presented in previous sections has referred to absolute area, rather than area overhead. This is because the area overhead of the debugging architecture depends on the ratio of flip flops to other circuit elements in an integrated circuit, which varies from chip to chip.

In Chapter 4, the debugging circuitry is added to an open-source processor (OpenRISC 1200). Since the debugging architecture is implemented on a prototype, the specific area overhead data for this design can be presented. This data is only valid for other designs if the proportion of area attributed to flip flops stays the same.

The area overhead was achieved by synthesizing the OpenRISC 1200 processor RTL to

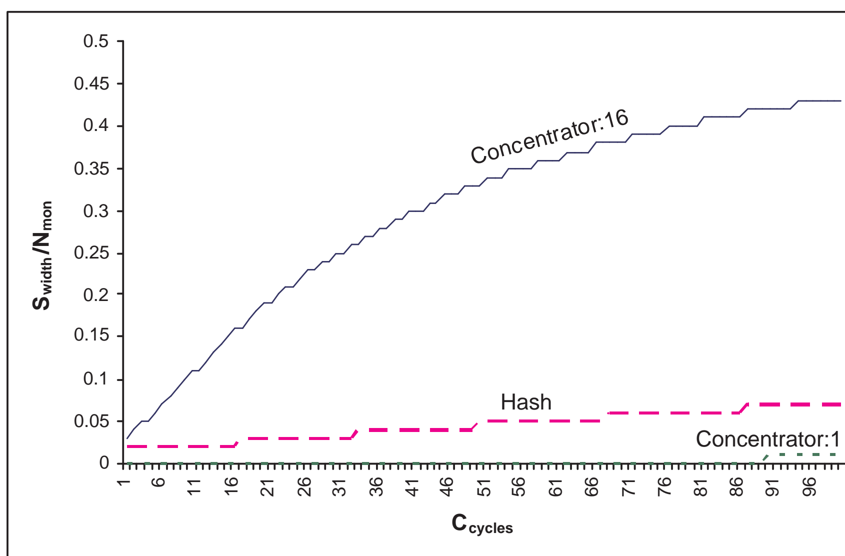

Figure 3.10: S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.6

the same TSMC 0.18um standard cell library that was used to calibrate the area model. The area of the debugging architecture produced by the area model was then divided by the area of the OpenRISC processor to get the area overhead.

The prototype uses $C_{cycles} = 1$ for reasons that will be explained in the next chapter. Additionally, hard-wired bits were used as the signature creation circuit with $S_{width}/N_{mon} = 0.4$. The area overhead contribution of the breakpoint circuit (using one partial breakpoint and two partial breakpoints) and the signature collection circuit are shown in Figure 3.15. The minimization of the partial breakpoint bits with the goal of achieving a lower area overhead is discussed in Chapter 4.

3.6 Chapter Summary

This chapter presented a parameterized area model, which provided area estimates for several variations of each component in the debug architecture. The trade-offs associated with each component of the debug architecture were examined in terms of area. Additionally, the effects that improving the Signature Creation Circuit had on the Signature Collection Circuit were

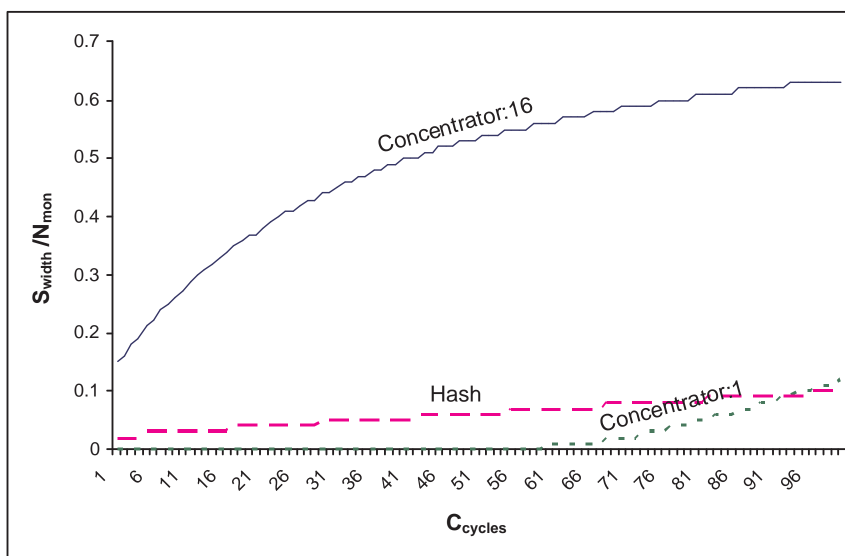

Figure 3.11: S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 0.8

examined. The chapter highlighted the importance of a partial breakpointing system with a low $bp_{fraction}$. It also explored the area overhead of implementing the debug architecture on the OpenRISC 1200 prototype (Chapter 4), which showed that the area overhead could dip slightly below 20% with a small $bp_{fraction}$ and a signature consisting of 40% of N_{mon} bits. Further reductions to the area overhead are possible by reducing the size of the signature, a topic which will likely be addressed in Flavio De Paula's PhD thesis[22].

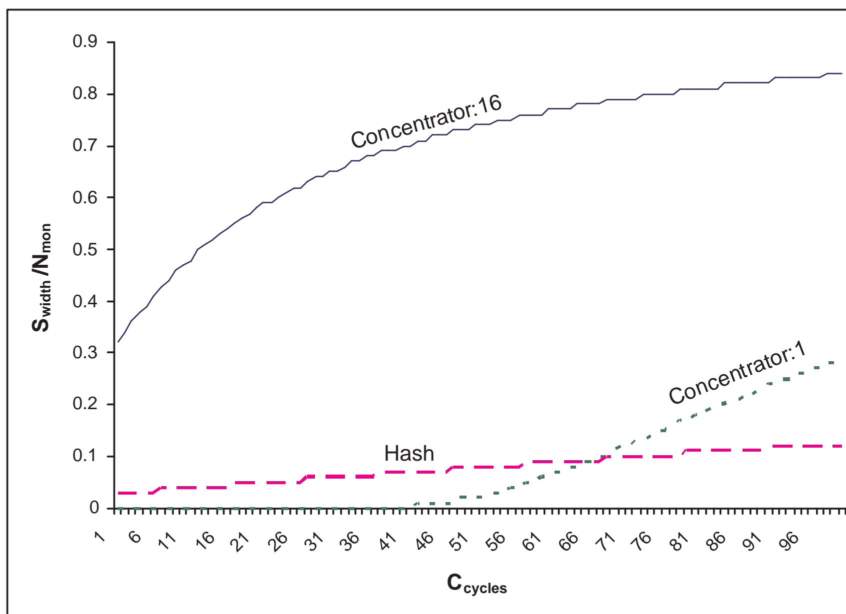

Figure 3.12: S_{width}/N_{mon} vs. C_{cycles} for $Nmon$ of 10000 and Baseline with S_{width} of 1.0

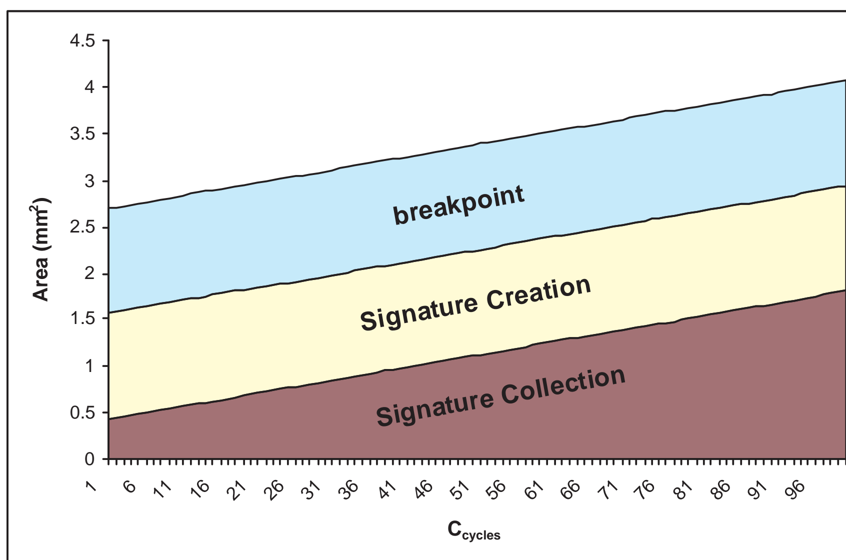

Figure 3.13: Area Contribution of Architecture Components using $N_{mon} = 10000$, $S_{width}/N_{mon} = 0.25$ and a Concentrator with a Coarseness of 16

