HARDWARE ASSERTIONS FOR MITIGATING SINGLE-EVENT UPSETS IN FPGAS

HARDWARE ASSERTIONS FOR MITIGATING SINGLE-EVENT UPSETS IN FPGAS

BY

STEFAN DUMITRESCU, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2020) McMaster University
(Electrical and Computer Engineering) Hamilton, Ontario, Canada

TITLE: Hardware Assertions for Mitigating Single-Event Upsets in FPGAs

AUTHOR: Stefan Dumitrescu
B.Eng. (Electrical Engineering),
McMaster University, Hamilton, Canada

SUPERVISOR: Dr. Nicola Nicolici

NUMBER OF PAGES: xiv, 94

Abstract

The memory cells used in modern field programmable gate arrays (FPGAs) are highly susceptible to single event upsets (SEUs). The typical mitigation strategy in the industry is some form of hardware redundancy in the form of duplication with comparison (DWC) or triple modular redundancy (TMR). While this strategy is highly effective in masking out the effect of faults, it incurs a large hardware cost. In this thesis, we explore a different approach to hardware redundancy.

The core idea of our approach is to exploit the conflict-driven clause learning (CDCL) mechanism in modern Boolean satisfiability (SAT) solvers to provide us with invariants which can be realized as hardware checkers. Furthermore, we develop the algorithms required to select a subset of these invariants to be included as part of a checker circuit. This checker circuit then augments the original FPGA design.

We find which look-up table (LUT) memory cells are sensitive to bitflips, then we automatically generate a checker circuit consisting of hardware invariants targeted towards those faults. We aim to reach 100% coverage of sensitizable faults. After extensive experimentation, we conclude that this approach is not competitive with DWC with respect to hardware area. However, we demonstrate that many bitflips will have reduced a detection latency compared to DWC.

Acknowledgements

No one achieves success solely through their own volition. Personal success requires the culmination of the experiences of friends, peers and mentors. My academic journey has been no exception. I have been fortunate to be accompanied by experienced guides who have shown me the way through the academic jungle.

I would like to extend my gratitude to my advisor, Nicola Nicolici. He has pushed me to constantly think deeper. When ideas ran dry, he helped me to brainstorm. When results did not look good, he helped to see them in a different light. When results did look good, he challenged me to look further. Without him, this work would not have been possible.

My work environment has played a pivotal part in my experience at McMaster. I would like to thank the past and present members of the Computer Aided Design and Test group: Alex Lau, Trevor Pogue, Amin Vali, Pouyan Mehvarzy, and Karim Mahmoud. Together we have engaged in many cerebral discussions, and we have supported each other throughout the highs and lows of academic life.

I would like to acknowledge my colleagues at Synopsys Canada ULC. My managers, Bimal Patel and Kathleen Lam, have been very supportive in encouraging me to complete my research.

I would like to thank my friends and family for the support they have provided me throughout my graduate experience.

Lastly, I acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Contents

Abstract **iii**

Acknowledgements **iv**

Definitions and Abbreviations **xii**

1 Introduction **1**

1.1 FPGA Architecture 2

1.2 Computer Aided Design for FPGAs 4

1.3 Radiation Effects in FPGAs 5

1.4 FPGAs in Harsh Radiation Environments 7

2 Background **10**

2.1 Area Redundancy 10

2.2 Temporal Redundancy 13

2.3 Information Redundancy 15

2.4 Scrubbing 16

2.5 Tool Based Techniques 18

2.6 Summary 20

3 SAT Prerequisites 21

3.1 Overview of SAT 21

3.2 SAT Solvers 22

3.3 CNF Representation of a Digital Circuit 27

3.4 Summary 31

4 Hardware Assertion Selection 32

4.1 FPGA CNF Representation 34

4.2 Sensitizable Bits vs Unsensitizable Bits 35

4.3 Assertion Mining 37

4.4 Assertion Filtering 39

4.5 Assertion Ranking 47

4.6 Greedy Assertion Selection 50

4.7 Final Assertion Evaluation 55

4.8 Summary 57

5 Experimental Results 58

5.1 Benchmark Circuits 58

5.2 Coverage Metrics 59

5.3 User Arguments 60

5.4 Hardware and Software Environment 65

5.5 Assertion Filtering Performance 66

5.6 Full Flow Performance 67

5.7 Flow Runtime Measurements 77

5.8 Summary 80

6 Conclusion and Future Work 81

6.1 Future Work 82

6.2 Concluding Remarks 84

List of Figures

1.1 Example of a fictional island-based FPGA routing architecture 3

1.2 FPGA CAD flow adapted from [1] 4

1.3 Classification of radiation effects in FPGAs 7

2.1 TMR and DWC 11

2.2 Parity checking circuit with n outputs using k parity bits 12

2.3 Reduced precision redundancy (RPR) adapted from [2] 13

3.1 Implicit traversal of the search tree 23

3.2 The effect of BCP on the search tree 24

3.3 Derivation of a conflict clause 26

3.4 Applying the Tseitin transformation 28

3.5 Example of a simple hardware invariant 29

3.6 Unrolling a digital circuit 30

3.7 A hardware invariant spanning multiple timeframes 31

4.1 Integration of hardware assertion selection with a typical FPGA flow 33

4.2 Major steps comprising hardware assertion selection 34

4.3 Construction of miter circuit for detecting sensitizable/unsensitizable bits. 37

4.4 Miter CNF configuration used in [3] 38

4.5 Example of time shifted invariants 39

4.6 Example of simple LUT based circuit 42

4.7 Unrolled version of Figure 4.6 42

4.8 Clauses for Figure 4.7 43

4.9 Implication graph demonstrating the derivation of a clause with selector variables 44

4.10 Reconstruction of states in differential fault simulation 48

4.11 Example of hardware assertions sharing flip-flops 52

4.12 CNF Formulation of at least one assertion detecting a bitflip 56

5.1 Comparison of different assertion filtering methods using three ISCAS89 circuits [4] 66

5.2 Comparison of bitflip coverage using ISCAS89 [4] and ITC99 circuits [5] 67

5.3 Effect on DSIM coverage by varying the area budget using ISCAS89 [4] and ITC99 circuits [5] 72

5.4 Effect on SAT coverage by varying the area budget using ISCAS89 [4] and ITC99 circuits [5] 73

5.5 Assertion versus output divergence timing using ISCAS89 [4] and ITC99 circuits [5] 75

List of Tables

5.1 Comparison of sensitizable bits, circuit size, and final checker circuit size 70

5.2 Runtime measurement in seconds of hardware assertion selection flow using ISCAS89 [4] and ITC99 circuits [5] 77

Definitions and Abbreviations

AI	Artificial intelligence
SIL	Safety integrity level
FPGA	Field-programmable gate array
IC	Integrated circuit
ASIC	Application specific integrated circuit
CAD	Computer aided design
DFT	Design for test
MOS	Metal-oxide semiconductor
CMOS	Complementary MOS
NRE	Non-recurring engineering
LUT	Look-up table
IO	Input/output
PLL	Phase-locked loop

DSP	Digital signal processing
RAM	Random access memory
SRAM	Static RAM
CRAM	Configuration RAM
SEE	Single-event effect
SET	Single-event transient
SEL	Single-event latchup
SEU	Single-event upset
SEFI	Single-event functional interrupt
TID	Total ionizing dose
LET	Linear energy transfer
TMR	Triple-modular redundancy
DWC	Duplication with comparison
CED	Concurrent error detection
ABFT	Algorithm-based fault-tolerance
ECC	Error-correcting code
SEC-DED	Single-error correcting double-error detecting
PLB	Programmable logic block

EDA	Electronic design automation
SAT	This term is used to refer to Boolean satisfiability
CNF	Conjunctive normal form
BCP	Boolean constraint propagation
CDCL	Conflict-driven clause learning
DAG	Directed acyclic graph
PI	Primary input
PO	Primary output
PSI	Pseudoinput
PSO	Pseudooutput

Chapter 1

Introduction

We as a species have developed an unquenchable thirst for data. The need for data processing has permeated every aspect of our lives. We find it in our personal computers and our mobile devices. Furthermore, applications in space, medicine, military, nuclear reactors, and particle physics also need this performance. The harsh radiation environments of these applications imposes addition reliability requirements. Naturally, we use redundancy to fulfill these requirements. But reliability and performance can often be at odds with each other. Thus, while our electronics are evolving to match our need for processing power, we are also constantly developing strategies to ensure their reliability.

While microprocessors have guided our way into the information age, our data processing needs made us look towards more parallel architectures. We need custom digital logic. Application specific integrated circuits (ASICs) are the natural choice for high volume consumer electronics. However, these devices have high non recurring engineering (NREs) costs. For low volume applications, field programmable gate arrays (FPGAs) have emerged as an attractive solution. They offer high flexibility, lower cost than ASICs and reduced design time.

1.1 FPGA Architecture

Field-programmable gate arrays (FPGAs) are integrated circuits (ICs) that can be configured to perform a wide variety of digital logic functions. FPGAs contain look-up tables (LUTs), flip-flops, embedded memories, input/output (IO) blocks, transceivers, phase-locked loops (PLLs) and flexible arithmetic units often referred to as digital signal processing blocks (DSPs). The FPGA interconnect is a network of wires that can be configured to connect these components together to form logic networks. This combination of interconnect and components allows the FPGA to implement most digital circuits.

FPGAs from Xilinx and Intel use an island-based routing architecture. This routing can be well described as islands of logic floating in an ocean of interconnect. Other routing architectures exist, but island-based is the most common. The components are arranged in a 2-D array. The interconnect consists of horizontal and vertical tracks. Figure 1.1 shows the layout of the components and interconnect. Components are connected to vertical and horizontal tracks using connection boxes. The vertical and horizontal tracks can then be connected with switch boxes to create more complicated paths. Furthermore, there are implementation details like global tracks and different length tracks. The specific details of how connection and switch boxes are implemented as well as the lengths of different tracks are device and vendor specific.

An FPGA will contain two layers: the user layer and the configuration layer. The user layer is what implements the desired functionality. This consists of LUTs, flip-flops, DSP slices, and IO blocks. The configuration layer is the configuration memory

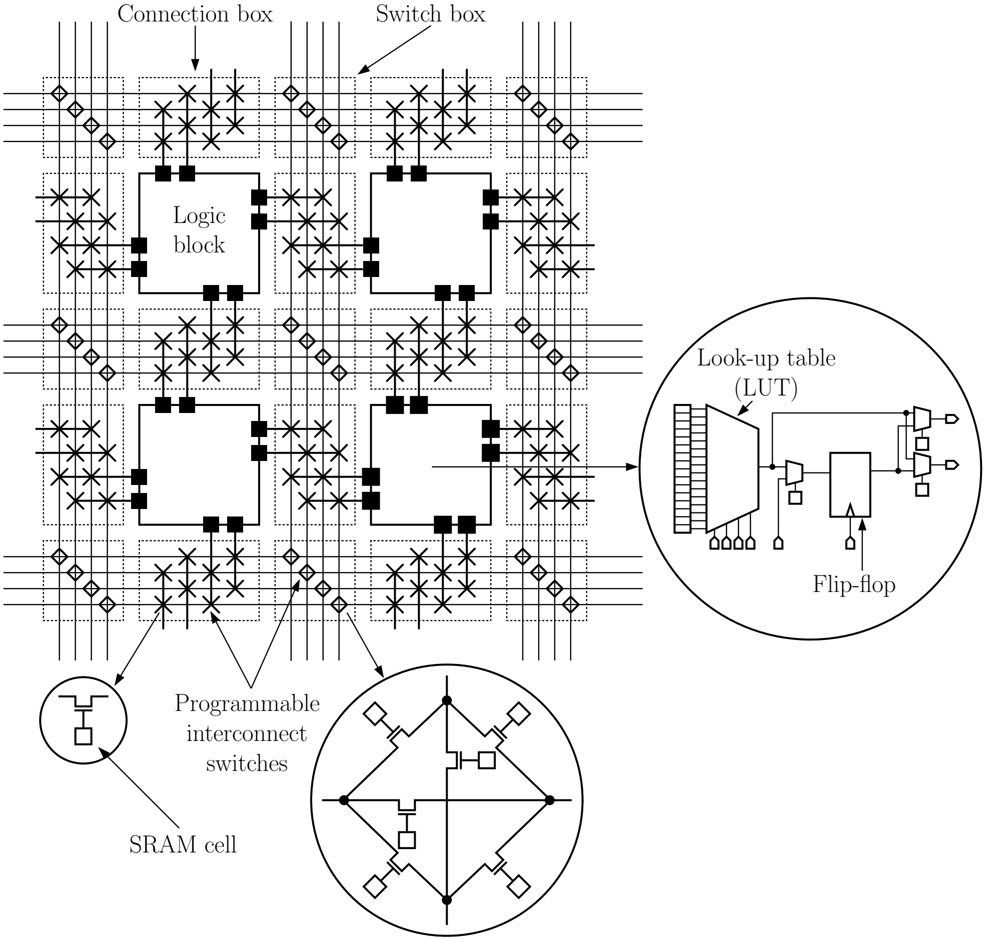


Figure 1.1: Example of a fictional island-based FPGA routing architecture

that is required to configure these components. For example, an SRAM cell could control the input multiplexer to the flip-flop of a logic block. The configuration of the switches in the interconnect is also controlled by SRAM cells. The components and interconnect are both configured through configuration RAM (CRAM).

1.2 Computer Aided Design for FPGAs

Implementing a design in an FPGA requires all programming switches to be specified. Human designers cannot feasibly manage the millions of programmable bits of FPGAs, thus computer aided design (CAD) tools were developed to map high level descriptions of circuits into usable implementations of FPGA configurations called bitstreams. The high level description of the circuit is often called register transfer level (RTL) and is often implemented in a language like Verilog. An FPGA synthesis engine compiles the RTL into a logic network consisting of components that exist in that FPGA device family. The placement step assigns each component to a physical location on the device. Finally, the route step finds an interconnect configuration that will implement the connections of the logic network.

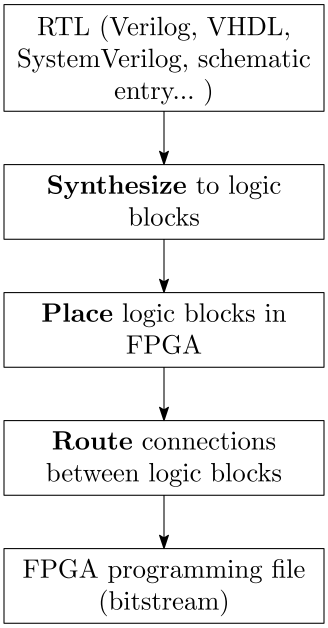

RTL (Verilog, VHDL, SystemVerilog, schematic entry...)
Synthesize to logic blocks
Place logic blocks in FPGA
Route connections between logic blocks
FPGA programming file (bitstream)

Figure 1.2: FPGA CAD flow adapted from [1]

1.3 Radiation Effects in FPGAs

At the high level, radiation effects can be categorized as dose effects or single-event effects (SEEs). Single-event effects are caused by the impact of a single particle whereas dose effects represent accumulated damage from particle strikes over some time period. Prolonged exposure to radiation will eventually cause functional failures due to leakage currents. This is known as the total ionizing dose effect (TID) [6] and it is the degradation of a complementary metal-oxide semiconductor device (CMOS) due to the accumulation of a variety of radiation effects. TID effects are of concern in any radiation environment, and thus an FPGA with an appropriate TID rating must be used.

Linear energy transfer (LET) models the energy that is deposited by a high energy particle into a specific material. LET is a function of the particle type, material type, and particle energy. When an ion with sufficient LET passes through a semiconductor device, it will leave behind a cylindrical track of electron-hole pairs. In a reverse-biased p-n junction, the reverse current is minuscule. Thus, if this strike reaches a reverse-biased p-n junction, this can cause a voltage or current spike [7]. Such an event is known as a single-event transient (SET). This transient may then propagate throughout the logic of the circuit or through the clock network to affect a latch or memory element. The transient may also be severe enough to cause other effects.

A transient can have destructive effects as well. For example, an ion strike could activate a parasitic thyristor, placing the device into a high current state [8]. This is known as a single-event latchup (SEL).

The transient could also occur inside a memory element like an SRAM cell or a flip-flop. If the SET occurs in the bistable element of an SRAM cell, then the charge collection caused by the ion strike could cause the bit to flip [9]. This is known as a single-event upset (SEU). SRAM-based FPGAs are especially susceptible to this type of fault because they can contain millions of bits of configuration memory. This type fault is non-destructive, but it will persist until the bit is rewritten.

A single-event transient may become latched into a critical device control register. Example effects of this could be triggering a built-in self test or a device reset [10]. These effects interrupt the functionality of the device and are thus known as single-event functional interrupts (SEFIs). They may last for some amount of time or require a device reset to recover.

Multiple technologies exist for implementing the configuration memory. The CRAM may be implemented using Flash, static RAM (SRAM), or antifuse technology. Each one of these technologies has its own benefits and drawbacks. Despite SRAM-based FPGAs being very susceptible to single event upsets (SEUs), they are attractive for many reasons. For a new process technology, SRAM FPGAs are usually the easiest to manufacture. They are less susceptible to total ionizing dose effects (TID) than flash FPGAs and their reconfigurability gives them an advantage against antifuse FPGAs. Reconfigurability allows a reduction in development time, and the ability to perform updates in the field. Due to these advantages, this thesis will focus exclusively on SRAM-based FPGAs.