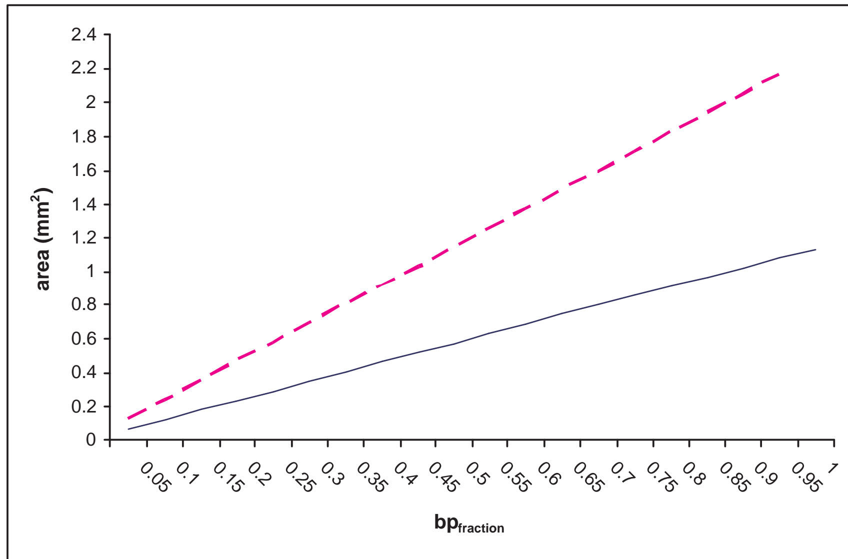

Figure 3.14: Breakpoint Area vs $bp_{fraction}$

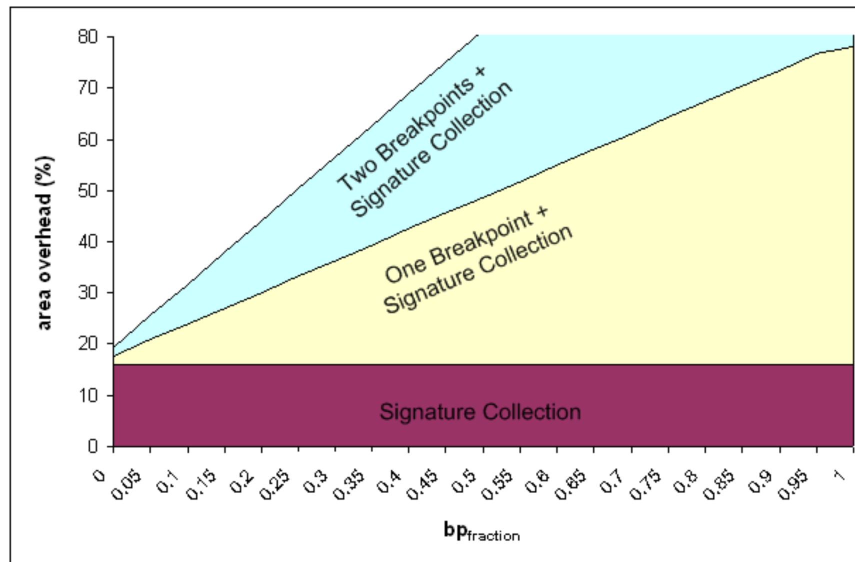

Figure 3.15: PPrototype Area Overhead Contribution vs $bp_{fraction}$

Chapter 4

Prototype and Breakpoint Bit Selection

As a proof-of-concept of the BackSpace approach, a prototype was developed consisting of a design to be debugged and the debugging hardware described in Chapters 2 and 3. This prototype was implemented on an FPGA, which was connected to a PC running BackSpace in order to run debugging experiments. Section 4.1 motivates the need for this prototype.

The hardware implemented on the FPGA is described in Sections 4.2 and 4.3. The interaction between the host-PC and the prototype is explained in Section 4.4. Section 4.5 provides details on the sources of non-determinism found in our prototype. Preliminary results are shown in Section 4.6. Finally, the choice of breakpoint bits for a two partial breakpoint system operating on the prototype is explored in Section 4.7.

4.1 Motivation for Prototype

Although BackSpace was previously tested with a simulated design, a hardware prototype was required because software simulation is slow. The speed at which the hardware prototype can run means that longer experiments are feasible. This advantage is especially significant for our debugging methodology because the chip must be re-run many times to obtain a trace. In our prototype, since the signature collection circuit has only one cycle of history, the chip must be re-run at least once for every cycle of trace required. This is because with only one cycle of signature history, BackSpace can only go back one cycle at a time. When

simulating the operation of the OpenRISC 1200 rather than prototyping it, obtaining large traces becomes infeasible.

In addition to having a speed advantage over simulation, a hardware prototype also has the potential to expose problems that would not have been discovered in simulation. One such problem that was discovered in hardware, but that was not noticeable in simulation, is the problem of non-determinism exposed by crossing clock domains.

4.2 OpenRISC 1200

A Verilog implementation of an open source processor called the OpenRISC 1200 was chosen as the baseline design. The OpenRISC 1200 is a Harvard microarchitecture 32-bit scalar RISC processor with a 5-stage integer pipeline. It also supports virtual memory and basic DSP functionality. The processor comes packaged with an optional Universal Asynchronous receiver/transmitter (UART) module for serial communication.

When choosing a processor for the prototype, the number of gates and flip flops was an important consideration. It was desirable that the processor be as large as possible while staying within the limits imposed by our implementation of the BackSpace algorithm. The latter concern is critical, since the computation of pre-images becomes very time-consuming for larger states. The OpenRISC was felt to provide a compromise between earlier work with small microcontrollers [23] and potential later work with industrial designs such as the OpenSPARC.

Along with the OpenRISC, another open source processor, the Leon3 by Aeroflex Gaisler, was also considered. This choice was ruled out because the Leon3 is geared more towards users who would like to quickly implement a system on a supported board. The documentation provided by Gaisler provides information on how to use scripts that will dynamically generate the VHDL based on the user's board and their chosen processor specifications. Unfortunately, since we do not have access to one of the supported boards, using the Leon3

would have been difficult. The documentation for the OpenRISC, however, includes information on how to modify the source code manually so that the user can customize their implementation. Considering the significant changes to the RTL that are involved in adding the debug architecture, the lack of automation for the OpenRISC was actually considered a benefit.

The OpenRISC RTL was downloaded from [11]. Instruction from [25] were used to pare down the RTL to the essentials and customize it for use on a Xilinx FPGA. The RTL was then synthesized with Synopsys Design Compiler using the ISCAS 89 gates library to facilitate the insertion of our debugging architecture.

4.3 Debugging Architecture Insertion

The debugging architecture presented in Chapters 2 and 3 was added to the OpenRISC processor by modifying the gates-level Verilog. Each component of our debugging architecture is described below:

Signature Collection Circuitry In our prototype, rather than pre-selecting which state bits would be used to construct the signature, we monitor *all* 3007 state bits and allow external software to select the bits that constitute the signature. Although we do not expect a realistic implementation to monitor all state bits, this allows us to investigate different state selection strategies (and different values of N_{mon}) without having to re-synthesize the design.

Signature Creation Circuitry Since we collect all 3007 state bits, and provide them all to the external software, we do not employ any sort of signature creation circuitry on chip. The 3007 state bits are stored in a single 3007-bit wide register. In our implementation, the register is in addition to the flip-flops that are part of the design. The details of how this register is unloaded to external software can be found in Section 4.4.1.

Breakpoint Circuitry We implemented two full-breakpoint schemes. Since there are 3007 state bits, this required adding 6014 flip-flops to store the match target registers, and two 3007-input wide XOR to compare the current state to the target state. Since we wished to experiment with partial-matching schemes, we added mask registers, which indicate whether each state bit is to take part in the match process, as shown in Figure 4.1. In a real implementation, the mask register would not be required, since the selected state bits could be hardwired to the XOR circuitry. Using the mask, however, allows us to experiment with different breakpoint strategies without having to re-synthesize the design. The loading of the target state mask registers and the target state registers is described in Section 4.4.1.

Figure 4.1: Breakpoint Masking for Partial Breakpoints

4.4 System Operation

The debugging system consists of the BackSpace algorithm, the OpenRISC 1200 augmented with debug architecture, and the communication interface between the BackSpace algorithm and the debug architecture. This communication interface is used to observe and control the operation of the OpenRISC 1200, and is described in Sections 4.4.1 and 4.4.2. The following simplified system level description of the operation for the prototype helps to motivate the types of observability and controllability that are required. A more detailed and realistic debug flow is provided in Chapter 2.

1. The breakpoint register is loaded with an initial target state, which acts as the crash state.

2. A signal is sent to OpenRISC 1200 to make it run until the breakpoint signal is produced.

3. The signature is unloaded from the Signature Collection Circuit.

4. BackSpace calculates the pre-image consisting of a set of candidate states.

5. The breakpoint register is loaded with the first candidate state.

6. A signal is sent to OpenRISC 1200 to make it run until the breakpoint signal is produced or a timeout occurs. If a timeout occurs, the candidate is not the predecessor state. In this case, the breakpoint register is loaded with the next candidate state, and the process is repeated until a breakpoint signal is produced.

7. Steps 3 to 6 are repeated until the trace is long enough.

4.4.1 System Hardware

In order to accelerate the implementation of OpenRISC 1200, it was incorporated into a larger hardware system design intended for use on the Amirix AP1000 prototyping board.

The system design in which the OpenRISC was incorporated was provided by CMC Microsystems as a Xilinx EDK project targeted towards to the Amirix AP1000 board. The overall system, with the OpenRISC incorporated, is shown in Figure 4.3. The original system provided by the CMC consisted of a PowerPC, which is a hard IP block embedded on the Virtex II chip, a UART, and a memory controller, along with some C code to perform some basic memory and UART operations.

The OpenRISC was connected to the 32-bit on-board peripheral bus (OPB) of this original system and was configured using the EDK "create peripheral wizard". This wizard creates an interface, called IPIF, between the PowerPC and the peripheral to facilitate communication. This interface consists of a set of software registers in the peripheral, as well as a set of C functions for the PowerPC, which are used to access these software registers. Table A.1 in Appendix A shows how each of the 32 software register was used in our implementation.

The most important use of the software registers was the loading and unloading of the breakpoint registers and the signature collection circuit. The breakpoint write select registers (slv_reg9 and slv_reg10) use one-hot encoding to address into the 3007-bit target state register. The target state register is loaded using 47 separate 64-bit loads, for which the data is found in the breakpoint data write software registers (slv_reg7 and slv_reg8) as shown in Figure 4.2. In similar fashion, the signature collection circuit is unloaded by using the trace buffer read select software registers and the trace buffer data read software registers. Some software registers are also used to send and receive memory reads/writes to/from the PowerPC.

The OpenRISC includes its own UART module, which was directly connected to the second UART serial port provided on the Amirix AP1000 board, bypassing the OPB bus. The first UART serial port was used by the PowerPC for its I/O requirements. These two UART ports were connected to the PC's COM1 and COM2 serial ports.

There are several advantages to adding the OpenRISC to the EDK design provided by the

Figure 4.2: Loading of Target State Register using Software Registers

CMC, rather than creating a new design specifically for the OpenRISC. By using the CMC design that was made for the Amirix 1000 board, many low level implementation details were already taken care of. For example, the CMC design had already mapped FPGA pin numbers to net names. This made it easy to access the FPGA I/O. Additionally, because the PowerPC was already correctly connected to the memory controller, the on-board SDRAM was easily accessible using C code running on the PowerPC. One of the UARTs was also easily accessible from the PowerPC using a custom C printf function, which accessed a driver written for the AP1000 board.

In order to leverage this existing functionality, all memory accesses were sent to the PowerPC, which then translated them into SDRAM memory accesses using a routine written in C. The memory addresses and data were also sent by the PowerPC to the UART to enhance visibility onto the OpenRISC design and the debugging architecture. This methodology is similar to the one used by the Research Accelerator for Multiple Processors (RAMP) team to help them implement the OpenSPARC processor on a Xilinx Virtex5 FPGA [37]. They used the embedded Microblaze processor in a similar way that we are using the PowerPC processor.

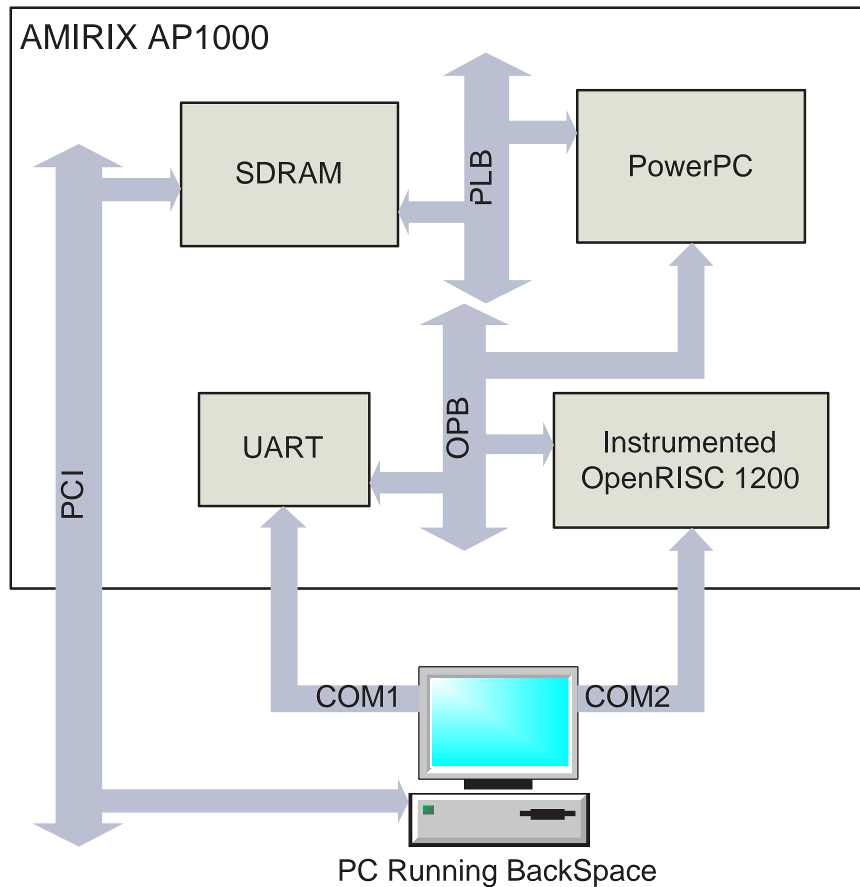

Figure 4.3: System Level View of Debugging Methodology

4.4.2 Communication between BackSpace and the Hardware

The Amirix AP1000 prototyping board was connected to a PC's PCI slot. A software module called APControl, which is bundled with the Amirix board, provides access to the on-board SDRAM used by the PowerPC. The BackSpace software, running on Linux, can thereby access the on-board SDRAM by issuing APControl commands. The net effect of this functionality is that both BackSpace and the code running on the PowerPC have access to the same memory space.

A mailbox system was used to leave messages in memory between BackSpace and the code running on the PowerPC. The mailbox system which we used was developed by Flavio De Paula and will be reported in his PhD thesis [22]. Specific memory locations are used by

BackSpace to store values which the PowerPC loads onto the debugging architecture, such as the Breakpoint Register and the Breakpoint Mask Register. Other memory locations are used by the PowerPC to unload data from the OpenRISC for use by BackSpace, such as the contents of the Signature Collection Circuit. Additional memory locations are used by both the PowerPC and BackSpace to send and receive acknowledgements and status updates.