SEUs can occur in both user memory (flip-flops or embedded memory) or configuration memory. Since the FPGA's functionality is determined by the configuration RAM, an SEU occurring in this configuration RAM can change the intended functionality of the device. An SEU could occur in a LUT changing the function implemented by that LUT. It could occur in a routing switch, accidentally, connecting or disconnecting parts of the circuits.

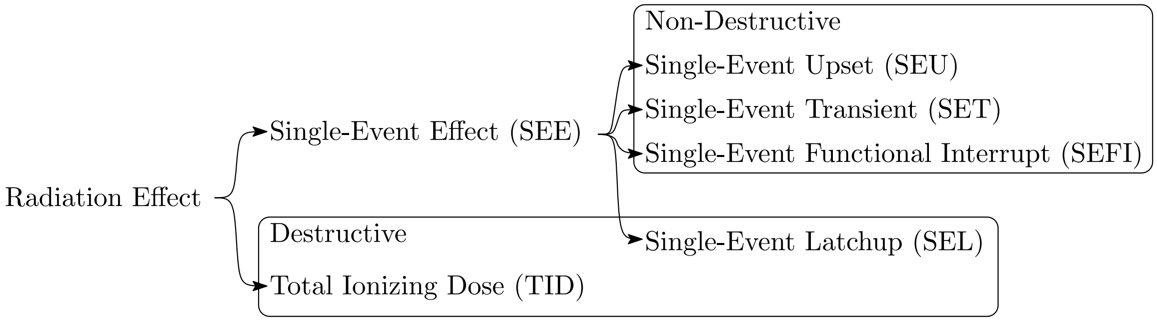

Radiation Effect
Single-Event Effect (SEE)
Non-Destructive
Single-Event Upset (SEU)
Single-Event Transient (SET)
Single-Event Functional Interrupt (SEFI)
Destructive
Single-Event Latchup (SEL)
Total Ionizing Dose (TID)

Figure 1.3: Classification of radiation effects in FPGAs

1.4 FPGAs in Harsh Radiation Environments

The use of FPGAs in satellites allows for data compression and the computation of intermediate results. It is then more beneficial to transmit these intermediate results. For example, some US missions could have payload data requirements of 1–5 terabytes raw uncompressed data [11]. However this requirement can be drastically reduced using lossless data compression and intermediate calculations. However due to the susceptibility to radiation effects, SRAM-based FPGAs are usually limited to use in non-mission critical parts of the payload like on-board signal processing and sensor data processing [12].

In terrestrial environments, susceptibility to SEEs can be mitigated through additional shielding. However, this is not always viable. In particle accelerators, particles are collided at extreme energies to produce new particles and study their properties. The readout electronics for the detectors used in the ALICE experiment are controlled by SRAM-based FPGAs [13]. Another example is the CMS detector. Radiation tests demonstrated that most of the digital components would perform reliably except for the Xilinx Virtex-6 FPGAs. These devices were shown to suffer from multiple SEUs per chip per day [14].

+

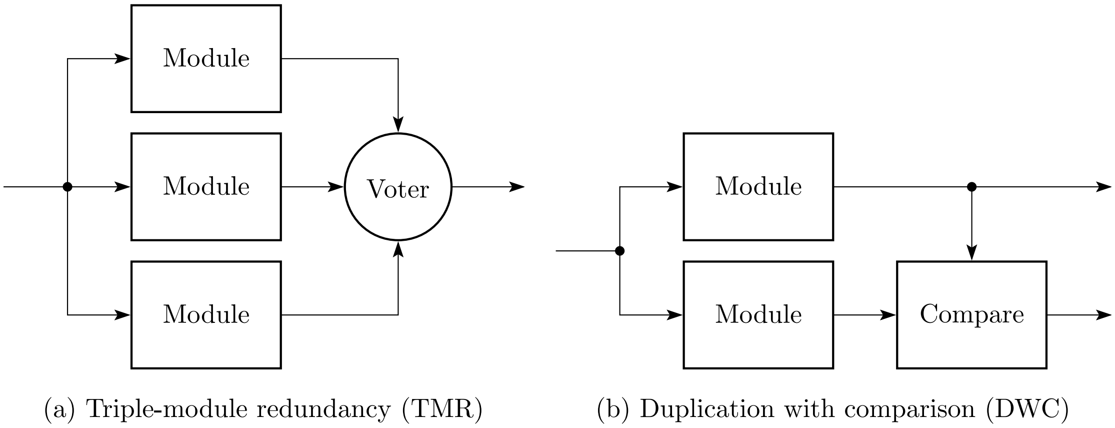
Chapter 2

Background

Given the flexibility of SRAM-based FPGAs, they are desirable even in harsh radiation environments. It is no wonder then that much research has gone into mitigating the effects of SEUs. Some methods are general fault tolerance methods for all digital circuits while others are specifically targeted towards FPGAs. The strategies are diverse, each imposing various tradeoffs in terms of area, time, memory and complexity. In this chapter, we provide an overview of the methods used to detect and mitigate SEUs in FGPAs.

2.1 Area Redundancy

The most common fault-tolerant technique is the introduction of additional copies of the logic circuit. Triple-modular redundancy (TMR) involves creating three copies of the same circuit. The outputs of the circuits can then be fed to a voter circuit [15]. In the case of a single error in any one of the three circuits, it can be masked out by the other two. Duplication with comparison (DWC) is a similar method that only requires a single copy and comparison logic. The area cost is reduced compared to TMR, but it is no longer possible to recover from a single error. Both TMR and DWC have a high area overhead but benefit from instant detection of errors.



(a) Triple-module redundancy (TMR)

(b) Duplication with comparison (DWC)

Figure 2.1: TMR and DWC

Given a logic network, not all internal nodes are equally likely to propagate errors. Samadrula et al. [16] propose to triplicate only the subcircuits which are deemed sensitive to SEUs. First the probability of each internal node is estimated, then, starting from the circuit outputs proceeding to the inputs, subcircuits are identified as chains of sensitive gates. The Boolean values of the internal nodes are evaluated using a selected probability threshold. A gate in this chain is deemed sensitive if an inversion of one of its inputs can change the output. The area overhead from triplicating only these subcircuits is about 60% - 70% of full TMR. Depending on the nature of the circuit, this form of reduced redundancy can effectively mask out most errors.

Digital circuits can be divided into persistent and non-persistent portions. The persistent portion of the circuit is the portion of the circuit in which an SEU can result in a corrupt state that can persist indefinitely. Partial TMR places emphasis on first covering the persistent components of the circuit with TMR [17]. The portion of the design in which state outputs are fed back into logic for the next state are given priority. Applying TMR here gives the highest return in protecting against persistent errors.

Although TMR is expensive in terms of area, it is well understood and straightforward to implement. Often, proposed alternatives do not actually increase the reliability and turn out to have even higher overhead. In alternatives, the mitigation circuitry can increase the SEU sensitive cross section [18].

TMR introduces at least 200% area overhead. There are other forms of spatial redundancy that use less. Instead of triplicating the functional logic, only one redundant path is added. The outputs of both the functional logic and the redundant path are fed to a checker. If the redundant path is simply a copy of the functional logic, then this is simply duplication with comparison (DWC). However, the redundant path does not have to be a copy of the functional logic. If a systematic code can be constructed from the outputs of the circuit, then the redundant path can be a parity prediction circuit that only computes the parity bits of that systematic code [19].

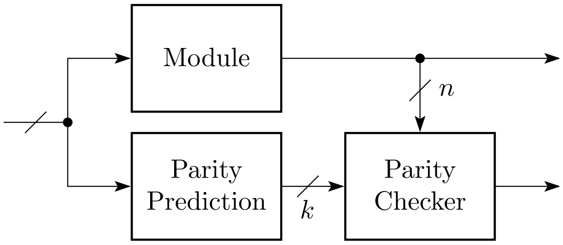
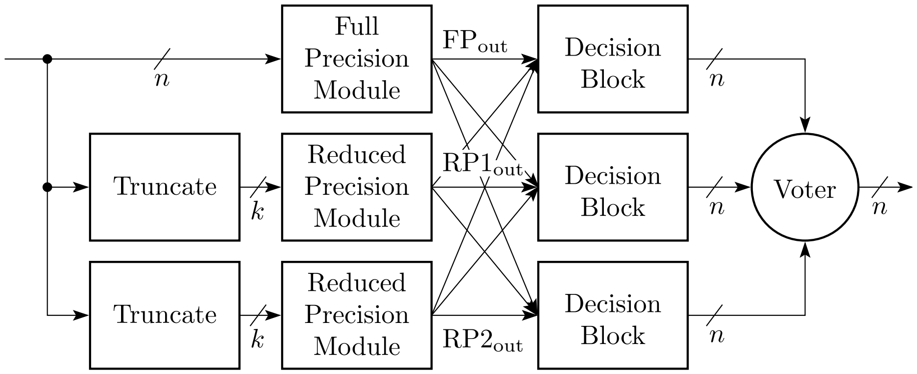


Figure 2.2: Parity checking circuit with n outputs using k parity bits

Some FPGA applications primarily require arithmetic. In these cases, a specially targeted technique known as reduced precision redundancy can be used [2]. This introduces additional copies of the modules in question, but at a reduced precision.

If there is an error in the full precision module, then the reduced precision output will be used.



```
Where a "Decision Block" implements:
if (($|FP_{out} - RP1_{out}| > T_h$) AND ($RP1_{out} = RP2_{out}$))
    output ⇐ $RP2_{out}$
else
    output ⇐ $FP_{out}$
```

Figure 2.3: Reduced precision redundancy (RPR) adapted from [2]

2.2 Temporal Redundancy

Rather than introducing additional logic, it is possible to introduce redundancy across time. By repeating calculations across multiple clock cycles, it is possible to detect SETs (Single Event Transients). However, it is unlikely an SEU would be detected using in such a manner. After the initial upset event, the fault might not immediately propagate to an observable point. A permanent fault would remain dormant until it is finally excited by an appropriate set of inputs. The temporally redundant computations would then yield the same result. Thus, simply repeating the same operation multiple times is not sufficient to detect an SEU.

If the circuit some kind of repeating structure, then it can be a candidate for temporal partitioning. For example, arithmetic circuits like adders and multipliers can be split into multiple lower precision operations [20–22]. The operation on is partitioned into multiple lower precision operations on partial words. For example, in the case of addition, each word could be split into upper bits and lower bits. In the first clock cycle, the lower bits are added. Subsequently, the upper bits are added in the second clock cycle. In each clock cycle a lower precision operation can be duplicated or triplicated.

Lima et al. [23] present a hybrid approach of temporal and area redundancy. During the normal operation of no faults or just a single fault, the circuit operates without a performance penalty. A DWC scheme continuously compares the two outputs. When a mismatch is detected, the outputs are held in registers, and additional clock cycles are spent searching for which of the two circuits contains the fault by using a concurrent error detection (CED) scheme. Assuming a decent CED scheme can be found for the block being duplicated, this method can use less area than TMR while being able to mask all of the faults.

Algorithm-based fault-tolerance (ABFT) is an approach that can apply to linear algebra based algorithms like matrix multiplication, QR decomposition and FFT [24]. For example, in matrix multiplication, the first matrix can be augmented with a row of checksums and the second matrix can be augmented with a column of checksums. The resultant matrix contains an additional row and column of checksums. Extra validation hardware in the form of an additional multiply accumulate add (MAC) unit can be added to compute an expected checksum for the resultant matrix. If an inner parallel loop hardware architecture is used for the matrix calculation, then the overhead for this additional validation hardware is very small.

2.3 Information Redundancy

A strategy for protecting the integrity of memory bits is to insert redundant memory bits. These additional bits allow for the use of error correcting codes (ECC) to detect and correct errors. ECC is widely used to protect memory in many applications: NAND flash, DVDs, CDs, hard disks, DRAM and more. The Hamming family of codes [25] is a popular strategy that allows for the correction of one bit per codeword. Extending the Hamming code with a whole word parity bit allows it to also detect double bit errors (but not correct them). These extended codes are known as single error correcting double error detecting (SEC-DED) codes.

SEC-DED codes are a popular choice for the protection of FPGA embedded memories. When the memory is accessed, the parity bits and the data can be decoded to correct single bit errors and detected double bit errors. This method is not limited to embedded memories. Configuration frames also typically store ECC bits. In Xilinx FPGAs, a specific decoder module can then be instantiated to perform error detection/correction. In the Xilinx Virtex 5 FPGA, a configuration frame is 1312 bits. When reading one of these configuration frames, a 12-bit syndrome value can be calculated. A syndrome value of 0 represents no error. If the 12th bit is 1, then the bottom 11 bits will point to the location of a single bit error in the configuration frame. Otherwise, if the 12th bit is 0, but the bottom 11 bits are not 0, then there is a double bit error [26].

SEC-DED codes are complementary to CRC checksums. While SEC-DED codes allow for single error correction, there exists the possibility of miscorrection of multiple bit errors. Thus, for a complete strategy, CRC checking should be used as well [27].

With the ongoing minimization of process technology, multiple cell upsets are becoming more prevalent. It is becoming increasingly likely that multiple adjacent bits could be upset [28–30]. This has spurred the development of various truncated Hamming codes. In the standard Hamming code, each non zero syndrome value maps to a single-bit error. A truncated Hamming code has more possible syndrome values than there are possible single bit errors. The truncated Hamming codes then map the unused syndrome values to different types of errors [31–33]. These syndrome values can be mapped to double or triple adjacent errors, or any arbitrary length adjacent errors.

2.4 Scrubbing

Scrubbing refers to the action of rewriting the configuration memory after the initial configuration has occurred. Upsets in configuration memory are repaired by overwriting them with the correct bits. It should be noted that even though methods like TMR are able to mask out single upsets, eventually the upsets will accumulate and cause a failure. Therefore a method to repair errors is required [34].

Blind scrubbing is when these writes are done without any reading of the configuration memory. The scrubber has no idea what upsets may or may not exist in the configuration memory. The configuration memory is periodically rewritten with a known good copy. Thus, radiation hardened external memory and some sort of controller is required. The effectiveness of the scrubbing increases with the scrubbing frequency. Xilinx recommends that the scrub rate should be 10 times greater than the expected upset rate. However, because the configuration memory is written periodically, many unnecessary configuration writes will occur. Thus there is a greater chance the interface to the configuration memory will be affected by a SEFI.

In order to avoid unnecessary configuration writes, readback scrubbing can be performed instead. A readback scrubbing controller will compare the contents of configuration frames with the golden copy either directly or indirectly [35]. A readback scrubber must be able to detect the presence of an error. This can be done by directly comparing against a golden copy of the configuration memory.

To reduce the amount of total memory required by the controller, each configuration frame can be augmented with a CRC checksum. The controller can then compute the checksum for the configuration frame and compare it to the golden copy instead of storing the entire bitstream. This allows the controller to also pinpoint which exact frame contains an error and correct only that frame.

If error-correcting codes (ECC) are used, then the golden copy only needs to be accessed in the presence of an uncorrectable error. Single error correcting codes can pinpoint the location of a single bit error through a syndrome value. Unfortunately, ECC also introduces the probability of miscorrection because multiple bit errors can sometimes produce the same syndrome as a single bit error.

Scrubbing can be internal or external to the FPGA. In internal scrubbing, the scrubbing controller is implemented in the FPGA. This can make the scrubber itself vulnerable to SEUs. In contrast, external scrubbers can use radiation hardened hardware. Although an internal scrubber can efficiently correct single errors using ECC, external blind scrubbers have been shown to be more reliable. Internal scrubbers encounter unrecoverable errors more often [36]. It is also possible to extract the benefits of both types of scrubbing by combining the techniques [27]. The internal scrubber can fix single bit upsets and diagnose multiple bit upsets with CRC. The external scrubber then fixes any multiple bit upsets.

Not all configuration bits are equally likely to cause a failure. One proposal is to create a scrubbing schedule that will scrub different configuration frames at different rates [37]. Bits that will be more likely to cause an error can be scrubbed more often. This can improve the mean time to failure (MTTF). However this benefit is reduced with increased configuration frame size.

2.5 Tool Based Techniques

The various steps in the FPGA design flow can be improved to take SEU sensitivity into account. The synthesis and place and route steps are natural points to hook into to provide SEU specific optimizations.

Placement can be performed with the goal of minimizing scrubbing time. If the scrubbing is performed on a frame level, then it can be beneficial to tightly pack all the sensitive CLBs into as few frames as possible [38]. Such an approach is possible by analyzing the sensitivity of each CLB and then using placement constraints to fit them into specific configuration frames.

A checkpointing strategy requires the checkpointing interval to be large enough to be able to include a scan of all relevant configuration frames. When the scrubbing time is reduced, then checkpointing with rollback can become effective at mitigating SEUs [39].

The majority of FPGA configuration bits are used for the interconnect [1] which is highly sensitive to soft errors. Zarandi et al. [40] propose to augment the typical simulated annealing algorithms used for place and route with SEU sensitivity terms. They define three types of faults that can occur: open, short, and bridging faults. Routing should avoid bridging faults at all costs as they can connect two separate domains of the circuit, even two TMR domains.