4.5 On-Chip Non-Determinism

While running an application on the OpenRISC and testing the breakpoint circuit, it was observed that, given identical starting conditions and external stimuli, some bits differ from run to run. This resulted in a breakpoint signal sometimes being produced, and sometimes not under the same conditions. This was due to the OpenRISC 1200 main memory running on a different clock domain than the OpenRISC 1200 core. Access to the SDRAM, onto which the OpenRISC's memory is mapped, is on a clock domain running at 80MHz, while the OpenRISC 1200 core is on a separate independent clock domain running at 10MHz. The relative phase of the two clocks can vary from run to run, which means that the number of clock cycles required to access memory from the OpenRISC 1200 can vary. Methods for coping with non-determinism are discussed in Chapter 2.

4.6 Preliminary Results

Using a simple "Hello World" program running on the OpenRISC 1200, the BackSpace algorithm was given control of the debugging hardware in order to produce traces from arbitrary crash states. The BackSpace algorithm cycled through each candidate state in a pre-image until one of them resulted in a successful breakpoint. The only limitation was that if the pre-image grew larger than 300 cycles, the BackSpace algorithm would stop running. The system was generally able to backspace several hundred cycles before this situation occurred.

Detailed experiments were run with two applications: gcd (euclidean algorithm for greatest common divisor) and prime (Sieve of Eratosthenes for computing prime numbers). The results were presented in [24]. Three crash states were inserted for each application for a total of six traces. BackSpace was run starting from each crash state until the 300 state pre-image limit was reached or until a 500 cycle trace was produced. Three out six traces reached the pre-defined BackSpace limit of 500 cycles. The average trace depth of the other three traces was 378 cycles before reaching the 300 state pre-image limit.

The raw trace data produced by BackSpace for all three applications consisted of the value of every flip flop for several hundred cycles preceding the crash state. Since it was difficult to understand of the operation of the chip from this raw data, a script was written to convert this data into Value Change Dump (VCD) format, which is a standard waveform database format. This VCD file could then be viewed with any waveform viewer. A sample trace produced by BackSpace is shown in Figure 4.4.

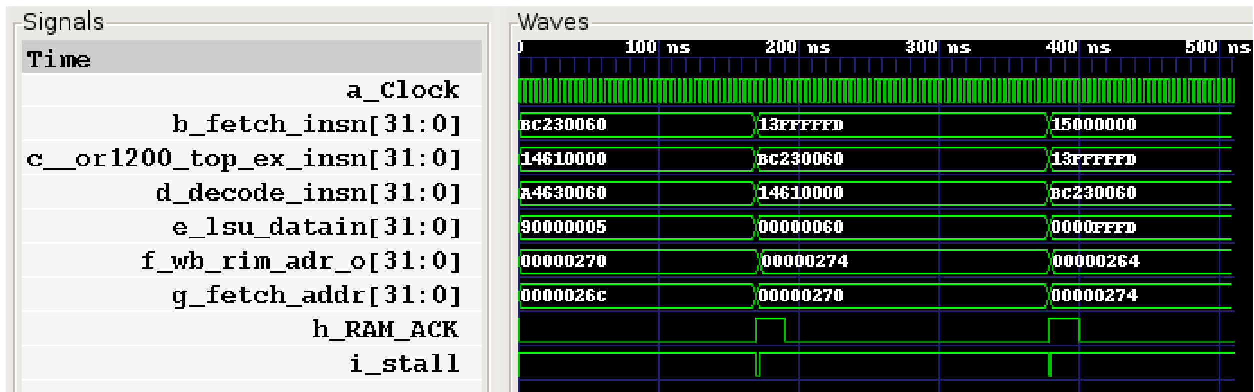

Figure 4.4: Waveform Viewer Representation of Trace Produced by BackSpace

4.7 Partial Breakpoint Bit Choice

It is difficult to devise a general methodology for picking breakpoint bits that is applicable to all systems. Methods for doing this are a potential area of future research and are discussed in Chapter 5. The choice of partial breakpoint bits for one particular system is a more

self-contained problem, and is addressed for our FPGA prototype.

The best bits to use for the breakpointing circuit are those that differentiate between correct runs and spurious runs. These bits are the ones that are likely to change when some non-determinism occurs, allowing them to expose the differences between a correct run and a spurious run. Although there are likely some bits that are better than others, it is not obvious which ones these are.

4.7.1 Choosing Bits Randomly

The first method to choosing breakpoint bits is to do so randomly. Choosing bits randomly requires no knowledge of the design, which makes it easier than choosing bits intelligently. In this section, we estimate how effective randomly choosing bits would be.

It is not feasible to experimentally test how effective randomly choosing bits is because we would have to run too many experiments. We would need to run the experiments for a range of number of breakpoint bits. We would also need to run the prototype many times for each data point in order to eliminate noise caused by choosing bits in a lucky or unlucky way. Additionally, this would need to be done for different benchmarks.

Instead, we estimate the effectiveness of randomly choosing breakpoint bits analytically. We estimate the probability that the breakpoint circuit will produce a breakpoint signal on a spurious run. The probability of a false breakpoint occurring on a spurious run is P_{fb}. In order to produce a breakpoint signal, the breakpoint circuit must increment the match counter *target_match_counter* times. This means that over the course of the spurious run, *target_match_counter* false matches must occur to generate a false breakpoint signal.

We make the simplifying conservative assumption that if a false match occurs once, enough of them will eventually occur on the spurious run so that a false breakpoint signal is generated. This means that P_{fb} can be simplified to be the probability that a false match occurs anytime during a spurious run, regardless of the *target_match_counter* value. This assumption can be justified by noting that the small differences between the spurious run

and the correct run will likely persist for many cycles. This can lead the breakpoint circuit to generate more than one false match during this time.

The probability of a false breakpoint can be calculated assuming that the bits in the partial breakpoint circuit are randomly chosen. We define the state in the spurious path with the least number of bits that differ from the full breakpoint state as the corresponding state. We denote the number of bits that differ between the full breakpoint and its corresponding state in the spurious run as $Bits_{differ}$ and the number of bits in the partial breakpoint as $Bits_{pbp}$. A false match occurs if none of the bits that differ are part of the partial breakpoint. The probability of a false match occurring can be expressed as the number of combinations of differing bits that do not include partial breakpoint bits divided by the total number of combinations of differing bits, as shown below.

$$P_{fb} = \frac{\binom{N_{mon} - Bits_{pbp}}{Bits_{differ}}}{\binom{N_{mon}}{Bits_{differ}}} \tag{4.1}$$

4.7.1.1 Modeling Random Bit Choice with Real Data

We perform experiments on the prototype to find the average number bits that differ between a breakpoint state and its corresponding state in the spurious run. In order to gather this data, a Hamming circuit was added to the prototype to compute the number of bits that differ between the programmed breakpoint and the current state. The number of differing bits is called the Hamming value and the Hamming circuit produces a value every cycle. The smallest Hamming value produced during the entire run of the chip is equivalent to the number of bits that differ between the target breakpoint state and its corresponding state in the spurious run. If the value is zero, the run was not spurious.

The average of the Hamming values from several runs was approximately three. Substituting $Bits_{differ} = 3$ into Equation 4.1 provides an equation for the probability of a spurious false matches occurring for our prototype:

$$P_{fb} = \frac{\binom{N_{mon}-Bits_{pbp}}{3}}{\binom{N_{mon}}{3}} \tag{4.2}$$

Figure 4.5 shows the probability of a spurious false match occurring vs. the fraction of N_{mon} bits in the partial breakpoint for our prototype using this equation.

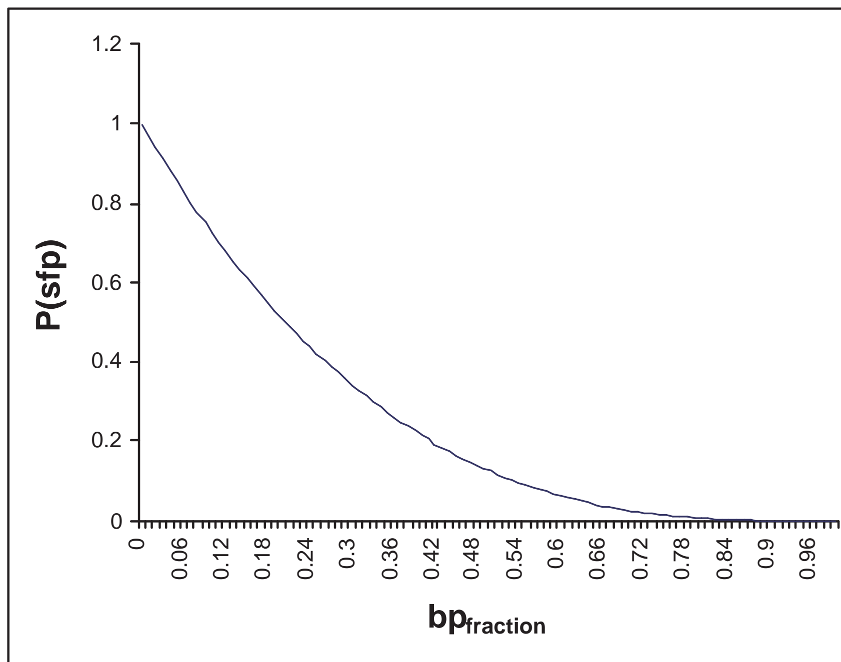

Figure 4.5: Probability of a Spurious False Match vs. $bp_{fraction}$

The figure shows that $bp_{fraction}$ must be very high to avoid spurious false matches. This means that choosing bits randomly is an ineffective methodology. This motivates the need to choose the breakpoint bits intelligently according to which bits are most likely to differ in a spurious run.

4.7.2 Choosing Bits Intelligently

The second method for choosing the breakpoint bits is to do so intelligently, by trying to determine which bits are most likely to differentiate between a correct run and a spurious run.

4.7.2.1 Characterizing Spurious Runs

Since the breakpoint circuit is meant to differentiate between spurious runs and correct runs, it is important to understand how a spurious run differs from a correct run. The basic operation of a system for a spurious run will be very similar to the operation for a correct run, since the same input patterns are applied for both runs. In order for the system to be functionally correct, it should behave in a very similar way given the same input patterns. The differences then, are in *when* the operations are performed. The operations that are performed during a spurious run may occur slightly earlier or later than those that are performed in the correct run.

When comparing a state from the correct run and a state from the spurious run, the differences can be attributed to two sources. The first reason why bits could differ is because the state in the spurious run could be at a very different stage of execution than the state in the correct state. The second reason why bits could differ between the states is because some non-determinism causes the correct run and the spurious run to vary slightly, even if they are in similar stages of execution.

Because there are likely to be many bits that differ between states located at very different stages of execution, it is relatively easy to differentiate these states. It is more difficult to differentiate states that differ as a result of non-determinism, because they could have very few differing bits. It is critical that these few differing bits be part of the breakpoint bits.

4.7.2.2 Gathering Differing Bit Data

To get the target breakpoint bits, we first gather data about the differences between a target breakpoint and its corresponding state in the spurious run. The bits in the breakpoint state can be compared against the bits in the corresponding state in the spurious run to find the bits that differ.

We gathered data on the prototype using a new 3007-bit wide register called the Hamming state register. The Hamming state register is not part of the debug architecture, but is

used to gather information about the prototype. It stores a copy of the breakpoint state's corresponding state that occurs in the spurious run. Whenever the Hamming value is at minimum for the current run, the values of the state bits are copied to the Hamming state register. At the end of the run, the Hamming state register holds the state in the spurious run that has the fewest bits differing with the target breakpoint state.

Data was collected on the prototype running three different applications on the OpenRISC 1200. Sixteen different target states were programmed into the breakpoint circuit, running the chip 30 times for each one. This process was repeated for each application, for a total of 1440 runs of the prototype.

For each Hamming state collected over the 1440 runs, the bits that differ always belong to the same set of 496 bits. A good solution would therefore be to use these 496 as the breakpoint bits. In order to have a partial breakpoint that is smaller than 496 bits, these bits are ranked so that for a smaller breakpoint, only the bits which are most likely to help differentiate between true and spurious runs are used. Two methods of ranking these bits, and the associated results are presented below.

Method 1: Probability of Bits Differing A simple way of ranking the bits in a breakpoint is to use the frequency with which those bits are able to differentiate between a correct run and a spurious run. Using the bits that are most often different between a breakpoint state and the corresponding state in the spurious run should result in a set of bits that is likely to avoid a false match. Bits that are rarely different between the breakpoint state and the corresponding state will not help in most cases since monitoring these bits will not differentiate between the correct and spurious runs. The top 40 most frequently differing bits are shown in Table B.1 in Appendix B.

Method 2: Bit Correlation and Probability of Bits Differing Simply considering the frequency with which a bit is different between a breakpoint state and its corresponding state in the spurious run does not take into account that some bits are highly correlated.

When differentiating between a target state and a corresponding state in a spurious run, it is possible that a set of bits could always be different together. If one bit differentiates between the two particular states, all bits in its correlated set also differentiate between those particular states. In such a case, it is only necessary to include one of these bits in the breakpoint circuit to avoid false match signals when these bits are different since no extra differentiating ability would be gained by including more than one bit from this set. The pair-wise correlations of the 496 bits considered in the previous section were computed and a weight function which considers both the frequency of the bits being different and the correlation between bits was developed and is presented below:

$$Weight(bit) = P_{diff} * (1.05 - Max(correlation)) \tag{4.3}$$

This weight function is re-computed after each bit is added to the set of breakpoint bits. P_{diff} is the probability that a bit differentiates between a target state and its corresponding state in the spurious run. $Max(correlation)$ is the maximum correlation between this bit and any bit already in the partial breakpoint. The correlation values range from -1 to 1. The 1.05 value was chosen to shift the correlation value onto the range 0.05 to 2.05. A low value of 0.05 was chosen instead of 0 in order to act as a tiebreaker between bits that have a correlation of 1. The top 40 bits according to this weight function are provided in Table B.2 in Appendix B.

4.7.2.3 Results

The effectiveness of both methods of prioritizing bits was quantified by measuring the number of false breakpoints. A better set of bits results in fewer false breakpoints. The tests were performed on the prototype running three different programs: uart, gcd, prime. For each of the programs, 50 runs were performed at 15 different breakpoint locations for a total of 750 data points per program.

The number of correct breakpoints, false breakpoints and runs that did not lead to

a breakpoint were counted. The total number of spurious paths is the number of false breakpoints plus the number of runs that did not lead to a breakpoint. The total number of false breakpoints was divided by the total number of spurious paths to get the probability of the breakpoint circuit producing a false match on a spurious path. To quantify the trade-off between the number breakpoint bits and that breakpoint circuit's effectiveness, the above experiment was repeated for a range of breakpoint bit widths. The number of bits in the breakpoint was scaled from 1 bit to 496 bits, in increments of 15 bits.

The results for Method 1 are shown in Figure 4.6. The results for Method 2 are shown in Figure 4.7. The average across the three programs for each of these two methods is shown side by side in Figure 4.8.