Modern FPGAs typically implement LUTs that are fracturable into smaller LUTs. The Xilinx Virtex series of FPGAs provides 6-input LUTs that can implement two 5-input functions. Das et al. [41] propose using logic restructuring and decomposition to create 5 input functions which contain mostly 0s or 1s. These 5-input functions are duplicated inside the 6-input LUT. Then by using an AND gate, 0 to 1 errors can be masked out. Similarly, using an OR gate, 1 to 0 errors can be masked out. While Lee et al. [42] propose to implement the AND/OR gates inside the fanout LUTs, Das et al. [41] propose using spare carry chain logic instead.

The authors measure the number of bits that can be masked out in this manner for every LUT. If this number falls below some threshold, then the LUT can be replaced by two new LUTs such that the fault masking is maximized, i.e. the new LUTs have a higher proportion of 0 or 1 bits. A different approach is to decompose the LUT such that the probability of a fault of that LUT reaches a primary output is minimized [43]. This decomposition can be extended to use other combining logic and not just AND/OR gates.

Another synthesis-based technique is to attempt to remap logic blocks to predefined fault tolerant programmable logic block (PLB) templates. Fault tolerant Boolean matching can improve the the percentage of input vectors for which the circuit does not produce the expected outputs by 25% without any penalty in area [44].

The in-place X-filling algorithm [45] exploits don't care bits that cannot be excited without the presence of a fault. However, if an SEU occurs, these bits might be accessed after all. Thus they should be set in the way that would most likely mask a fault appearing on a LUT input.

2.6 Summary

In this chapter, we explored the diverse strategies of SEU detection and mitigation in FPGAs. The strategies discussed are not necessarily mutually exclusive. Indeed, it is common to combine strategies because no single strategy is bullet proof. Most modern FPGAs include CRC checking and frame ECC, so it is natural to include these in the overall error mitigation strategy.

In future chapters we will discuss our proposed SEU detection strategy which will fall under the umbrella of area redundancy. The most common method is to use TMR, but this incurs a large area overhead. Even DWC imposes a high overhead of over 100%. Parity prediction can sometimes be less costly, but creating the appropriate systematic code is a difficult problem. This motivated us to create a different flow for generating checker circuits. Our checker circuit would be based on provable properties of the functional logic. These properties can map directly to hardware assertions.

Chapter 3

SAT Prerequisites

The previous chapters provided some background on SEU mitigation in FPGAs. In this chapter, we present the prerequisites for constructing hardware assertions. Our method for generating hardware assertions relies on a byproduct of SAT solvers. Thus we will explain some of the inner workings of modern SAT solvers.

3.1 Overview of SAT

The Boolean satisfiability problem (also known as SAT) was the first problem to be proven as NP-complete [46]. Despite being NP-complete, algorithms have been developed that can solve surprisingly large instances. This has allowed SAT to be used in a wide range of applications including electronic design automation (EDA).

A Boolean formula is satisfiable if there exists an assignment of its variables such that the formula evaluates to TRUE. Otherwise, if there exists no such assignment, the formula is unsatisfiable.

Boolean formulas are typically specified in conjunctive normal form (CNF). This form consists of a conjunction of a set of clauses. A single clause is a disjunction of a set of literals. A literal is a variable or its negation. For example, if x is a variable, then both x and $\neg x$ are possible literals. In the clause $(x_1 \vee \neg x_2)$, the literals are x_1 and $\neg x_2$. The following expression represents an example of a SAT formula in conjunctive normal form: $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_2 \vee x_5 \vee \neg x_6)$.

3.2 SAT Solvers

SAT Solvers are applications that solve SAT instances. They accept input in conjunctive normal form (CNF) and output either a satisfying set of assignments to its variables if the problem is satisfiable or `UNSAT` otherwise.

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [47] is a backtracking search algorithm that lies at the heart of most modern SAT solvers. With a few modern additions to this backtracking search algorithm, modern SAT solvers can handle problems with millions of variables.

In every iteration of the DPLL algorithm, a variable is chosen. The variable is assigned to either 0 or 1. If the assignment results in having a clause that is unsatisfied, then the algorithm backtracks. If we have tried only one value, then we try the other value (i.e. $0 \rightarrow 1$ or $1 \rightarrow 0$), otherwise, we must backtrack to the previous level of the search tree. Figure 3.1 demonstrates how this process implicitly traverses a search tree. When unsatisfiable partial assignments are reached, that branch of the tree is cut off by backtracking. The leaves represent complete satisfying assignments. If a leaf is reached, the algorithm returns the associated assignment. If no leaves can be reached, the algorithm returns `UNSAT`.

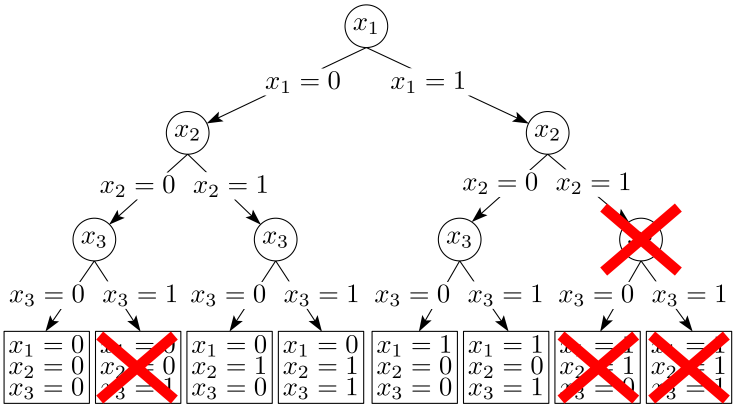


Figure 3.1: Implicit traversal of the search tree

3.2.1 Boolean Constraint Propagation

Literals in a clause can be partitioned into assigned and unassigned literals. As the algorithm progresses, more unassigned literals will become assigned literals. At some point, a clause may contain only one unassigned literal. A unit clause is a clause which contains only one unassigned literal. In a unit clause, then the remaining variable must be assigned such that it satisfies the clause.

In every iteration, the assignment might result in new unit clauses. In DPLL, these new unit clauses are used to make new assignments before a new iteration of the algorithm. This rule is referred to as the unit propagation rule and the repeated application of this rule is known as Boolean constraint propagation (BCP). Boolean constraint propagation is repeated until a conflict is reached or until all possible propagations have been performed at the current decision level. When lazy data structures containing two watched literals per clause are used, BCP proves to be very powerful and efficient for large clauses [48].

When using BCP, we call each iteration of the DPLL procedure a decision level.

Multiple variables can be assigned at a particular decision level. The first is the decision variable, then the following are the variables that are assigned due to the repeated application of the unit propagation rule. Consider the formula $(x_1 \vee x_2) \wedge (x_4 \vee x_3 \vee x_5) \wedge (x_1 \vee x_4 \vee \neg x_5)$. Figure 3.2 demonstrates the search tree corresponding to this particular formula. In this search tree, variables x_2 and x_5 are assigned due to BCP. Then, a conflict is detected at $d = 2$, so x_4 and x_5 have to be unassigned and then the other branch ($x_4 = 1$) is explored. The conflict occurs because the assignment of $x_5 = 1$ causes $\neg x_5 = 0$ and $C_3 = 0$, thus that particular partial assignment cannot satisfy all clauses.

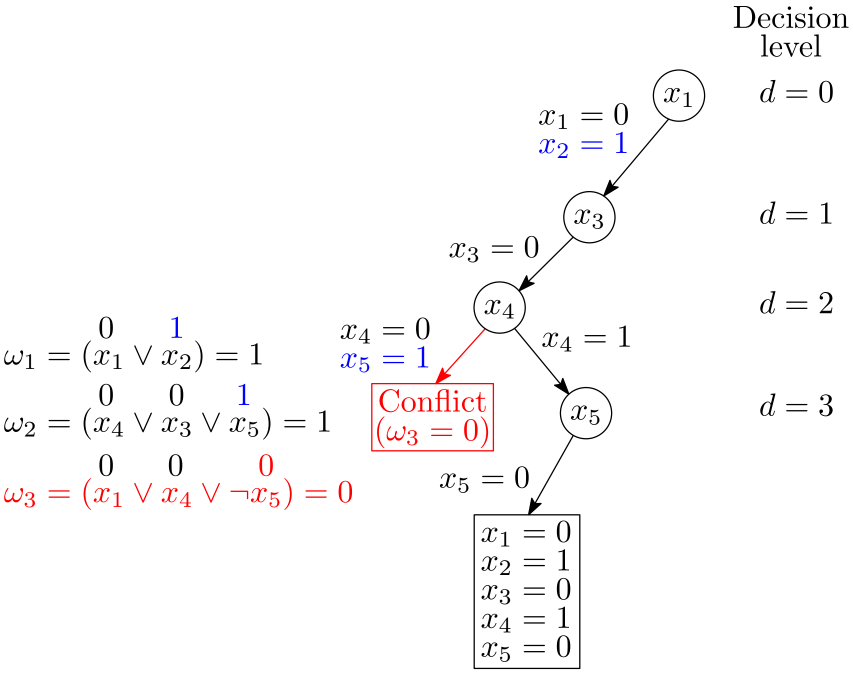

Decision level
$d=0$
$d=1$
$d=2$
$d=3$
x_1
$x_1=0$
$x_2=1$
x_3
$x_3=0$
x_4
$x_4=0$
$x_5=1$
$x_4=1$
Conflict ($\omega_3=0$)
x_5
$x_5=0$
$x_1=0$
$x_2=1$
$x_3=0$
$x_4=1$
$x_5=0$
$\omega_1 = (\overset{0}{x_1} \vee \overset{1}{x_2}) = 1$
$\omega_2 = (\overset{0}{x_4} \vee \overset{0}{x_3} \vee \overset{1}{x_5}) = 1$
$\omega_3 = (\overset{0}{x_1} \vee \overset{0}{x_4} \vee \overset{0}{\neg x_5}) = 0$

Figure 3.2: The effect of BCP on the search tree

3.2.2 Conflict-Driven Clause Learning

Learned clauses are clauses that can be derived from the original SAT formula. If the SAT formula evaluates to `TRUE` then any learned clause must also evaluate to `TRUE`.

In the DPLL algorithm, when a conflict is reached, the algorithm backtracks to the previous decision level to attempt the other branch. If the other branch also does not work, it backtracks once again to a lower decision level. This kind of backtracking is called chronological backtracking because each decision level is visited by the backtrack. However, it is possible that the underlying reason for the conflict was an assignment at a much lower decision level. In this case, the algorithm would waste time searching branches that could never lead to a satisfiable assignment.

The DPLL procedure was enhanced with non-chronological backtracking by the GRASP solver [49]. In GRASP, every time a conflict occurs, the conflict is analyzed to find the underlying assignments that caused the conflict to occur. These underlying assignments are known as the conflict set. Then, instead of backtracking to the previous level, the algorithm can backtrack to the most recent decision variable of the conflict set.

This conflict set yields a derived conflict clause that can be appended to the original set of clauses in order to prune the search space. This conflict analysis lies at the heart of most modern SAT solvers and is called conflict-driven clause learning (CDCL).

Figure 3.3 shows the derivation of a conflict clause from the analysis of the implication graph formed by the assignments and from BCP. The arrows represent assignments due to applications of the unit propagation rule, and they are labeled with the clause that implies that assignment. Cuts of the implication graph correspond to

$$\begin{aligned}
\omega_1 &= x_1 \vee x_4\\
\omega_2 &= x_1 \vee \neg x_3 \vee \neg x_8\\
\omega_3 &= x_1 \vee x_8 \vee x_{12}\\
\omega_4 &= x_2 \vee x_{11}\\
\omega_5 &= \neg x_7 \vee \neg x_3 \vee x_9\\
\omega_6 &= \neg x_7 \vee x_8 \vee \neg x_9\\
\omega_7 &= \neg x_7 \vee x_8 \vee \neg x_{10}\\
\omega_8 &= \neg x_7 \vee x_{12} \vee \neg x_{12}
\end{aligned}$$

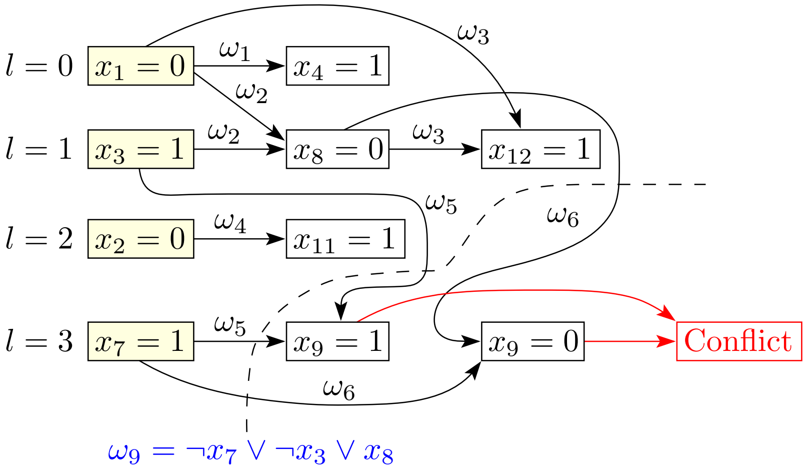


Figure 3.3: Derivation of a conflict clause

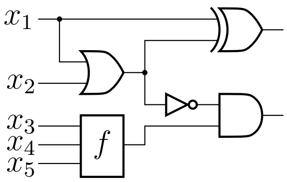
conflict clauses. This can be demonstrated by the cut corresponding to clause ω_9 in Figure 3.3. This cut identifies the nodes $x_7 = 1$, $x_3 = 1$, and $x_8 = 0$ as being sufficient to create a conflict. The clause ω_9 then encodes that these three assignments are not allowed together. In a SAT solver, usually only one cut is used. In this example, the learned clause ω_9 is added to the clause database. Then, in future explorations of the search space, clause ω_9 will constrain the search space such that the algorithm does not waste time backtracking.

This clause learning procedure generates clauses that must be true if the original formula is true. Taatizadeh and Nicolici [3] noted that the clauses learned from this procedure would map to circuit invariants if the SAT formula being solved was a CNF encoding of some digital circuit. This insight will allow us to generate a large candidate set of invariants which will be narrowed down and combined to construct a checker circuit.

3.3 CNF Representation of a Digital Circuit

In order to generate a pool of hardware invariants for a digital circuit using the CDCL mechanism, we must first encode the circuit in CNF. Larrabee [50] demonstrated how a digital circuit can be converted into a CNF by using the transformation proposed by Tseitin [51]. We can regard a digital circuit as a set of gates that form a directed acyclic graph (DAG). Then new variables are added to the output of every gate. The appropriate CNF clauses are constructed for each node of the graph and then combined to form the final CNF. This final CNF formula encodes assignments to the inputs and outputs of the circuit as well as the intermediate nodes that are consistent. When we input this formula into a SAT solver, the conflict-driven clauses will correspond to relationships of the internal circuit signals.

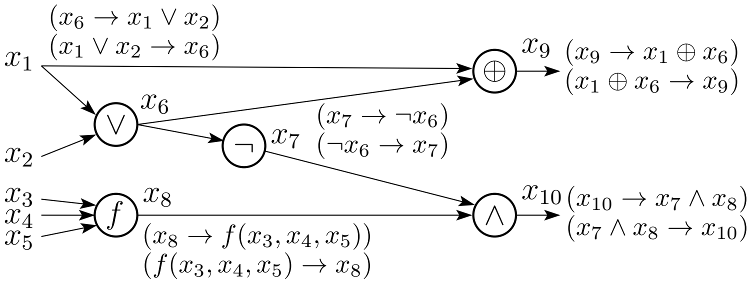
In the case of an FPGA LUT we would be able to obtain the appropriate clauses from the truth table of the LUT. Each distinct set of inputs to the LUT would have its own LUT. Figure 3.4 provides an example of how the Tseitin transformation can be applied to obtain a CNF formula. In the example, the block f represents a 3-input LUT.