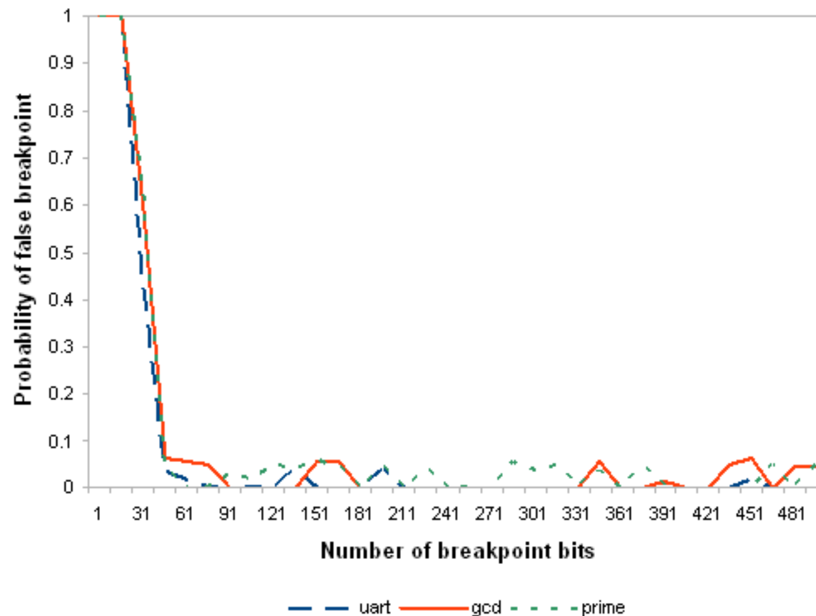

Figure 4.6: Probability of False Match vs. Number of Breakpoint Bits, Prioritizing Bits using Probability of Bits Differing

The results show that for each method, adding more bits to the breakpoint at first greatly reduces the number of false matches observed, but offers diminishing returns in that beyond a certain point, adding more breakpoint bits has little effect.

The important conclusion from the graph is that, in contrast with choosing bits randomly,

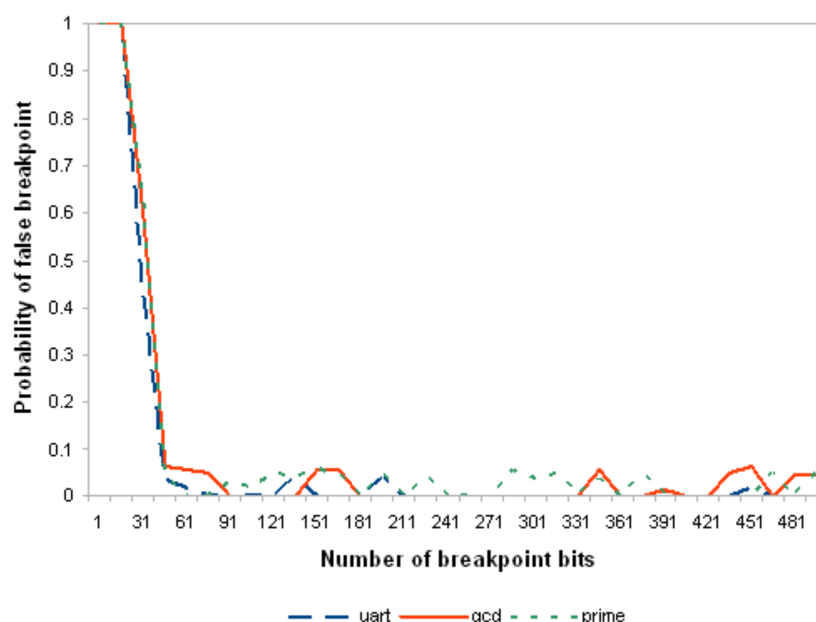

Figure 4.7: Probability of False Match vs. Number of Breakpoint Bits, Prioritizing Bits using Correlation-Based Weight Function

choosing bits intelligently makes it possible to use very few bits in the breakpoint and still maintain a very low chance of a false breakpoint signal occurring. Additionally, Method 2 is significantly better than Method 1. Using 46 bits in the breakpoint, for example, results in a 4.6% chance of obtaining a false match using Method 1 and a 1.1% chance in Method 2. Using 61 bits results in a 2.4% chance for Method 1 and a 0.2% chance for Method 2.

4.8 Chapter Summary

In this chapter, we described the methodology used to prototype our debugging solution. The OpenRISC 1200 was chosen as the processor to be debugged, and was synthesized into basic gates. The design was then augmented with our debugging architecture: a signature collection circuit and a breakpoint circuit. Using Xilinx EDK, the augmented OpenRISC 1200 was incorporated into an SoC design provided by the CMC. The existing functionality of the CMC design facilitated the debugging of our implementation and helped us speed up

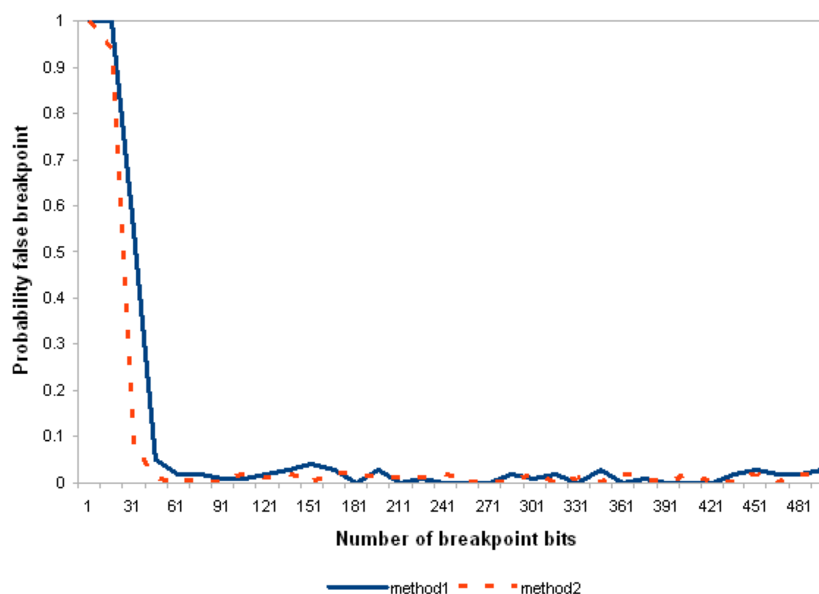

Figure 4.8: Average Probability of False Match vs. Number of Breakpoint Bits for Both Prioritizing Methods

the prototyping process. Finally, we connected the hardware to a PC running our BackSpace algorithm. The PC and the Prototype were able to interact with a mailbox communication system. With a working prototype completed, we were able to run more in depth experiments than had been possible through simulation. We were able to create traces hundreds of cycle in length starting from various crash states.

We then gathered data from the prototype to determine the bits that best differentiate between correct and spurious runs. This data was used to identify a set of 46 breakpoint bits that reduced the chance of false breakpoint to 1.1%. Using the area model presented in Chapter 3, the area overhead of this breakpoint circuit is estimated at 5.09%.

Chapter 5

Conclusion

5.1 Summary

This thesis explored "practical considerations" for the BackSpace debug flow. We considered the area-overhead of the on-chip supporting architecture in addition to on-chip non-determinism and signal propagation delay.

In Chapter 2, we presented modifications to the breakpoint circuit, which significantly reduces its area overhead. A breakpoint system consisting of two partial breakpointing circuits and a new debug flow were used to provide state-accurate breakpoint triggering in a non-deterministic environment. Additionally, through pipelining and further modifications to the debug flow, we eliminated the requirement for the breakpoint signal to be generated and propagated in one cycle.

In Chapter 3, we presented a detailed area model that estimates the standard cell area of each variation of the debugging architecture components. Several signature creation schemes were examined and compared. It was concluded that unless the signature collection circuit has many cycles of storage, a hard-wired signature is the best option. We examined the interaction between architectural parameters to study area tradeoffs for each architectural component. This insight led us to choose appropriate parameters for an FPGA prototype. Using these parameters, the main contributors to area overhead were found to be the breakpoint circuit and the signature collection circuit. This stressed the need for a small set of breakpoints bits to use in the breakpoint circuit.

In Chapter 4, we presented the details of an FPGA prototype, which consisted of our

debugging architecture implemented on OpenRISC 1200(an open-source 32-bit RISC processor with a 5-stage integer pipeline). A system consisting of a PowerPC processor, an SDRAM memory controller, and a UART module was connected to the instrumented OpenRISC to provide visibility onto its memory accesses and to facilitate communication with a PC. The BackSpace algorithm was run on the PC and communicated with the prototype with a mailbox system. The prototype allowed for extensive experimentation and testing of the BackSpace algorithm. We were able to BackSpace for hundreds of cycles starting from various target states.