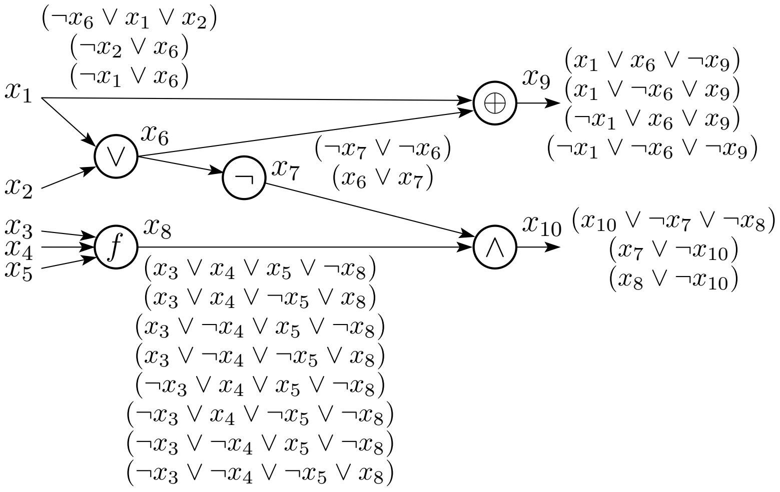
(a) Example circuit

x_3	x_4	x_5	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(b) Truth table of f



(c) Adding the intermediate variables


$(\neg x_6 \vee x_1 \vee x_2)$
$(\neg x_2 \vee x_6)$
$(\neg x_1 \vee x_6)$
x_1
x_2
$\vee$
x_6
$\oplus$
x_9
$(x_1 \vee x_6 \vee \neg x_9)$
$(x_1 \vee \neg x_6 \vee x_9)$
$(\neg x_1 \vee x_6 \vee x_9)$
$(\neg x_1 \vee \neg x_6 \vee \neg x_9)$
$\neg$
x_7
$(\neg x_7 \vee \neg x_6)$
$(x_6 \vee x_7)$
x_3
x_4
x_5
f
x_8
$(x_3 \vee x_4 \vee x_5 \vee \neg x_8)$
$(x_3 \vee x_4 \vee \neg x_5 \vee x_8)$
$(x_3 \vee \neg x_4 \vee x_5 \vee \neg x_8)$
$(x_3 \vee \neg x_4 \vee \neg x_5 \vee x_8)$
$(\neg x_3 \vee x_4 \vee x_5 \vee \neg x_8)$
$(\neg x_3 \vee x_4 \vee \neg x_5 \vee \neg x_8)$
$(\neg x_3 \vee \neg x_4 \vee x_5 \vee \neg x_8)$
$(\neg x_3 \vee \neg x_4 \vee \neg x_5 \vee x_8)$
$\wedge$
x_{10}
$(x_{10} \vee \neg x_7 \vee \neg x_8)$
$(x_7 \vee \neg x_{10})$
$(x_8 \vee \neg x_{10})$

(d) Final clauses

Figure 3.4: Applying the Tseitin transformation

3.3.1 Hardware Invariants

We have mentioned that learned clauses are invariants of a particular digital circuit. These invariants (which we also refer to as hardware assertions) can be mapped to OR gates as shown in Figure 3.5. In this example, if the output of the AND gate is 1, then the output of the OR gate is 1 as well. Since the learned clauses should also be satisfied if the original formula is satisfied, the output of the OR gate should be 1. If the output is not 1, then the CNF encoding must not represent the actual circuit. A fault must have occurred that resulted in a different CNF encoding. Thus the invariant may no longer be valid.

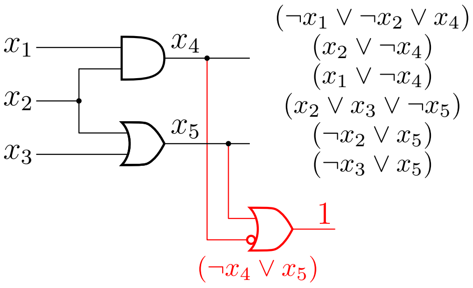
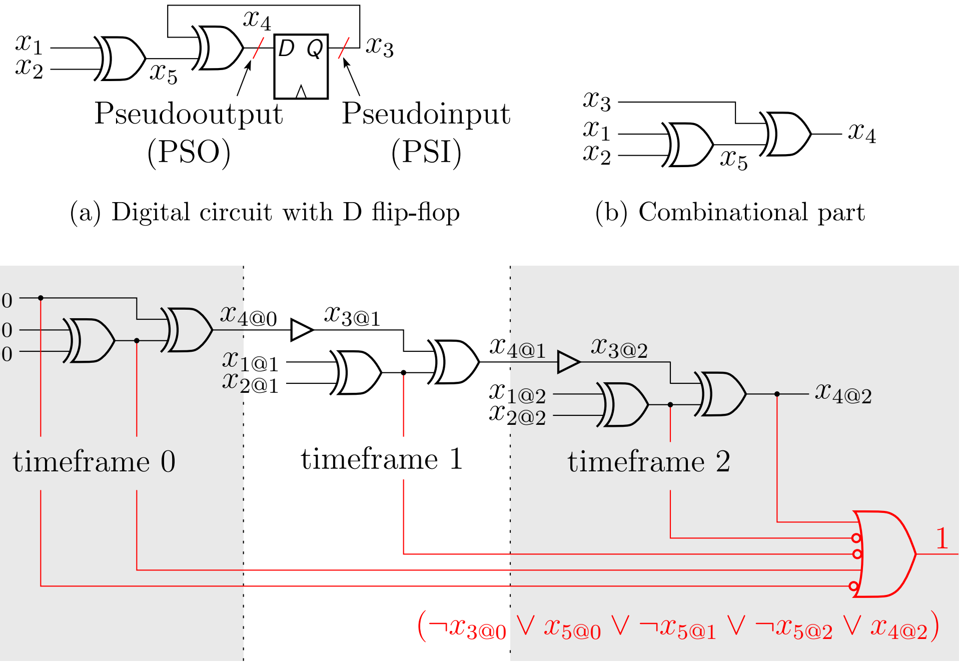


Figure 3.5: Example of a simple hardware invariant

3.3.2 Unrolling a Digital Circuit

The procedure we have mentioned can be used to generate a CNF representation for a combinational circuit. In the case of circuits with sequential elements, the inputs and outputs of the state elements must be cut, and these become pseudooutputs and pseudoinputs respectively. Performing these cuts effectively makes the circuit combinational. However, running the SAT solver would ignore interesting properties that occur solely due to the sequential nature of the circuit.

If we "unroll" the circuit across k timesteps, then we would be allowing the SAT solver to learn invariants that can only be observed after k timesteps have passed in the sequential circuit. This “unrolling” consists of cloning the cut circuit and stitching the pseudoinputs (PSIs) and pseudooutputs (PSOs) together. Figure 3.6 demonstrates time frame expansion across 3 clock cycles. It also shows an example of an invariant that is only possible due to time frame expansion. The notation $x_{i@k}$ represents a signal x_i at the kth unrolling step.



(a) Digital circuit with D flip-flop

(b) Combinational part

(c) Unrolling and stitching PSOs and PSIs

Figure 3.6: Unrolling a digital circuit

3.3.3 Hardware Invariants Across Multiple Timeframes

The hardware invariant of Figure 3.6 spans multiple timeframes. To represent this invariant with digital hardware we must use flip-flops to capture the values of the previous time frames. Figure 3.7 shows how this invariant can be added to the original circuit. The maximum timeframe of the invariant will represent the current time t, then previous time frames will be $t-1$, $t-2$, and so on. We will call the number of time frames the invariant spans the depth d of the invariant. The notation $x_{i,t-j}$ represents that this signal captures the value of x_i, j clock cycles ago.

It is worth noting that this hardware invariant may not always equal 1 now. The invariant will only be valid after $d-1$ clock cycles have passed. This is relevant when circuits are reset. When constructing checker circuits we will have to take note of the maximum depth of any invariant. The output of the checker circuit will only be valid after that amount of time has passed.

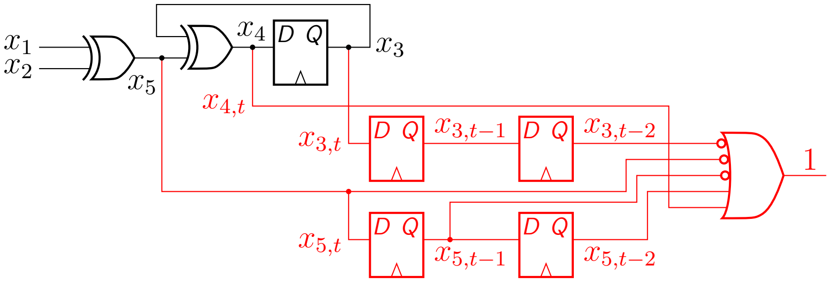


Figure 3.7: A hardware invariant spanning multiple timeframes

3.4 Summary

In this chapter, we have given a brief overview of the inner workings of a SAT solver. We have also described how the CDCL procedure of modern SAT solvers can be leveraged to generate hardware invariants. In the next chapter, we will outline the automated procedure we use to collect these invariants, and how to use them to construct checker circuitry. We will show that by modifying an extensible SAT solver, we can repurpose the conflict analysis procedure to collect a large pool of candidate invariants to repurpose as hardware assertions.

Chapter 4

Hardware Assertion Selection

In the previous chapter, we discussed the concepts used by modern SAT solvers. We showed how digital circuits can be represented in CNF format so that we can form SAT queries based on them. We showed how learned clauses can map to hardware assertions. In this chapter we will discuss our proposed flow for the construction of a checker circuit using hardware assertions that are obtained as a byproduct of a CDCL-based SAT solver. Then we will discuss how the performance of this circuit can be benchmarked.

One of the steps of the synthesis procedure is called technology mapping which maps digital logic to device specific blocks that are present in the FPGA. One of the output products of a typical FPGA compilation flow is a simulation netlist. The technology mapping can be deduced from the simulation netlist because the contents of the simulation netlist contains all of the low-level primitive modules a commercial logic simulator will need to simulate the design. The instantiated logic blocks will typically contain muxes, LUTs and flip-flops and these components are sufficient to construct a simple CNF representation of an FPGA minus the interconnect. Our flow parses the simulation netlist to construct a simplified CNF representation of the design implemented by the FPGA.

A requirement of our flow is that once a design has been technology mapped, the technology mapping must no longer change. We can only add components, not change the existing mapping. The flow that we have devised is shown at a high level in Figure 4.1. We reuse the simulation netlist and combine it with the checker circuit as the input for a recompilation. By reusing the simulation netlist, we ensure that the technology mapping of the design does not change and our selected hardware assertions will remain valid throughout the second compilation. The synthesis tool realizes that the contents of the simulation netlist are already low-level primitives, thus it will not need to do any remapping.

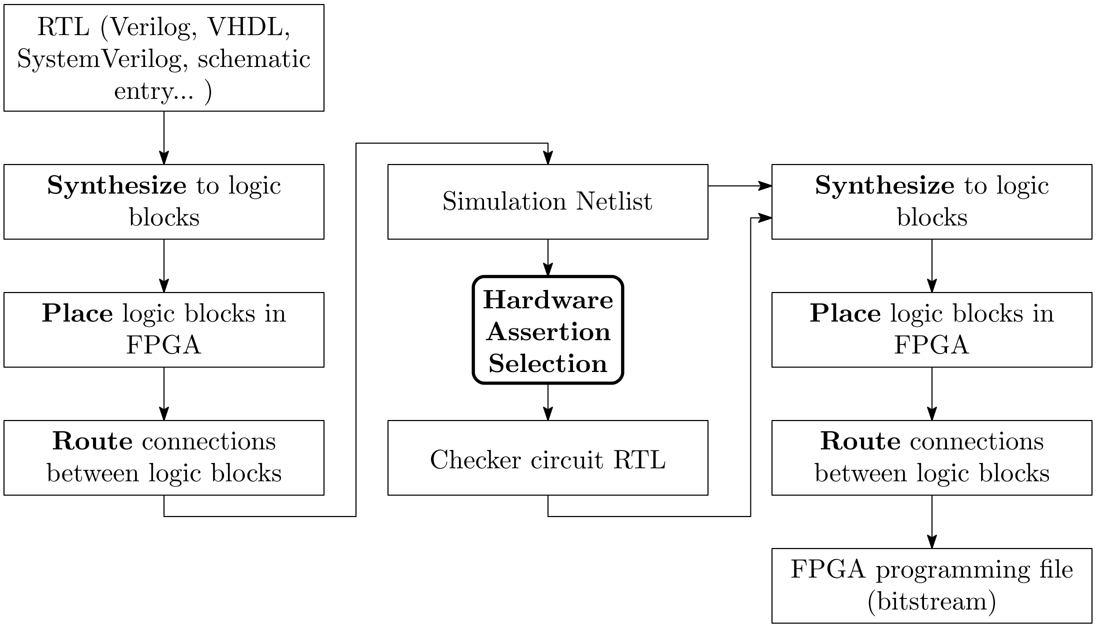

RTL (Verilog, VHDL, SystemVerilog, schematic entry...)
Synthesize to logic blocks
Place logic blocks in FPGA
Route connections between logic blocks
Simulation Netlist
Hardware Assertion Selection
Checker circuit RTL
Synthesize to logic blocks
Place logic blocks in FPGA
Route connections between logic blocks
FPGA programming file (bitstream)

Figure 4.1: Integration of hardware assertion selection with a typical FPGA flow

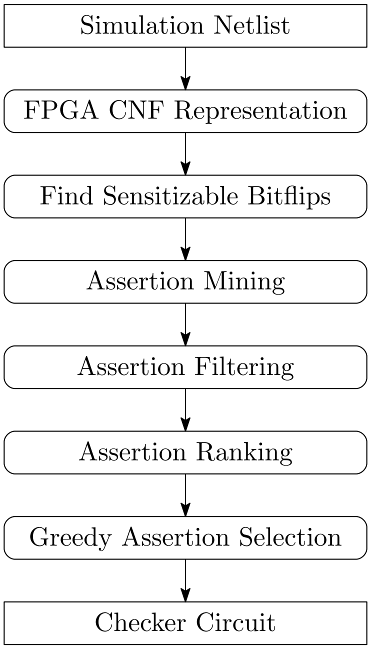


Figure 4.2: Major steps comprising hardware assertion selection

Figure 4.2 outlines the major steps of hardware assertion selection in Figure 4.1. Each of these major parts will be detailed in this chapter.

4.1 FPGA CNF Representation

Muxes, LUTs and flip-flops make up the user-level hardware in the FPGA. Unfortunately, it is not possible to deduce the configuration memory from the simulation level netlist. That information is typically locked away in the internal data structures of FPGA design tools. FPGA vendors do not reveal a detailed description of exactly what each programmable switch controls. Some of the functionality can be deduced by reverse engineering or by piecing together information from datasheets and device handbooks. However this is not enough to obtain a complete picture.

Given the lack of information regarding FPGA internals, we do not know how to exactly construct a logic network including every single reprogrammable switch. However, we do know the configuration memory that configures the LUTs. Thus we can conceptualize a logic network containing LUTs, muxes, and flip-flops, where LUTs are defined by SRAM cells.

Due to the imperfect knowledge of device specific internals, we will focus on detecting SEUs in the SRAM cells that define LUTs only. It should be noted that the hardware assertions we create could still detect SEUs occuring in SRAM cells that control the interconnect, multipliers, memories and other components. However, without being able to construct an exact CNF representation of those components, we will not be able to quantify those effects. Thus, we will restrict ourselves with using only the simplified FPGA model of LUTs, muxes, and flip-flops.

4.2 Sensitizable Bits vs Unsensitizable Bits

To judge the effectiveness of our assertions, fault simulation must be performed for all SEU sensitive bits. However, not all LUT bits are worth considering. If we can reduce the size of the set of potential bitflips we are interested in then we can reduce the amount of simulations required.

We can decide if LUT bits are sensitive by formulating the question as a SAT problem. Figure 4.3 illustrates how to construct a miter circuit to detect if a particular LUT bit can affect the outputs or pseudooutputs of a circuit. Specifically, the circuit can be unrolled for some number of clock cycles, and then a miter circuit consisting of a good half and a faulty half is inserted in the final time frame.

Due to the structure of the circuit, certain states may not be reachable. As we further unroll the circuit, more states become unreachable. There may exist LUT bits which are only sensitizable by states that are only reachable within the few clock cycles. Thus, by unrolling the circuit, we can prune some bits which are unsensitizable because the states required to sensitize them are unreachable after k clock cycles have elapsed. This also means we assume no SEUs will occur during these k clock cycles after a reset. We are discounting the LUT bits which are only sensitizable in the first few clock cycles after a reset. In Figure 4.3, some states may not be reachable within two clock cycles. We assume that no bitflips will occur within those first two clock cycles.

For every possible SEU, we construct a SAT encoding from the miter circuit with the appropriate fault inserted in the corresponding CNF clause of the faulty half. The fault is simply inserted by changing the appropriate LUT clause of the faulty half. The final timeframe of the unrolled circuit contains a good half and a faulty half, the outputs of these halves are XORed together and then ORed together. If the output of the final OR is 1, then the bitflip is detectable.

Unlike assertion mining, we solve the SAT instance without any modification of the solver. We simply want to know if the output of the miter circuit can be 1. We do this for all possible faults to create the set S of sensitizable LUT bits. All assertion ranking will then be done only considering the bits of S.

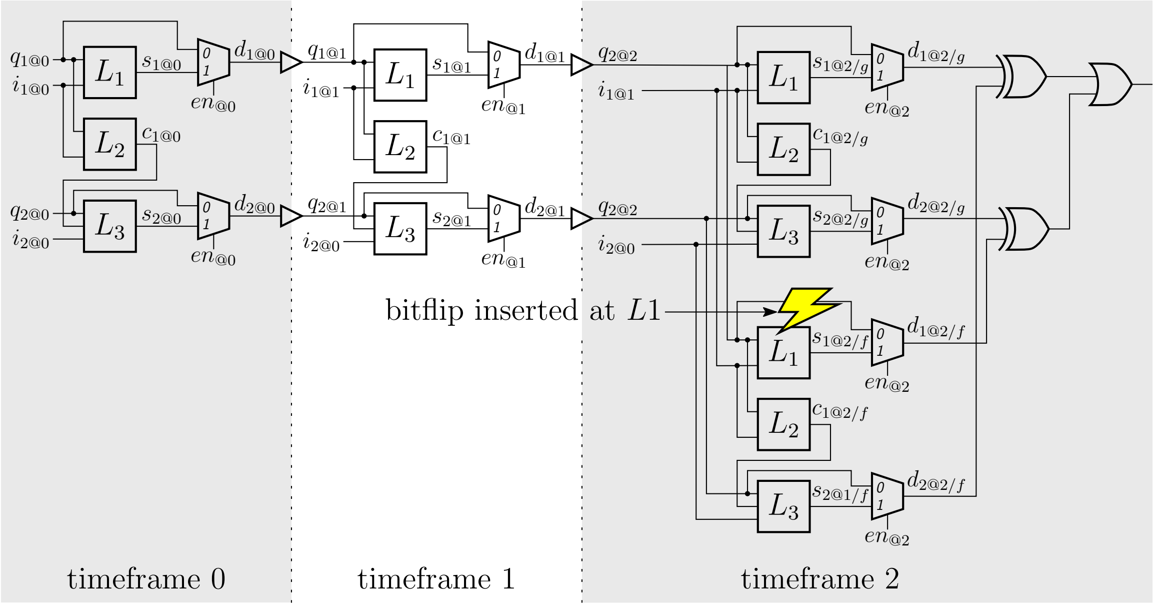


Figure 4.3: Construction of miter circuit for detecting sensitizable/unsensitizable bits.

4.3 Assertion Mining

As mentioned in the previous chapter, we can leverage the CDCL mechanism in a SAT solver to develop a large pool of invariants. One simple modification we will add to the SAT solver is to register a callback function for conflict occurences. In an iteration of the DPLL procedure of the SAT solver, if a conflict occurs, then the callback function is called. This callback function will insert the clause into a database if the clause has not been seen before. It is possible that the callback function could receive a clause that has already been learned, or a clause that is a time-shifted variant of an already learned clause. We consider these clauses equivalent and thus reject them.

For a CNF representation of a digital circuit, a satisfying assignment will quickly be found but we need the search to be prolonged in order to develop a large pool of invariants. Thus, if a satisfying assignment is found we will modify the solver to force a new descent of the search. Furthermore, rather than using variable selection heuristics to pick the variable to branch on, we tell the solver to always select a random variable to branch on and to ensure the polarity of the branching (the order in which 0 and 1 is searched) is also random. This ensures that each new search will traverse different parts of the search space.

Taatizadeh and Nicolici [3] also used a solver to mine assertions, however their approach relied on first extending the circuit by creating a logically equivalent CNF and XORing it with the CNF representing the original circuit. This allowed them to extend the runtime of the SAT solver. Some clauses will then contain literals that are not solely limited to the original CNF. This adds an additional step of having to filter out irrelevant clauses. In contrast, by simply forcing new descents in the solver, we avoid having to do this extra work.

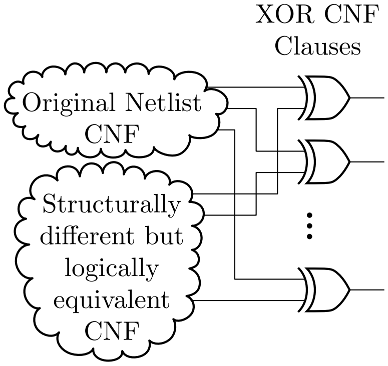


Figure 4.4: Miter CNF configuration used in [3]

Without some sort of timeout, the modified solver will run indefinitely. Thus we have elected to keep track of the time between accepted clauses. If the time exceeds some user-defined threshold or if the pool's size has reached some user-defined maximum, then we terminate the assertion mining step. These measures are necessary to restrict the memory usage and runtime of the flow.

4.3.1 Time-Shifted Invariants

As discussed in Section 3.3.2, unrolling the circuit across k timeframes can help us discover more invariants. Some of these will be invariants that span multiple time frames, others will be invariants that are only valid after more than one clock cycle. However, there will also be invariants that are time-shifted versions of other invariants. Thus before accepting an invariant we must also check that it is not a time-shifted version of an invariant we have already collected. Figure 4.5 shows an example of three invariants $\omega_{L_1}, \omega_{L_2}, \omega_{L_3}$ that are time shifted versions of each other.

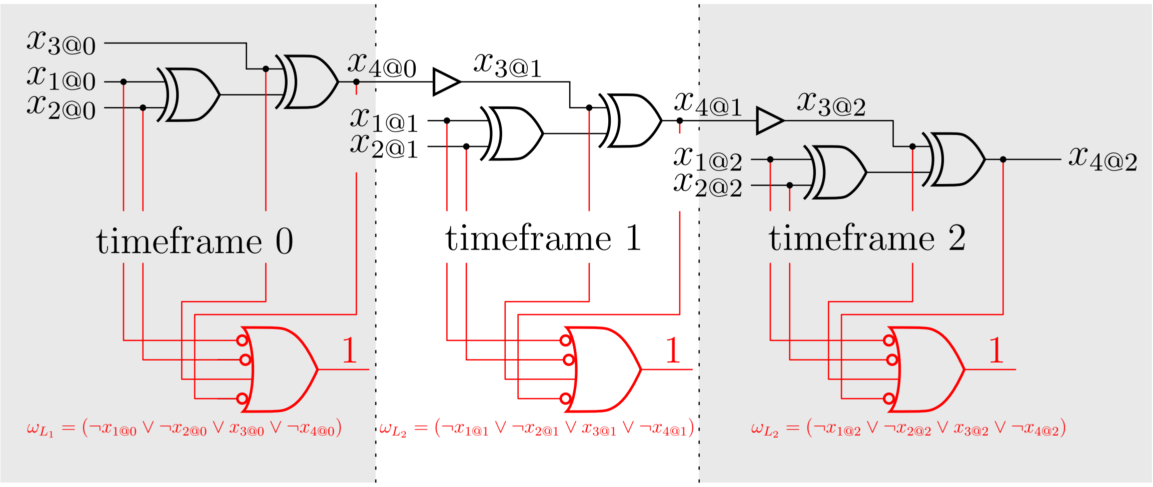

$x_{3@0}$
$x_{1@0}$
$x_{2@0}$
$x_{4@0}$
$x_{3@1}$
$x_{1@1}$
$x_{2@1}$
$x_{4@1}$
$x_{3@2}$
$x_{1@2}$
$x_{2@2}$
$x_{4@2}$
timeframe 0
timeframe 1
timeframe 2
1
1
1
$\omega_{L_1} = (\neg x_{1@0} \vee \neg x_{2@0} \vee x_{3@0} \vee \neg x_{4@0})$
$\omega_{L_2} = (\neg x_{1@1} \vee \neg x_{2@1} \vee x_{3@1} \vee \neg x_{4@1})$
$\omega_{L_2} = (\neg x_{1@2} \vee \neg x_{2@2} \vee x_{3@2} \vee \neg x_{4@2})$

Figure 4.5: Example of time shifted invariants

4.4 Assertion Filtering

Assertion filtering allows us to reduce the number of clauses that we must carry forward to the remaining steps of the flow. Thus, we must develop some meaningful heuristic that will allow us to quickly decide which clauses to keep. To do this, we introduce the concept of selector variables which is common in incremental SAT solving. Suppose we have a SAT formula $\varphi = \omega_1 \wedge \omega_2 \wedge \ldots \wedge \omega_n$. If a clause specifies the behaviour of a LUT L_j for bit k of that LUT and if that bit is sensitizable, then we will add the selector literal $L_{j,k}$ to that clause as shown in Figure 4.8. Note that we only add selector variables for the bitflips which have been proved sensitizable in Section 4.2, otherwise we do not care about them.

The function of these selector variables is to activate or deactivate the clause. The polarity of all selector literals is positive, so we can run the SAT solver with the partial assignment of 0 to all selector variables to perform the SAT query with all clauses activated. On the other hand, if any selector variable is assigned to 1, then its corresponding clause is deactivated. The clause would be satisfied from the beginning of the search, thus it would not imply any assignments to any variable. It could never appear as an intermediate node in a conflict analysis graph.

During the assertion mining procedure, we run the SAT solver with all selectors partially assigned to zero. It may seem that this accomplishes nothing because we do not deactivate any clauses. However, now the clauses we get from conflict analysis contain selector variables. In conflict analysis, every edge of the implication graph that is traveled corresponds to a clause resolution operation. However, because selector literals always have positive polarity, they can never be eliminated through clause resolution. Thus if any clauses with selectors were used in the derivation of a conflict clause, those selectors will appear in the final learned clause. Suppose that we obtained a learned clause ω_{learned} and we knew all the clauses ω_i, with $i \in I$, where I is the set of all indices i for which ω_i was used in the derivation of the clause ω_{learned}, then:

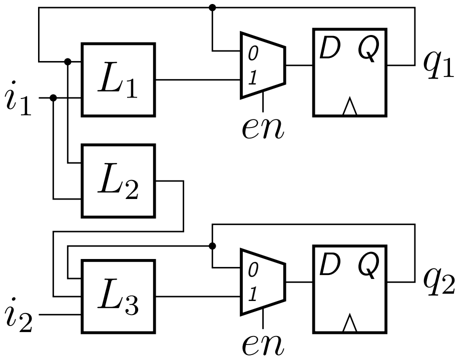
$$\bigwedge_{i \in I} \omega_i \to \omega_{\text{learned}}$$

$$\neg\omega_{\text{learned}} \to \neg \left(\bigwedge_{i \in I} \omega_i \right)$$

$$\neg\omega_{\text{learned}} \to \bigvee_{i \in I} \neg\omega_i \tag{4.4.1}$$

This tells us that if ω_{learned} was to be converted into a hardware assertion, then if its output was zero at least one of its clauses used for its derivation must also be zero. If a clause corresponding to a LUT bit was zero, then that would mean that a bitflip has occurred in that LUT. Thus we may want to prioritize clauses containing a large number of selector variables. If a selector variable is present in a clause, then because we have only added selectors to LUT bit clauses, the presence of the selector indicates the learned clause may be able to detect a bitflip in that LUT bit.

The following set of figures demonstrate how we use selector variables. Figure 4.6 is a simple 2-bit accumulator circuit constructed using LUTs, muxes and flip-flops. Figure 4.7 shows how this circuit can be unrolled across 3 clock cycles and Figure 4.8 lists the CNF encoding of this circuit. Since we have two 2-bit LUTs and one 3-bit LUT, we require 16 selector variables for LUT clauses. Each $L_{i,j}$ is the selector variable for the jth bit of the ith LUT. Figure 4.9 then shows how the conflict analysis procedure would discover a learned clause. Each enumerated cut of the implication graph adds an additional vertex to the conflict side of the cut. This corresponds to resolving the current conflict clause with the clause that labels the edge we just incorporated. All selector variables are first set to 0 such that all clauses are activated.



i_1	q_1	L_1
0	0	0
0	1	1
1	0	1
1	1	0

i_1	q_1	L_2
0	0	0
0	1	0
1	0	0
1	1	1

L_2	i_2	q_2	L_3
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 4.6: Example of simple LUT based circuit

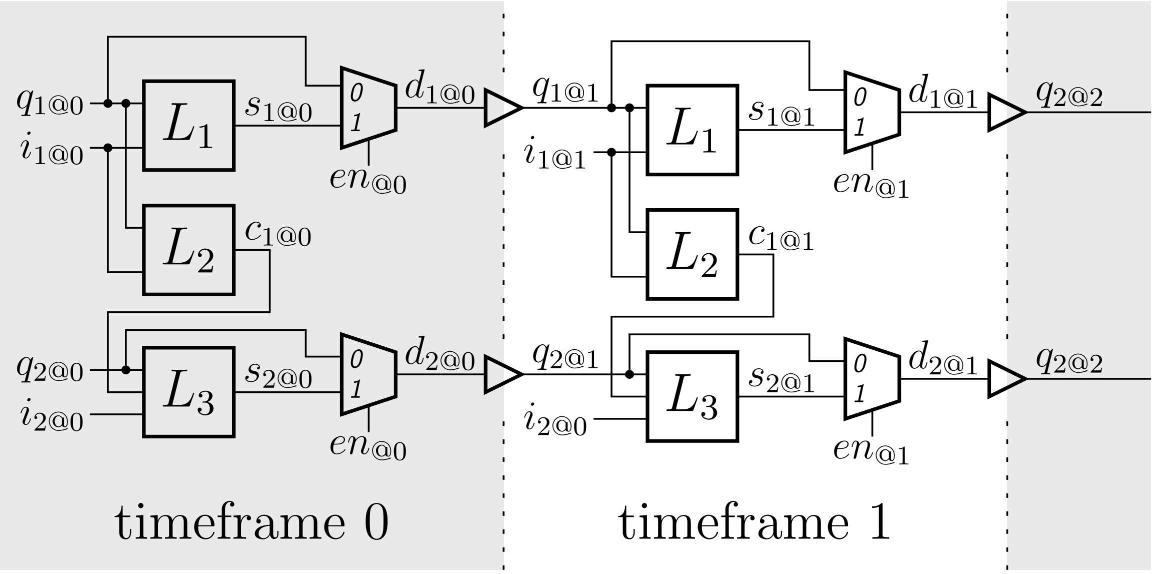


Figure 4.7: Unrolled version of Figure 4.6

$$\begin{aligned}
\omega_1 &= (L_{1,0} \vee i_{1@0} \vee q_{1@0} \vee \neg s_{1@0}) \\
\omega_2 &= (L_{1,1} \vee i_{1@0} \vee \neg q_{1@0} \vee s_{1@0}) \\
\omega_3 &= (L_{1,2} \vee \neg i_{1@0} \vee q_{1@0} \vee s_{1@0}) \\
\omega_4 &= (L_{1,3} \vee \neg i_{1@0} \vee \neg q_{1@0} \vee \neg s_{1@0})
\end{aligned}$$

$$\begin{aligned}
\omega_5 &= (L_{2,0} \vee i_{1@0} \vee q_{1@0} \vee \neg c_{1@0}) \\
\omega_6 &= (L_{2,1} \vee i_{1@0} \vee \neg q_{1@0} \vee \neg c_{1@0}) \\
\omega_7 &= (L_{2,2} \vee \neg i_{1@0} \vee q_{1@0} \vee \neg c_{1@0}) \\
\omega_8 &= (L_{2,3} \vee \neg i_{1@0} \vee \neg q_{1@0} \vee c_{1@0})
\end{aligned}$$

$$\begin{aligned}
\omega_9 &= (L_{3,0} \vee c_{1@0} \vee i_{2@0} \vee q_{2@0} \vee \neg s_{2@0}) \\
\omega_{10} &= (L_{3,1} \vee c_{1@0} \vee i_{2@0} \vee \neg q_{2@0} \vee s_{2@0}) \\
\omega_{11} &= (L_{3,2} \vee c_{1@0} \vee \neg i_{2@0} \vee q_{2@0} \vee s_{2@0}) \\
\omega_{12} &= (L_{3,3} \vee c_{1@0} \vee \neg i_{2@0} \vee \neg q_{2@0} \vee \neg s_{2@0}) \\
\omega_{13} &= (L_{3,4} \vee \neg c_{1@0} \vee i_{2@0} \vee q_{2@0} \vee s_{2@0}) \\
\omega_{14} &= (L_{3,5} \vee \neg c_{1@0} \vee i_{2@0} \vee \neg q_{2@0} \vee \neg s_{2@0}) \\
\omega_{15} &= (L_{3,6} \vee \neg c_{1@0} \vee \neg i_{2@0} \vee q_{2@0} \vee \neg s_{2@0}) \\
\omega_{16} &= (L_{3,7} \vee \neg c_{1@0} \vee \neg i_{2@0} \vee \neg q_{2@0} \vee s_{2@0})
\end{aligned}$$

$$\begin{aligned}
\omega_{17} &= (\neg en_{@0} \vee \neg s_{1@0} \vee d_{1@0}) \\
\omega_{18} &= (\neg en_{@0} \vee s_{1@0} \vee \neg d_{1@0}) \\
\omega_{19} &= (en_{@0} \vee \neg q_{1@0} \vee d_{1@0}) \\
\omega_{20} &= (en_{@0} \vee q_{1@0} \vee \neg d_{1@0})
\end{aligned}$$

$$\begin{aligned}
\omega_{21} &= (\neg en_{@0} \vee \neg s_{2@0} \vee d_{2@0}) \\
\omega_{22} &= (\neg en_{@0} \vee s_{2@0} \vee \neg d_{2@0}) \\
\omega_{23} &= (en_{@0} \vee \neg q_{2@0} \vee d_{2@0}) \\
\omega_{24} &= (en_{@0} \vee q_{2@0} \vee \neg d_{2@0})
\end{aligned}$$

$$\begin{aligned}
\omega_{25} &= (d_{1@0} \vee \neg q_{1@1}) \\
\omega_{26} &= (\neg d_{1@0} \vee q_{1@1})
\end{aligned}$$

$$\begin{aligned}
\omega_{27} &= (d_{2@0} \vee \neg q_{2@1}) \\
\omega_{28} &= (\neg d_{2@0} \vee q_{2@1})
\end{aligned}$$

$$\begin{aligned}
\omega_{29} &= (L_{1,0} \vee i_{1@1} \vee q_{1@1} \vee \neg s_{1@1}) \\
\omega_{30} &= (L_{1,1} \vee i_{1@1} \vee \neg q_{1@1} \vee s_{1@1}) \\
\omega_{31} &= (L_{1,2} \vee \neg i_{1@1} \vee q_{1@1} \vee s_{1@1}) \\
\omega_{32} &= (L_{1,3} \vee \neg i_{1@1} \vee \neg q_{1@1} \vee \neg s_{1@1})
\end{aligned}$$

$$\begin{aligned}
\omega_{33} &= (L_{2,0} \vee i_{1@1} \vee q_{1@1} \vee \neg c_{1@1}) \\
\omega_{34} &= (L_{2,1} \vee i_{1@1} \vee \neg q_{1@1} \vee \neg c_{1@1}) \\
\omega_{35} &= (L_{2,2} \vee \neg i_{1@1} \vee q_{1@1} \vee \neg c_{1@1}) \\
\omega_{36} &= (L_{2,3} \vee \neg i_{1@1} \vee \neg q_{1@1} \vee c_{1@1})
\end{aligned}$$

$$\begin{aligned}
\omega_{37} &= (L_{3,0} \vee c_{1@1} \vee i_{2@1} \vee q_{2@1} \vee \neg s_{2@1}) \\
\omega_{38} &= (L_{3,1} \vee c_{1@1} \vee i_{2@1} \vee \neg q_{2@1} \vee s_{2@1}) \\
\omega_{39} &= (L_{3,2} \vee c_{1@1} \vee \neg i_{2@1} \vee q_{2@1} \vee s_{2@1}) \\
\omega_{40} &= (L_{3,3} \vee c_{1@1} \vee \neg i_{2@1} \vee \neg q_{2@1} \vee \neg s_{2@1}) \\
\omega_{41} &= (L_{3,4} \vee \neg c_{1@1} \vee i_{2@1} \vee q_{2@1} \vee s_{2@1}) \\
\omega_{42} &= (L_{3,5} \vee \neg c_{1@1} \vee i_{2@1} \vee \neg q_{2@1} \vee \neg s_{2@1}) \\
\omega_{43} &= (L_{3,6} \vee \neg c_{1@1} \vee \neg i_{2@1} \vee q_{2@1} \vee \neg s_{2@1}) \\
\omega_{44} &= (L_{3,7} \vee \neg c_{1@1} \vee \neg i_{2@1} \vee \neg q_{2@1} \vee s_{2@1})
\end{aligned}$$