The second part of Chapter 4 discussed the choice of breakpoint bits for the two partial breakpoint system on the prototype. We compared two different algorithms for intelligently prioritizing breakpoint bits based on experimental data. For each bit, we computed the probability that the bit will differentiate a breakpoint state and its corresponding state in a spurious path. The first algorithm used only this metric to prioritize breakpoints bits. The second algorithm used this probability and the pair-wise correlations of these probabilities. The second algorithm performed better and was able to find a set of 46 breakpoint bits that differentiate between correct and spurious runs 98.9% of the time for our prototype.

5.2 Contributions

This section outlines the main contributions of this thesis.

1. The first main contribution of this thesis was a method to make the debug methodology practical by addressing the area overhead, while also coping with chip realities such as non-determinism and signal propagation delay. A partial breakpoint-circuit was developed, which dramatically reduced the area overhead of the breakpoint circuit, while allowing the debug flow to remain state accurate.

2. The second main contribution of this thesis was an area model, which estimates the standard cell area of the three main components of the debugging architecture. The

model was used to examine the tradeoff between the effectiveness of each breakpoint architecture with its associated area overhead.

3. The third main contribution of this thesis was a hardware prototype of a 32-bit processor instrumented with our debugging architecture.

4. The fourth main contribution of this thesis is a demonstration that it is possible to select a small effective set of breakpoint bits. An algorithm was developed to find these breakpoint bits using experimental data gathered from the prototype.

5.3 Limitations and Future Work

This section presents and explains limitations of the BackSpace debugging methodology. We propose future work or reference existing work that addresses these limitations where possible. We first address the area overhead of the debugging architecture, which still has room for improvement. We next identify limitations of our current method for selecting breakpoint bits and suggest ways in which this selection could be automated. Third, we address the vulnerabilities of BackSpace to non-determinism and propose changes that could limit these vulnerabilities. Finally, we explore scenarios where BackSpace will not find all bugs on chip and discuss other work that can be used to in conjunction with BackSpace to alleviate this problem.

5.3.1 Reducing Area Overhead

The area overhead of the debugging architecture is still large due mainly to the area incurred by the signature collection circuit. For our prototype, the value S_{width} is 40% of N_{mon}, which results in a signature collection circuit with an area overhead of close to 16%. The breakpoint circuit, on the other hand, now represents 5% of the overall area overhead, half of which is static, and will not grow with N_{mon}. To improve area overhead, the main focus must be on

reducing the size of S_{width}.

One possible way of doing this is to use information about the application running on the chip to constrain the pre-image computation performed by BackSpace. If BackSpace was able to gain more information about the possible predecessor states using this method, less information would be required from the on-chip signature, which would allow for a smaller S_{width}.

Another possibility is to increase the depth of the signature collection circuit and reduce S_{width}, in the hopes that BackSpace can derive as information from a larger number of smaller signatures. Even if the number of bits in memory stays the same, a reduction in S_{width} leads to a reduction in the area of the sense amps used by SRAM memory, which is a major area contributor.

5.3.2 Automating Breakpoint Bit Selection

The current method of choosing breakpoint bits relies on experimental data. There are two ways of gathering this data. The first is by having an FPGA prototype with sources of non-determinism. The second, and more realistic in a commercial environment, is by having a cycle accurate simulator with realistic injected sources of non-determinism. As an alternative to choosing breakpoint bits based on experimental data, the bits could be based on structural information found in the RTL. The bits that would likely make the best breakpoint bits are those that would show long lasting changes as a result of non-determinism. Ideally, these bits could be identified automatically so that the generation of breakpoint bits could be inserted into the Computer Aided Design (CAD) flow, which would encourage adoption of the BackSpace debugging methodology.

The breakpoint architectures presented in this thesis produce a breakpoint signal based on the values of state bits for one cycle. It is possible that a smaller breakpoint circuit could be achieved by the generation of a breakpoint signal based on state bit values over multiple cycles. Additionally, it is worth examining whether matching on a function of state bits,

rather than simply the state bits themselves, would result in a smaller breakpoint circuit.

5.3.3 Studying and Coping with Non-Determinism

As presented in this thesis, the debugging architecture is capable of coping with non-determinism at the expense of debugging time. Chapter 2 explored how to create state accurate breakpoints given sources of non-determinism. The solution was to differentiate spurious runs from correct runs so that the spurious runs could be ignored. In a complex I.C with a long run leading to a crash state, it may become infeasible to repeatedly re-run the chip until a correct run occurs. The number of runs between correct runs may be too great.

It would be useful to be able to extract information from spurious runs rather than re-running until a correct run occurs. It is unclear, however, when information in spurious runs could be safely used without undermining the integrity of the state trace. Another method that could be used to limit the severity of this problem is to segment the design into different debugging domains. A debugging domain could be debugged independently from other domains so that sources of non-determinism outside of it could be ignored. A third method that could be used to reduce the negative impact of non-determinism is to use some of the deterministic replay ideas presented in [42] and [35] to increase the proportion of runs that are considered correct.

5.3.4 Multiple-Bug Scenarios

The methodology presented in this thesis aims to find a valid path that leads to a buggy state. It does not aim to find *all* the valid paths that lead to a buggy state. It is possible that given a multitude of paths leading to a buggy state, BackSpace finds one that is possible, but not the most likely to occur. The hope is that even if there are several paths that lead to a buggy state, the root cause of the bug will be similar between in all of these paths. In the unlikely event that completely different root causes manifest themselves in the exact same

buggy state, BackSpace will likely only catch one of these bugs.

Another possible limitation of this methodology is that one bug may block the visibility of a different bug. For example, if a bug that occurs after n cycles on a specific path crashes the chip, an undiagnosed bug may still lurk further down that path and would have been observed had the first bug not crashed the chip. The chip cannot encounter that bug until the first bug is fixed. Although BackSpace will help diagnose the first bug, the second bug will be left undetected. Work in [27] addresses this problem by with the use of on-chip reconfigurable logic to monitor and react to bugs, in the hopes of patching it either temporarily so that the next bug can be observed, or permanently to avoid having to re-manufacture the chip.

Bibliography

[1] Collett asic/ic verification study, 2004.

[2] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levina nd Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for socs. In *Design Automation Conference*, 2006.

[3] Infineon Technologies AG. Tricore 1 architecture manual, 2002. ver 1.3.3 www.infineon.com.

[4] Ehab Anis and Nicola Nicolici. Low cost debug architecture using lossy compression for silicon debug. In *Design, Automation & Test in Europe Conference and Exhibition*, 2007.

[5] Ehab Anis and Nicola Nicolici. On using lossless comrpession of debug data in embedded logic analysis. In *International Test Conference*, 2007.

[6] Bob Bentley. Validating a modern microprocessor. In *International Conference on Computer Aided Verification*, 2005.

[7] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. In *Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[8] Olivier Caty, Peter Dahlgren, and Ismet Bayraktaroglu. Microprocessor silicon debug based on failure propagation tracing. In *International Test Conference*, 2005.

[9] Intel Corp. Microprocessor quick reference guide, 2009. http://www.intel.com/pressroom/kits/quickreffam.htm.

[10] Peter Dahlgren, Paul Dickinson, and Ishwar Parulkar. Latch divergency in microprocessor failure analysis. In *International Test Conference*, pages 755–763, 2003.

[11] Marcus Erlandsson. Openrisc 1000: openrisc 1200, May 2008. `http://www.opencores.org/?do=project&who=or1k&page=openrisc1200`.

[12] Todd J. Foster, Dennis L. Lastor, and Padmaraj Singh. First silicon functional validation and debug of multicore microprocessors. *IEEE Transactions on Very Large Scale Integration Systems*, 15(5):495–504, May 2007.

[13] Farideh Golshan. Test and on-line debug capabilities of ieee std 1149.1 in ultrasparc-iii microprocessor. In *IEEE International Test Conference*, 2000.

[14] A.B.T. Hopkins and K.D. McDonald-Maier. Debug support for complex systems on-chip: A review. In *IEE Proceedings on Computers and Digital Techniques*, pages 197–207, 2006.