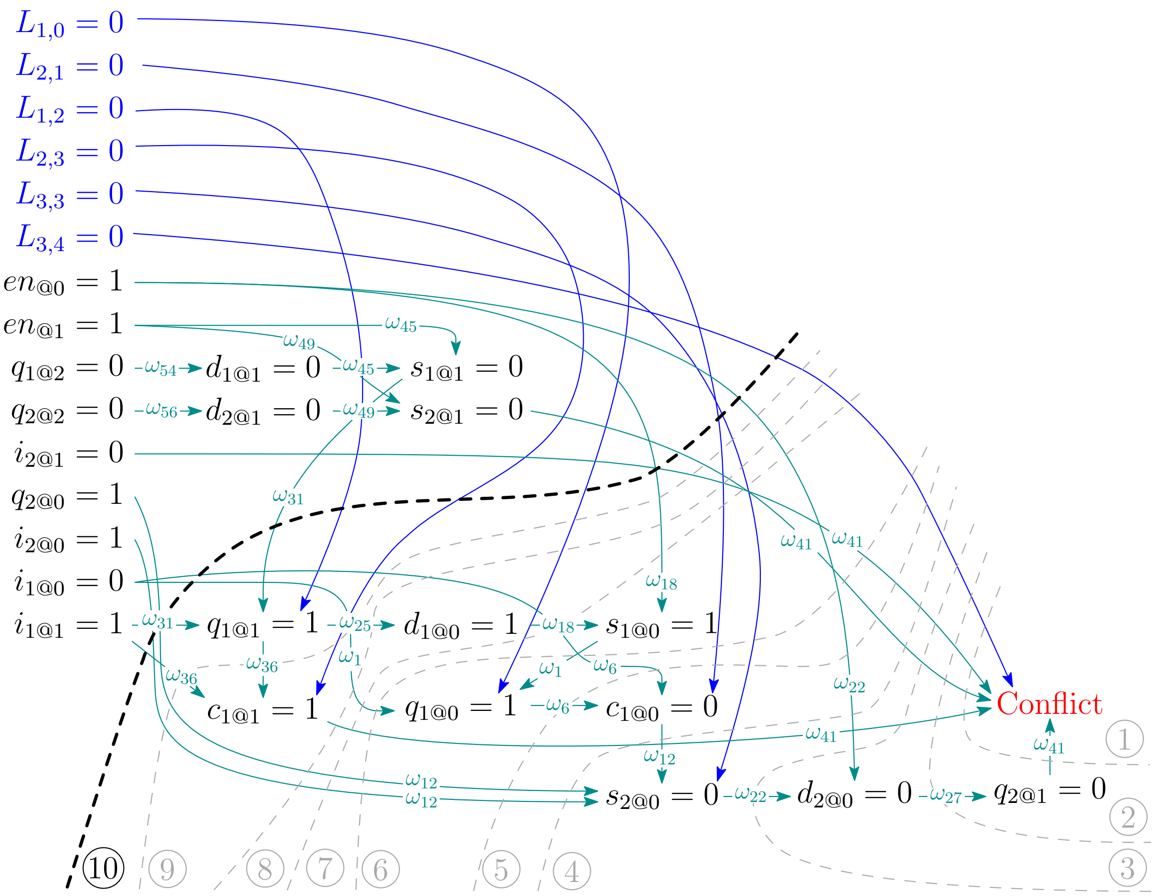
$$\begin{aligned}
\omega_{45} &= (\neg en_{@1} \vee \neg s_{1@1} \vee d_{1@1}) \\
\omega_{46} &= (\neg en_{@1} \vee s_{1@1} \vee \neg d_{1@1}) \\
\omega_{47} &= (en_{@1} \vee \neg q_{1@1} \vee d_{1@1}) \\
\omega_{48} &= (en_{@1} \vee q_{1@1} \vee \neg d_{1@1})
\end{aligned}$$

$$\begin{aligned}
\omega_{49} &= (\neg en_{@1} \vee \neg s_{2@1} \vee d_{2@1}) \\
\omega_{50} &= (\neg en_{@1} \vee s_{2@1} \vee \neg d_{2@1}) \\
\omega_{51} &= (en_{@1} \vee \neg q_{2@1} \vee d_{2@1}) \\
\omega_{52} &= (en_{@1} \vee q_{2@1} \vee \neg d_{2@1})
\end{aligned}$$

$$\begin{aligned}
\omega_{53} &= (d_{1@1} \vee \neg q_{1@2}) \\
\omega_{54} &= (\neg d_{1@1} \vee q_{1@2})
\end{aligned}$$

$$\begin{aligned}
\omega_{55} &= (d_{2@1} \vee \neg q_{2@2}) \\
\omega_{56} &= (\neg d_{2@1} \vee q_{2@2})
\end{aligned}$$

Figure 4.8: Clauses for Figure 4.7



$$\omega_{\text{learned}} = (\neg i_{1@1} \lor i_{1@0} \lor \neg i_{2@0} \lor \neg q_{2@0} \lor i_{2@1} \lor s_{2@1} \lor s_{1@1} \lor \neg en_{@0} \lor L_{1,0} \lor L_{2,1} \lor L_{1,2} \lor L_{2,3} \lor L_{3,3} \lor L_{3,4})$$

Figure 4.9: Implication graph demonstrating the derivation of a clause with selector variables

In general we will want to prioritize invariants that are small so that they will not incur a large hardware cost. But we will also want to have invariants that can potentially detect many bitflips. Thus, given that for each learned clause we also have its selectors which represent possible candidates for covered bitflips, let us define two possible filtering heuristics: the selector potential, and the selector difficulty potential.

We need to create a heuristic which attempts to maximize the number of bitflips covered while minimizing the hardware cost of the clause. The benefit of this heuristic that it is very quick to compute, then we simply need to sort the assertions by their selector potential and select the top assertions. Thus, for a clause c, we define the *selector potential* as:

$$\text{selector potential} = \frac{c.\text{selectors.size()}}{\text{AREA\_ESTIMATE}(c)} \tag{4.4.2}$$

Unfortunately, this heuristic has a problem. If we sort our assertions by selector potential, then it is possible that we will choose selectors which have a very high potential but significant overlap in the bitflips they cover. Thus we will define a procedure to select assertions by a potential which also aims to reduce bitflip coverage overlap.

Algorithm 1 shows how we can try to reduce the bitflip coverage overlap. We introduce an outer loop which will select num_remove assertions at a time. In each iteration of this outer loop we keep track of how many times a bitflip has appeared as a selector in our set of selected clauses F. Then we estimate the probability of a bitflip being covered as $\mathsf{selector\_coverage}[s]$ / F.size(). In order to place a greater emphasis on bitflips which are very unlikely to be covered, we use the negative logarithm of this ratio. Thus, we define the *selector difficulty potential* of a clause c as:

$$\text{selector difficulty potential} = \frac{\displaystyle\sum_{s \in c.\text{selectors()}} -\log\left(\frac{\mathsf{selector\_coverage}[s]}{F.\text{size()}}\right)}{\text{AREA\_ESTIMATE}(c)} \tag{4.4.3}$$

It is important to note that the $\mathsf{selector\_coverage}$ is computed using only the assertions that have been selected so far in F, we should not use learned clause database

P because in every iteration we want to place emphasis on bitflips that have not been covered so far and remove emphasis on bitflips which are already covered by multiple assertions.

Algorithm 1: Assertion filtering by selector difficulty potential

```
Input: The initial pool of learned clauses: $P$
       The number of clauses to carry forward: filtered
       The number of iterations: $N$
Data: Tallies of how many times each selector was encountered:
      selector_coverage
      The number of clauses to carry forward per iteration: num_remove
Output: The set of filtered clauses: $F$

num_remove ← filtered / $N$
foreach $1 \leq i \leq N$ do
    // Zero all selector counts and count them again based on
    // the clauses we have selected so far
    selector_coverage.clear()
    foreach clause $c \in F$ do
        foreach selector $s$ in $c$.selectors() do
            selector_coverage[$s$]++

    // Recalculate all clause scores of clauses remaining in $P$
    foreach clause $c \in P$ do
        score ← 0
        foreach selector $s$ in $c$.selectors() do
            score ← score − log(selector_coverage[$s$] / $F$.size())
        $c$.score ← score / AREA_ESTIMATE($c$)

    Sort all clauses in $P$ by score
    Remove the top num_remove clauses from $P$ and place them in $F$
return $F$
```

Both filtering methods make reference to an area estimate function. Algorithm 2 defines the algorithm used to quickly estimate the area. This is a simple area estimate that is proportional to the number of signals the assertion contains. This estimate also aims to account for flip-flops that will be needed to hold past values of signals.

We do not take into account any sharing of flip-flops that could potentially occur when multiple assertions are taken into account.

Algorithm 2: Quick assertion area estimate for assertion filtering

```
Input: An assertion $a$ with literals $l_{1@d_1}, l_{2@d_2}, \ldots, l_{n@d_n}$ where each $d_i$
       denotes the timeframe in which $l_i$ occurred
Output: The assertion cost $C$
$d_{\max} \leftarrow \max(d_1, d_2, \ldots, d_n)$
foreach $1 \leq i \leq n$ do
    $C \leftarrow C + d_{\max} - d_i + 1$
return $C$
```

4.5 Assertion Ranking

The next major step in the hardware assertion selection flow is to perform fault simulation of our circuit with all hardware assertions inserted. We must simulate every bitflip of interest to see if its effect can propagate to an assertion output. Doing this will allow us to construct a violation matrix V, where every element $v_{i,j}$ of V equals 1 if assertion a_i can detect bitflip b_j. Otherwise, element $v_{i,j}$ equals 0. We will need this matrix later to perform our final assertion selection.

4.5.1 Differential Fault Simulation

Differential fault simulation [52] is a fault simulation technique well suited to tracking faults in sequential circuits. In event-based simulation, a component is simulated only when its inputs have changed. Differential fault simulation exploits this by simulating one good circuit and multiple faulty circuits by keeping track of state differences. For a particular input vector, first the good circuit is simulated, then each faulty circuit is simulated by injecting the appropriate fault and by reconstructing the current state of the faulty circuit using the next state outputs of the previously simulated circuit.

Figure 4.10 shows how in differential fault simulation the sequential state elements of a circuit in clock cycle $i+1$ can be reconstructed from two circuits from clock cycle i. In this diagram, the notation $B_{i,k}$ represents the state of the "bad circuit" i at clock cycle k. G_i represents the state of the "good circuit", i.e. the circuit without any faults. Δ represents the difference in the pseudooutputs which are used to reconstruct the pseudoinputs in the next clock cycle.

A fault is detectable if it eventually propagates the outputs of the circuit. Thus, the outputs of the faulty circuits and the good circuit are compared at every time frame. In our case, we are also interested if faults reach assertion outputs. This means if an assertion output in a faulty circuit simulates as 0, then that assertion detects the fault corresponding to that faulty circuit.

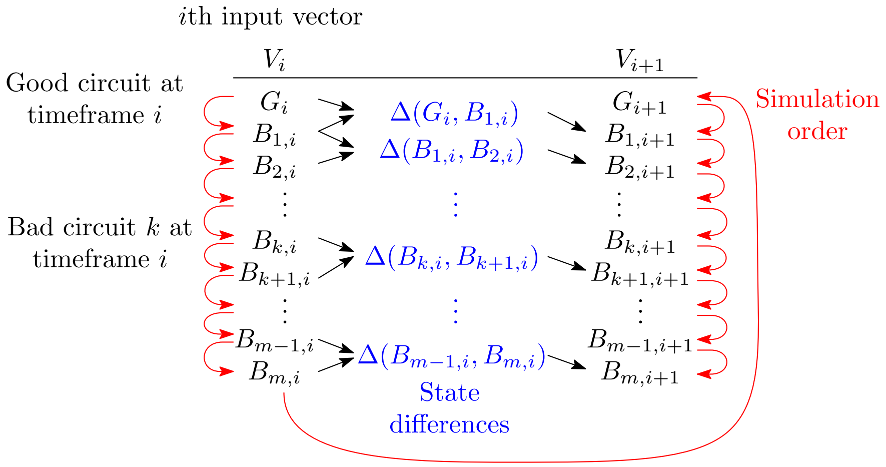


Figure 4.10: Reconstruction of states in differential fault simulation

In our proposed flow, we take the entire pool of assertions after they have been filtered, and convert them into hardware assertions that are simulated along with the logic network representing the original design. There are a few user-defined parameters including: the number of clock cycles to wait until inserting faults (this allows us to ensure all hardware assertions are valid, and ignores bits which are only sensitive in the first handful of clock cycles), the number of clock cycles to simulate, and the number of simulations to perform. During the simulation, whenever an assertion detects a fault, the corresponding entry in the violation matrix is set to 1. That entry is then proof that there exists an input sequence for which the assertion can detect the fault.

Inside the inner loop of the algorithm we inject a fault into the circuit and perform event-driven simulation. When no further events are to be processed, we look at the assertion outputs. If any assertion is 0, then we update its corresponding entry in the violation matrix to 1. Then, we store the pseudo outputs and remove the fault that has been injected in this loop iteration so we can simulate the next fault in the current clock cycle, or the “good circuit” in the next clock cycle.

Algorithm 3: Differential fault simulation for ranking assertions

```
Input: The circuit logic network with all assertions added
       The number of clock cycles: $C$
       The number of clock cycles to simulate before inserting faults: $C_{\text{ignore}}$
       The set of sensitive LUT bits: $S$
       The number of simulations: $N$
Output: The violation matrix: $V$
Initialize $V$ as all zeros
foreach $0 < n < N$ do
    Initialize the circuit status including initializing sequential state elements
      to 0
    foreach $0 < c < C$ do
        Recover current states of sequential elements
        Randomly assign inputs to 1 or 0
        Do event-driven simulation
        if $c >= C_{\text{ignore}}$ then
            foreach $s \in S$ do
                Recover current states of sequential elements
                Inject current fault $s$
                Do event-driven simulation
                foreach assertion output that is zero do
                    update $V$
                Store next state differences
                Remove the fault $s$
```

4.6 Greedy Assertion Selection

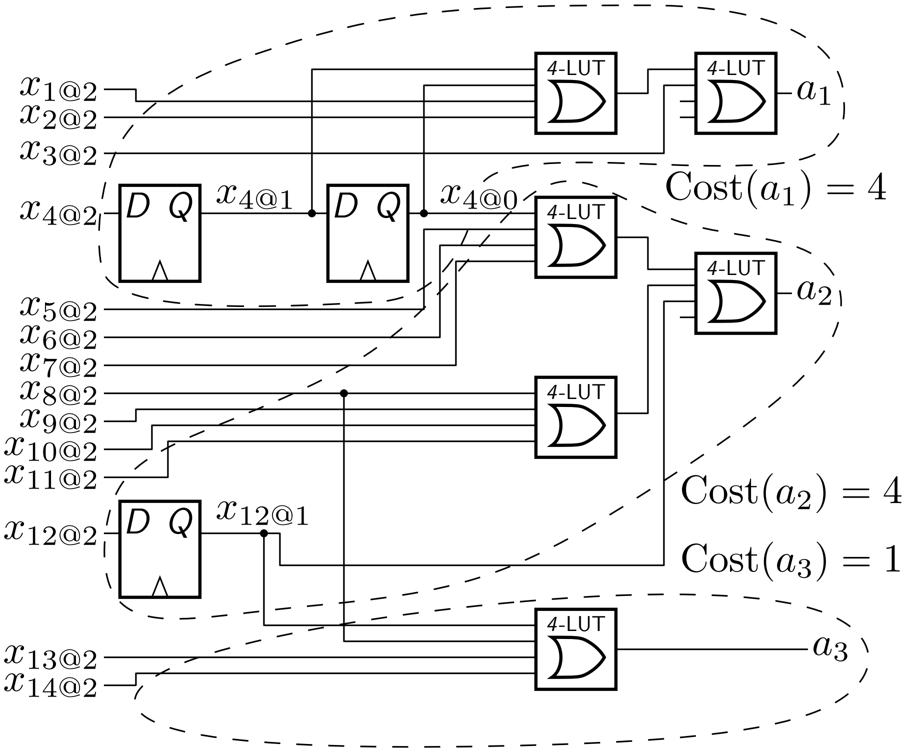
After using differential fault simulation to create a violation matrix V, we can use this violation matrix in a greedy algorithm to select the final set of assertions. In this algorithm we will keep selecting the current best assertion while we are within the area budget.

4.6.1 Assertion FPGA Area Estimation

To use our greedy selection algorithm, we will need a quick way to estimate the hardware area cost of an assertion. Given that our assertions will be implemented using FPGA logic elements we will say that using a LUT or flip-flop adds 1 to the area estimate. Thus we will need to estimate the total number of LUTs and flip-flops required.

Since an assertion is a disjunction of literals, it can be implemented using a tree of OR gates. In the case of an FPGA with k-bit LUTs, each LUT will implement a k-bit OR function. The assertion can thus be represented as a k-ary tree where each internal node represents an LUT, and each leaf represents a literal. Suppose we were to implement an assertion with n literals, where $n > 1$, then the total number of LUTs needed to implement the assertion will be $\left\lceil \frac{n-1}{k-1} \right\rceil$.

An assertion may also consist of time shifted literals of the form $x_{i@k}$. As shown in Section 3.3.3, these will require flip-flops to implement. However, once a time shifted version of a signal has been implemented in one assertion, it can be reused in another assertion. Figure 4.11 shows how flip-flops can be shared across multiple assertions. In this figure, the LUTs implement OR gates with or without inverters on the inputs. Thus, in our greedy assertion selection algorithm, when an assertion requiring flip-flops is selected, those flip-flops should only contribute to the total cost once. Algorithm 4 only counts the flip-flops once. Lines 3 – 8 calculate the maximum delays of each net captured by the assertion. Lines 8 – 16 then find the appropriate amount to add to the cost considering the cumulative flip-flops that have already been included in other assertions.



$$a_1 = (x_{4@1} \vee \neg x_{4@0} \vee x_{1@2} \vee x_{2@2} \vee \neg x_{3@2})$$

$$a_2 = (x_{4@0} \vee x_{5@2} \vee x_{6@2} \vee x_{7@2} \vee \neg x_{8@2} \vee \neg x_{9@2} \vee \neg x_{10@2} \vee \neg x_{11@2} \vee \neg x_{12@1})$$

$$a_3 = (\neg x_{12@1} \vee \neg x_{8@2} \vee x_{13@2} \vee x_{14@2})$$

Figure 4.11: Example of hardware assertions sharing flip-flops

Algorithm 4: FPGA area estimation using k-bit LUTs

```
Input: An assertion $a$ with literals $l_{1@d_1}, l_{2@d_2}, \ldots, l_{n@d_n}$ where each $d_i$
       denotes the timeframe in which $l_i$ occurred
       The associative array, cumulative_delays, representing all the
       flip-flops that have been added to implement the assertions selected
       so far
Output: The assertion cost $C$
$C \leftarrow \left\lceil \frac{n-1}{k-1} \right\rceil$
Initialize delays as an associative array
$d_{\max} \leftarrow \max(d_1, d_2, \ldots, d_n)$
foreach $1 \leq i \leq n$ do
    if delays[name($l_i$)] exists then
        delays[name($l_i$)] $\leftarrow$ max(delays[name($i$)], $d_{\max} - d_i$)
    else
        delays[name($i$)] $\leftarrow 0$
foreach (net, delay) $\in$ delays do
    if cumulative_delays[net] exists then
        if delay $>$ cumulative_delays[net] then
            $C \leftarrow C$ + delay − cumulative_delays[net]
            cumulative_delays[name($i$)] $\leftarrow$ delay]
    else
        $C \leftarrow C$ + delay
        cumulative_delays[name($i$)] $\leftarrow$ delay]
return $C$
```

4.6.2 Assertion Potential

We define the *assertion score* as the number of bitflips could detect given the violation matrix V. This is given by adding the number of ones in row i of matrix V. Then, the current best assertion is selected by a metric we call the assertion potential. The *assertion potential* is defined as the assertion's score divided by the assertion's area estimate.

$$\text{assertion score of } a_i = \sum_j v_{i,j} \tag{4.6.1}$$

$$\text{assertion potential of } a_i = \frac{\text{assertion score of } a_i}{\text{area estimate of } a_i} \tag{4.6.2}$$

The purpose of the assertion potential metric is to balance the area cost of the assertion with the amount of bitflips it can detect. We want to encourage the selection of assertions which have a small area cost but can detect a large number of bitflips.

These definitions should not be confused with Equations 4.4.2 and 4.4.3. Those define metrics that are used for assertion filtering. Assertion filtering is a heuristic method based only on the contents of the assertions. Assertion score and assertion potential are based on the results of differential fault simulation and used to inform the greedy selection algorithm.

4.6.3 Greedy Assertion Selection

Using the newly defined assertion potential, we define the greedy assertion selection algorithm. It is a simple algorithm where in every loop iteration we select the assertion with the best potential.

The objective of this algorithm is to maximize the number of bitflips covered while within the area budget. This is why the assertion potential is used as the selection criteria. However, it is possible for two assertions to have a high potential, but also a great overlap in the bitflips covered. Ideally, we would also like to minimize the overlap in bitflips covered, i.e., when selecting an assertion we want to consider its potential with respect to the bitflips that have not yet been covered. This is why Line 8 zeros the corresponding column of the violation matrix when and assertion has been selected.

Algorithm 5: Greedy assertion selection

```
   Input: The violation matrix: $V$
          The area budget: area_budget
   Output: The final set of assertions comprising the checker circuit
1  current_area $\leftarrow 0$
2  Calculate all assertion potentials
3  Sort assertions by potential
4  while current_area $<$ area_budget do
5      Select the assertion with the largest potential
6      Remove the selected assertion from the pool
7      Update current_area with area estimate of the assertion, also updating the
         cumulative delays
8      For each bitflip covered by the assertion, set the corresponding column of
         $V$ to all zeros
9      Recalculate all assertion potentials
10     Sort assertions by potential
```

4.7 Final Assertion Evaluation

4.7.1 DSIM Coverage

Given the set of assertions selected to comprise the hardware checker circuit, we will need to estimate its performance. We can once again use Algorithm 3 to do this. Instead of simulating the whole pool of assertions, we can now simulate only the final set of assertions. This will give us an idea of the percentage of sensitizable LUT bits that we can expect to detect with our assertions given random inputs. We will call this the DSIM coverage metric.

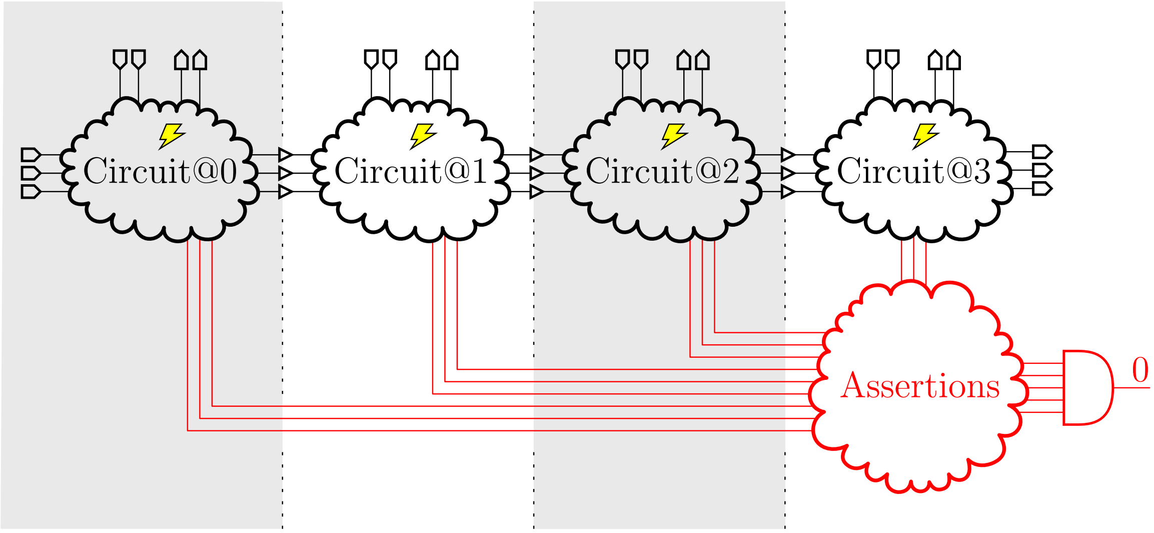


Figure 4.12: CNF Formulation of at least one assertion detecting a bitflip

4.7.2 SAT Coverage

Another question we can pose is: given this set of assertions and this bitflip present in the circuit, is there any input sequence of length k that will cause at least one assertion to be zero? Figure 4.12 shows how we can pose this question as a SAT problem. Unrolling is necessary because the clauses corresponding to the assertions will contain literals from different timeframes. The bitflip will be present in every timeframe.

All the assertions are ANDed together and we set the output of the AND gate to 0. If the SAT instance is satisfiable, then there exists an input sequence of length k for which the bitflip would be detected by the checker circuit. By performing this kind of SAT query for every sensitizable LUT bit we can find the percentage of the sensitizable bits for which a sensitizing sequence exists. We will call this the SAT coverage.

We can expect the SAT coverage to always be greater than or equal to the DSIM coverage. This is because if DSIM has shown that an assertion can detect a bitflip, then an input sequence that would satisfy the analogous SAT query could be constructed from the random inputs that were used during DSIM. However, an input sequence found using the SAT coverage method may only be a rare sequence that sensitizes the bitflip. It could be the case that the bitflip is rarely sensitized and thus it would not be detected within our DSIM experiments.

4.8 Summary

In this chapter we have laid out the steps of our proposed hardware assertion selection flow. We have shown how we can take the technology mapping of a digital circuit and create a set of hardware assertions based on it. We have used the CDCL mechanism of a SAT solver to mine a large pool of invariants. Then we simulated those invariants in the presence of bitflips proven to be sensitizable to create a coverage matrix. Finally, we ran a greedy selection algorithm to create a final set of hardware assertions.

Chapter 5

Experimental Results

In the previous chapter we proposed a flow for creating a checker circuit composed of hardware assertions. In this chapter, we will dive into the implementation details and experimental results. We will discuss the benchmarks and metrics we used to assess the performance of our hardware assertion checker circuits.

5.1 Benchmark Circuits

The ISCAS89 [4] and ITC99 [5] benchmark sets are circuits meant for researchers to develop algorithms based on circuits containing D flip-flops. These benchmarks offer a variety of circuits of varying size. The larger circuits begin to push the runtime of our algorithm to days and beyond, so these circuits are sufficient to push the boundaries of our tool. These circuits can be found online in the standard bench format. We convert these bench files into Verilog files using ABC [53]. Then we compile the verilog file in Quartus [54] to produce simulation netlist our tool expects.

5.2 Coverage Metrics

We use Algorithm 3 from the previous chapter to estimate the real world performance of our checkers. Using differential fault simulation we can track whether any assertion outputs are outputting 0 indicating the presence of a fault. However, it is possible the input sequence being used for simulation cannot cause the fault to propagate to a circuit output or pseudooutput. Thus, we also track whether the outputs or pseudooutputs diverge from the “good circuit”.

The run statistics from differential fault simulation are reported as follows:

```
simulating with cycles = 10000, wait = 5, sims per fault = 16
{output, pseudooutputs, assertion} divergence info:
 NO NPO  NA: 680
 NO NPO   A: 201
 NO  PO  NA: 36
 NO  PO   A: 79
  O NPO  NA: 0
  O NPO   A: 0
  O  PO  NA: 352
  O  PO   A: 3020

total DSIM bitflip coverage: 3300 / 4368 (75.55%)
total DSIM divergence coverage: 3099 / 3688 (84.03%)
```

The acronyms NO, NPO, NA, O, PO, A are a shorthand for the various events that occur during differential fault simulation. “O” represents the outputs diverging between a fault and the “good circuit”, likewise “PO” represents the pseudooutputs diverging. “A” represents an assertion firing. The addition of an “N” represents the negation of an event, e.g., and assertion did not fire, or outputs did not diverge.

The *total bitflip coverage* is computed from the addition of all rows containing “A”. The percentage is with respect to all bitflips inserted. This metric allows us to estimate the percentage of bitflips that could be detected after a specific number of clock cycles after bitflip injection.

$$\text{total bitflip coverage} = \frac{\text{\# of times assertions fired}}{\text{total bitflip injections}} \tag{5.2.1}$$

A second metric is the *total divergence coverage*, this second metric is motivated by the cases where no outputs diverge, no pseduooutputs diverge, and no assertions fire. Even if a bitflip is present, if no outputs or pseudooutputs diverge, then no errors have occurred yet. This metric aims to measure, when errors occur, how likely it is the errors are detected by assertions firing? This is computed similarly to the total bitflip coverage except we discount the cases when neither outputs nor pseudooutputs changed.

$$\text{total divergence coverage} = \frac{\begin{array}{c}\text{\# of times outputs or pseudooutputs}\\ \text{diverged and an assertion fired}\end{array}}{\text{\# of times outputs or pseudooutputs diverged}} \tag{5.2.2}$$

5.3 User Arguments

The hardware assertion selection flow has a number of user arguments that are passed through the command line.

5.3.1 Amount of Clock Cycles to Unroll Before Bitflip Injection (`--pre`)

During the process of finding sensitizable bitflips, this argument controls how many clock cycles to unroll the circuit as described in Section 4.2. With higher values, we prove more bitflips are unsensitizable due to unreachable states. On the other hand, higher values increase the size of the SAT instances and thus increase the runtime of SAT queries exponentially. To balance out these two effects, we keep this value at 5 for all our experiments.

5.3.2 Amount of Clock Cycles to Unroll After Bitflip Injection (`--post`)

This argument specifies how many clock cycles to unroll the circuit after bitflip injection. This is used to create the CNF representation that will be used for assertion mining and SAT ranking. This setting will also affect the differential fault simulation. If this argument takes the value k, then some of our candidate assertions may only be valid after k clock cycles. Thus, assertions are only simulated after k clock cycles have elapsed. Setting a higher value will allow us to discover assertions representing relationships spanning across multiple timeframes. On the other hand, higher values will impose additional runtime in assertion mining and assertion filtering. To balance out these two effects, we elected to keep this value at 5 for all our experiments.

5.3.3 Sensitizable Bitflip Timeout (`--timeout`)

Sensitizable bitflips are found through SAT queries as specified in Section 4.2. If the SAT query does not complete within this timeout value, then the bitflip is assumed to be sensitizable. For our experiments we maintain this value as 1 second.

5.3.4 Assertion Mining Timeout (`--learn-timeout`)

A callback function is called every time the SAT solver encounters a conflict. As specified in Section 4.3 the callback function decides to accept or reject the clause. If no new clauses have been accepted in the timespan of this timeout value, then the assertion mining step terminates. For our experiments we maintain this value as 1 second.

5.3.5 Assertion Mining Maximum (`--num-learn`)

In addition to terminating due to a timeout, the assertion mining step may also terminate when some maximum number of clauses has been reached. This argument provides the maximum number of clauses. For larger circuits, it is necessary to specify a maximum number of clauses due to computer memory constraints. For the results presented in this chapter we will use a value of 1000000.

5.3.6 Assertion Filter Method(`--filter-type`)

This argument specifies the assertion filtering method to use. We can choose between random filtering, selector potential filtering and selector difficulty potential filtering. The latter two choices have been defined in Section 4.4. Random filtering has been included as a means to verify the effectiveness of the filtering method.

5.3.7 Assertion Mining Filter (`--filter`)

This argument controls the amount of assertions to carry forward to differential fault simulation after assertion mining is complete. For the results presented in this chapter we will use a value of 100000.

5.3.8 Assertion Mining Filter Iterations (`--filter-iterations`)

When using selector difficulty potential filtering, this argument specifies the number of iterations in which assertions are selected to carry forward to differential fault simulation as described in Section 4.4. For the results presented in this chapter we have chosen to use a value of 100.

5.3.9 DSIM Cycles (`--cycles`)

This argument controls the number of clock cycles used to simulate assertions as of the assertion ranking step (Section 4.5). By increasing this number, more bitflips will be covered, and since this step acts as the feeder to the greedy assertion selection step (Section 4.6.3), the greedy selection algorithm will have a more accurate violation matrix and may be able to perform a better overall selection. However, this will increase the runtime of the differential fault simulation. To balance out these two effects, we have elected to use 1000 clock cycles as the default value.

5.3.10 Area Budget (`--area-budget`)

This argument tells the greedy selection algorithm when to stop selecting assertions to include in the checker circuit. Usually we will keep this value at 1.0. This ensures that the size of the checker circuit is about the same as the size of the original circuit. This will allow us to naturally compare the performance of this checker circuit against duplication with comparison.