[15] Yu-Chin Hsu, Furshing Tsai, Wells Jong, and Ying-Tsai Chang. Visibility enhancement for silicon debug. In *Design Automation Conference*, 2006.

[16] Ho Fai Ko and Nicola. On automated trigger event generation in post-silicon validation. In *Design, Automation & Test in Europe Conference and Exhibition*, 2008.

[17] Ho Fai Ko and Nicola Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 28(2):285–297, 2009.

[18] Johnny Kuan. Technical communication, 2009.

[19] K.D. Maier. On-chip debug support for embedded systems-on-chip. In *International Symposium on Circuits and Systems*, 2003.

[20] Subhasish Mitra and Kee Sup Kim. X-compact: An efficient response compaction technique for test cost reduction. In *IEEE International Test Conference*, page 311, 2002.

[21] Kartik Mohanram and Nur A. Touba. Eliminating non-determinism during test of high-speed source synchronous differential buses. In *VLSI Test Symposium*, 2003.

[22] Flavio M. De Paula. *BackSpace: Formal Analysis for Post-Silicon Debug.* PhD thesis, University of British Columbia. (in progress).

[23] Flavio M. De Paula, Marcel Gort, Alan J. Hu, Steve J. E. Wilton, and Jin Yang. Backspace: Formal analysis for post-silicon debug. In *Formal Methods in Computer Aided Design*, pages 35–44, 2008.

[24] Flavio M. De Paula, Marcel Gort, Alan J. Hu, Steve J. E. Wilton, and Jin Yang. Backspace: Moving towards reality. In *Workshop on Microprocessor Test and Verification*, 2008.

[25] Patrick Pelgrims, Tom Tierens, and Dries Driessens. Basic custom openrisc system hardware tutorial, 2004. `http://emsys.denayer.wenk.be/empro/openrisc-HW-tutorial-Xilinx.pdf`.

[26] Vaughan R. Pratt. Anatomy of the pentium bug. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 97–107, London, UK, 1995. Springer-Verlag.

[27] Brad Quinton. *A Reconfigurable Post-Silicon Debug Infrastructure for Systems-on-a-Chip.* PhD thesis, University of British Columbia, June 2008.

[28] Bradley R. Quinton. Technical communication, 2008.

[29] Bradley R. Quinton and Steven J.E. Wilton. Concentrator access networks for programmable logic cores on socs. In *IEEE Symposium on Circuits and Systems*, pages 45–48, 2005.

[30] Bradley R. Quinton and Steven J.E. Wilton. Post-silicon debug using programmable logic cores. In *Field-Programmable Technology*, pages 241–247, 2005.

[31] Bradley R. Quinton and Steven J.E. Wilton. Programmable logic core based post-silicon debug for socs. In *IEEE Silicon Debug and Diagnosis Workshop*, 2007.

[32] Mack W. Riley, Nathan Chelstrom, Mike Genden, and Shoji Sawamura. Debug of the cell processor: Moving the lab into silicon. In *International Test Conference*, 2006.

[33] Mack W. Riley and Mike Genden. Cell broadband engine debugging for unknown events. *IEEE Design and Test*, 24(5):486–493, 2007.

[34] S. Rusu, H. Muljono, and B. Cherkauer. Itanium 2 processor 6m: higher frequency and larger l3 cache. *Micro, IEEE*, 24(2):10–18, Mar-Apr 2004.

[35] Smruti R. Sarangi, Brian Greskamp, and Josep Torrelas. Cadre: Cycle-accurate deterministic replay for hardware debugging. In *International Conference on Dependable Systems and Networks*, 2006.

[36] MIPS Technologies. Ejtag specfication, document number; md00047, revision 03.10, 2005. `http://www.mips.com`.

[37] Thomas Thatcher and Paul Hartke. Opensparc t1 on xilinx fpgas - updates, 2008. `http://ramp.eecs.berkeley.edu/Publications/OpenSPARC T1 on Xilinx FPGAs - Updates (Slides, 1-17-2008).pdf`.

[38] Bart Vermeulen and Sandeep Kumar Goel. Design for debug: Catching design errors in digital chips. *IEEE Design and Test of Computers*, 19(3):35–43, May-June 2002.

[39] Bart Vermeulen, Mohammad Z. Urfianto, and Sandeep K. Goel. Automatic generation of breakpoint hardware for silicon debug. In *Design Automation Conference*, pages 514–517, 2004.

[40] Steve J.E. Wilton. *Architecture and Algorithms for Filed-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.

[41] Steve J.E. Wilton. Heterogeneous technology mapping for area reduction in fpgas with embedded memory arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):56–68, January 2000.

[42] Min Xu, Rastislav Bodik, and Mark D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, 2003.

[43] Joon-Sung Yang and Nur A. Touba. Expanding trace buffer observation window for in-system silicon debug through selective capture. In *VLSI Test Symposium*, 2008.

[44] Yervant Zorian, Erik Jan Marinissen, and Sujit Dey. Testing embedded-core-based system chips. In *IEEE International Test Conference*, 1998.

Appendix A

Usage of Software Registers

software register	purpose
slv_reg0	min hamming weight
slv_reg1	cycle counter low
slv_reg2	cycle counter high
slv_reg3	trace buffer data read low
slv_reg4	trace buffer data read high
slv_reg5	state read select low
slv_reg6	state read select high
slv_reg7	breakpoint data write low
slv_reg8	breakpoint data write high
slv_reg9	breakpoint write select low
slv_reg10	breakpoint write select high
slv_reg11	breakpoint mask data write low
slv_reg12	breakpoint mask data write high
slv_reg13	breakpoint data read low
slv_reg14	breakpoint data read high
slv_reg15	reset breakpoint
slv_reg16	state data read low
slv_reg17	state data read high
slv_reg18	breakpoint signal
slv_reg19	trace buffer read select low / breakpoint read select low
slv_reg20	trace buffer read select high / breakpoint read select high
slv_reg21	trace buffer read address
slv_reg22	cycle counter low
RAM_select	SDRAM read/write bit select
slv_reg24	Hamstate data read low
slv_reg25	Hamstate data read high
RAM_read	SDRAM read enable
RAM_write	SDRAM write enable
RAM_ack	acknowledge from SDRAM
RAM_address	SDRAM read/write address
RAM_data_write	SDRAM data to write
RAM_data_read	SDRAM data read

Table A.1: Uses of Software Registers

Appendix B

Bit Order Tables

bit index	fraction of time differing
0	0.9182
2244	0.9182
2530	0.5586
2529	0.4646
1	0.447
2243	0.447
2528	0.3557
2526	0.2661
2527	0.266
2242	0.2656
2	0.2538
3	0.1508
2241	0.1392
2523	0.1283
13	0.1084
7	0.1049
2237	0.1038
2524	0.0985
6	0.0956
2238	0.0926
2598	0.0875
4	0.0873
2251	0.0813
2240	0.0813
2496	0.0779
2597	0.0716
2250	0.0698
9	0.0678
2235	0.0678
2614	0.0675
8	0.0665
2236	0.0659
2495	0.0657
2501	0.0644
2249	0.0614
5	0.0591
2239	0.0568
2596	0.0514
2228	0.047
2233	0.0465

Table B.1: Hamming Bits

bit index	weight
0	83957.00
2530	79797.73
2529	47128.74
1	32479.76
2528	26449.06
2523	10591.72
13	9935.97
2598	7851.91
3	7594.46
2242	7239.87
6	7174.08
2251	6821.98
2526	6242.32
2	6110.50
2240	4046.18
2241	3810.02
2250	3568.40
8	3374.34
2525	3367.18
2228	3162.69
2614	2761.13
4	2586.04
9	2389.70
2248	2120.00
2249	2065.98
2233	2048.11
2229	2018.69
2227	1936.84
2496	1834.27
3001	1794.78
7	1653.18
10	1545.99
2246	1541.09
2597	1417.95
5	1394.75
2239	1318.40
2494	1271.32
2495	1141.87
2226	1137.06
2596	1081.55

Table B.2: Weighted Breakpoint Bits