5.3.11 Final DSIM Cycles (`--final-dsim-cycles`)

This argument specifies for how many clock cycles to run differential fault simulation to qualify the performance of the final checker circuit. This final simulation will be used to generate the coverage statistics that will be reported in our tables and graphs listed in this chapter. For the results presented in this chapter, we shall set this argument to 10000.

5.3.12 Number of Simulations (`--sims`)

This argument specifies the number of differential fault simulations to run for the circuit. In each simulation, every possible bitflip is injected once after k clock cycles have elapsed, where k is specified by the `--post` argument. Increasing the number of simulations will allow the coverage metrics to converge and allow the greedy selection algorithm have a more accurate violation matrix.

5.3.13 Threads (`--threads`)

This argument specifies the number of threads to use. Usually this will be kept the same as the number of simulations and we will use as many threads as the CPU can support running in parallel. If the number of simulations is larger than the supported parallel threads, then the simulations will be distributed across a thread pool containing how many threads were specified by this argument.

5.4 Hardware and Software Environment

The main PC used to run experiments was a dual Intel Xeon E5-2620 setup with 128GB of RAM. The workstation had 32 usable hardware threads and the compiler used was GCC 9.1.1. If any table or graph was generated from results originating from a different machine, then it will be noted alongside the table or graph in question.

Our proposed hardware assertion selection flow, uses the Minisat [55] based solver Glucose [56] to mine assertions and perform other SAT queries. We made one simple modification we add to Glucose to register a callback function for conflict occurrences. In an iteration of the search procedure of Glucose, if a conflict occurs, then we call the function and we must quickly decide to keep or discard the clause for our assertion database.

We have chosen to target the Cyclone IVe family of FPGAs [57] paired with Quartus 13.0sp1 [54]. These FPGAs are composed of 4 input lookup tables and flip-flop elements. Quartus can be configured to provide a simulation netlist as one of the outputs of its compilation. Our hardware assertion selection flow takes this simulation as an input.

5.5 Assertion Filtering Performance

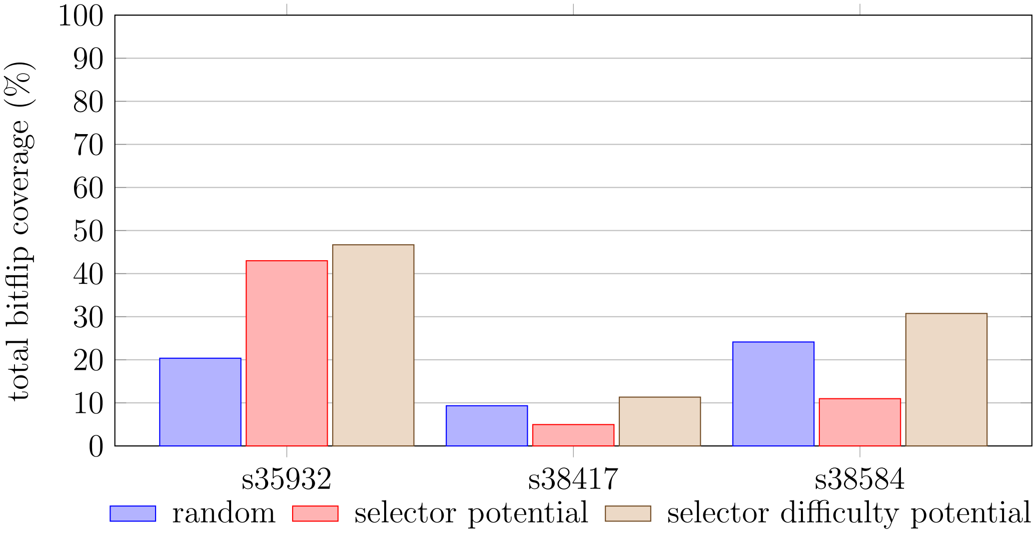


Figure 5.1: Comparison of different assertion filtering methods using three ISCAS89 circuits [4]

In Section 4.4 we discussed two different methods for assertion filtering: *selector potential* and *selector difficulty potential*. However, we discussed the possibility of bitflip coverage overlap. If the majority of assertions end up covering the same group of bitflips, then we might be better off simply selecting random assertions. Figure 5.1 compares these two methods against random assertion filtering. In order to focus on the effect of filtering, we set the (–filter) parameter to 10000 which is 10 times less than the rest of our experiments. We can see that in general, the *selector difficulty potential* filtering method performs the best. We can also see that it is possible for *selector potential* filtering to perform worse than even random filtering which could be due to the selected assertions having significant overlap in the selector variables they cover. Therefore, in subsequent results we will be using only the *selector difficulty potential* filtering method.

5.6 Full Flow Performance

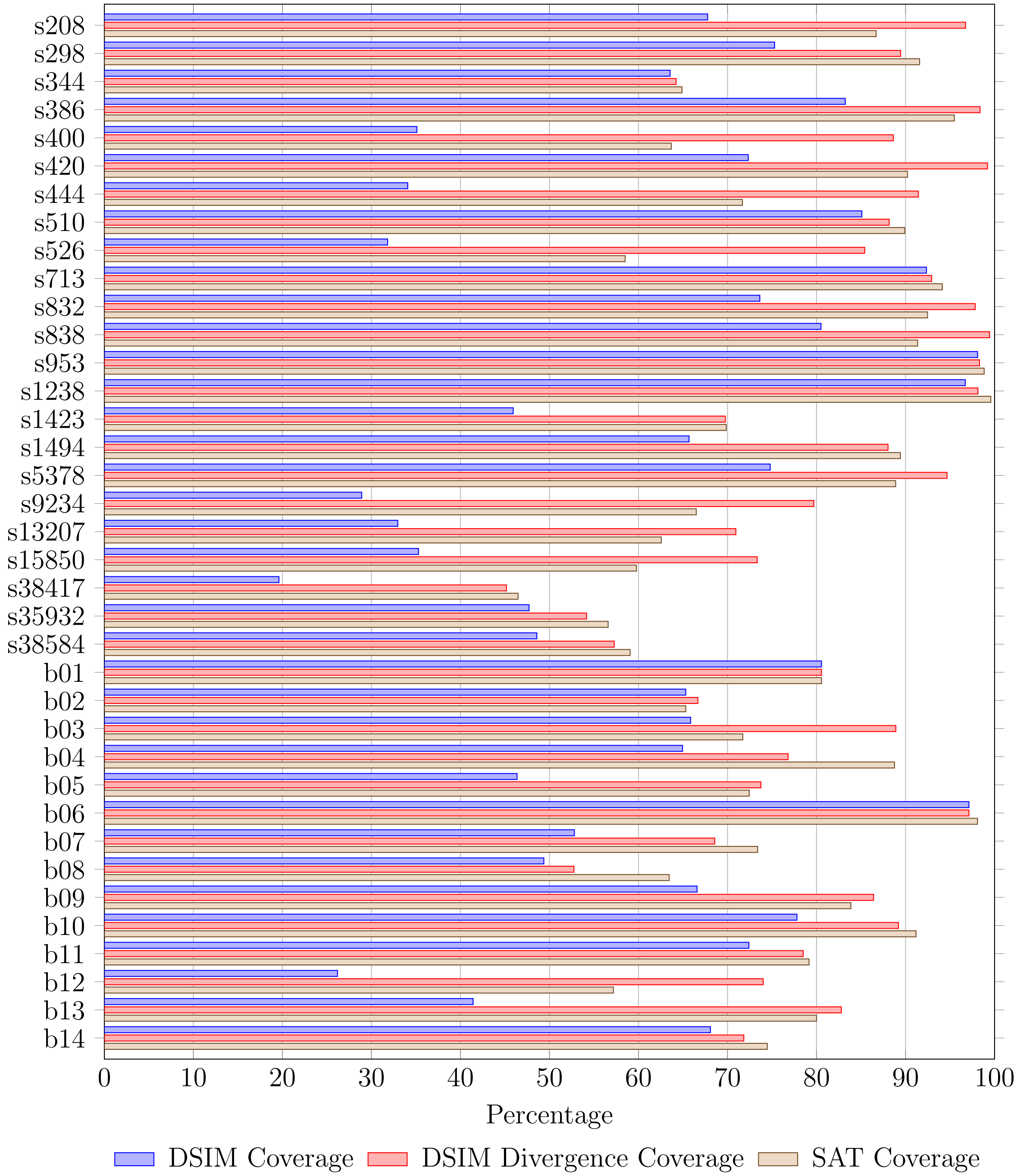


Figure 5.2: Comparison of bitflip coverage using ISCAS89 [4] and ITC99 circuits [5]

Figure 5.2 catalogs the bitflip coverage across a variety of circuits. Ideally, we would like to see all three coverage metrics place above 90% for all circuits. In order for a hardware assertions to be competitive with DWC, it should cover 100% of bitflips. However, it is evident that the performance for some circuits (e.g. s38417 and b12) is very poor. On the other hand, for circuits like s953 and s1238, all coverage metrics are close to 100%. We also observe that for some circuits, there is a big difference between the DSIM coverage, and the DSIM divergence coverage. This may be an indication that differential fault simulation was not run for long enough for bitflips to cause observable effects on circuit outputs and pseudooutputs.

One of the circuits in which we see a large difference between DSIM coverage and DSIM divergence coverage is in circuit b12. This circuit replicates the functionality of the Simon toy [58]. The player must match a sequence of increasing length. It is very unlikely that the full functionality of this circuit can be explored using purely random input vectors as is done in our differential fault simulation. It is unlikely that we could end up in a winning state purely through random simulation. Thus we speculate that the majority of possible bitflips are never being excited such that they can have an effect on the output.

Circuit b12 also features states which are controlled by counters. In these states, the progression towards the next state cannot occur until the counter reaches a certain value. This means that the length of sequences necessary to trigger certain functionality can be quite large. This phenomenon is also present in circuit b13, which appears to have a serializer FSM to communicate with some sensors. The serializer converts an 8-bit word into a serial bitstream, but waits 104 clock cycles to transmit each bit. It uses 832 clock cycles to transmit an entire word. Circuit b13 also shows a large discrepancy between the DSIM coverage and the DSIM divergence coverage, and the length of the state sequences could be one reason for this.

The performance that is revealed by the DSIM coverage data is likely to be too pessimistic. The random input vectors used by the differential fault simulation step may not accurately reflect the real-world operating environment of the circuit. For example, a benchmark circuit may contain a reset input that would normally not be asserted high. However, the random simulation would constantly toggle this input effectively restricting the states that are reachable through random simulation. On the other hand, the SAT coverage data may be too optimistic, revealing very specific requirements that would lead to a bitflip propagating to an assertion output given some input sequence. Such an input sequence could be very unlikely during the operation of the circuit, but its existence does reveal that the bitflip is detectable. Thus the real performance of the hardware assertion approach should lie somewhere in between the extremes revealed by these two metrics.

Circuit	Sensitizable Bits	Unsensitizable Bits	LUTs	Registers	Checker LUTs	Checker Registers
s208	180	188	23	8	25	3
s298	207	273	30	14	32	3
s344	228	188	26	15	30	4
s386	335	465	50	6	60	2
s400	210	526	46	21	55	4
s420	225	511	46	16	44	8
s444	210	526	46	21	54	4
s510	361	1,031	87	6	102	5
s526	231	441	42	21	48	3
s713	168	664	52	14	66	0
s832	538	1,302	115	5	147	0
s838	301	1,299	100	32	86	22
s953	930	1,710	165	29	202	1
s1238	1,118	2,146	204	18	255	3
s1423	775	1,689	154	74	155	41
s1494	1,333	2,779	257	6	227	51
s5378	3,541	3,115	416	160	514	15
s9234	1,667	2,605	267	124	253	57
s13207	4,813	5,379	637	422	522	258
s15850	5,301	7,611	807	441	687	158
s38417	15,447	23,097	2,409	1,387	1,952	283
s35932	17,681	18,127	2,238	1,472	1,827	146
s38584	13,367	25,865	2,452	1,154	2,400	232
b01	36	108	9	5	12	0
b02	15	49	4	4	2	0
b03	362	198	35	30	40	3
b04	1,656	1,032	168	66	170	10
b05	1,322	1,622	184	34	186	24
b06	56	104	10	9	11	0
b07	617	631	78	42	76	20
b08	230	394	39	21	47	4
b09	273	415	43	28	54	2
b10	346	646	62	17	75	2
b11	1,237	1,675	182	31	189	7
b12	2,565	3,963	408	119	356	33
b13	562	720	82	53	99	6
b14	8,549	17,499	1,628	215	1,616	21

Table 5.1: Comparison of sensitizable bits, circuit size, and final checker circuit size

Table 5.1 showcases the number of sensitizable bits. It also compares the original circuit size to the size of the checker circuit. The size of the checker circuit is obtained by combining all the assertions into one module and reducing the outputs of all assertions with a single OR gate. So this final checker area includes not only the hardware assertion area, but also the area needed to reduce them to a single output. This comparison reflects the accuracy of the area estimation we used for assertion selection in Section 4.6.1. We expect that DWC would also have a similar cost to original circuit, with some extra for the comparison logic. In order for our checker circuits to be competitive with DWC, they must cover 100% of bitflips while having a hardware cost that is less than or equal to DWC. Thus, we must conclude that the hardware assertion selection flow does not produce a checker circuit which can compete with DWC, even when both have a comparable area cost.

5.6.1 Area Budget Effect on Coverage

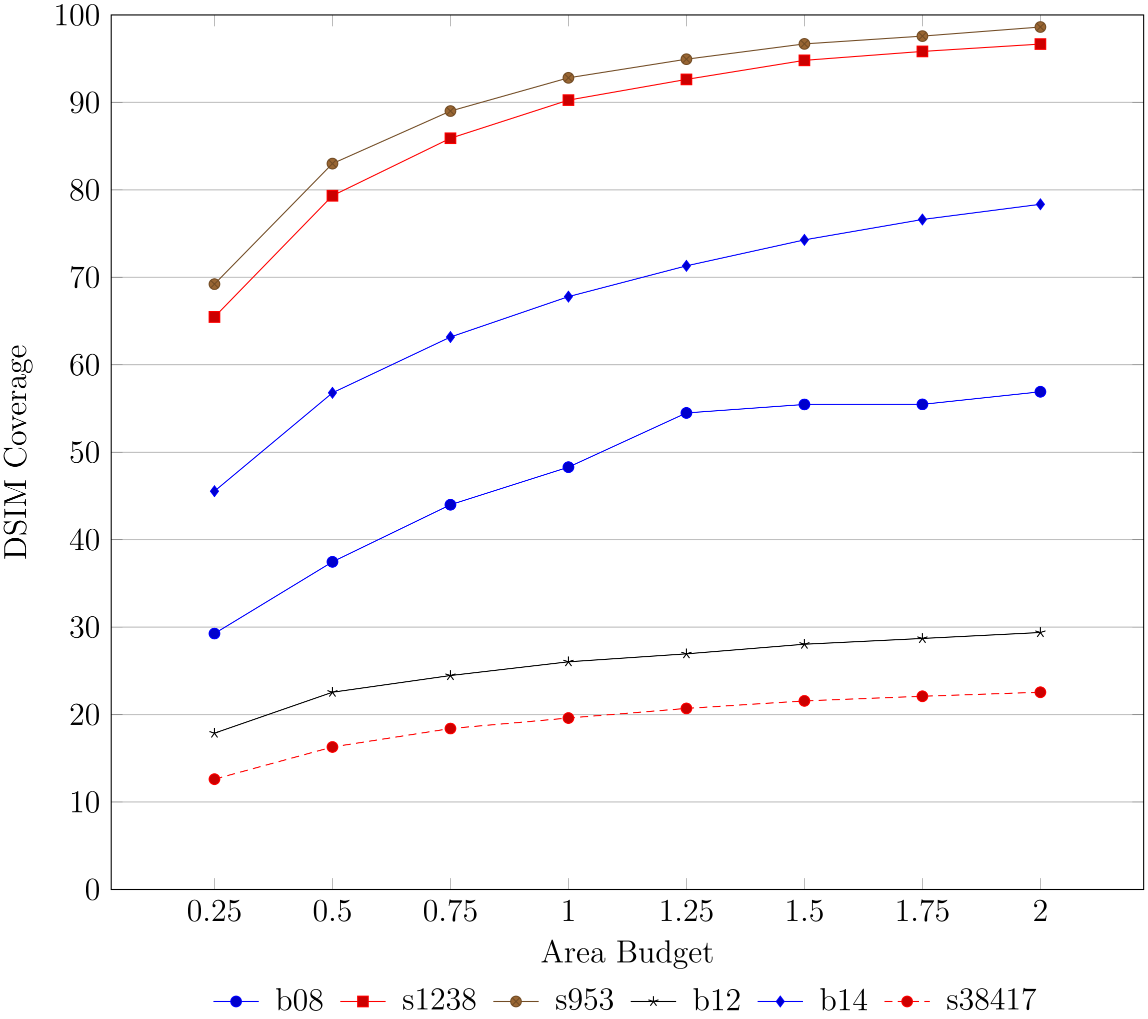


Figure 5.3: Effect on DSIM coverage by varying the area budget using ISCAS89 [4] and ITC99 circuits [5]

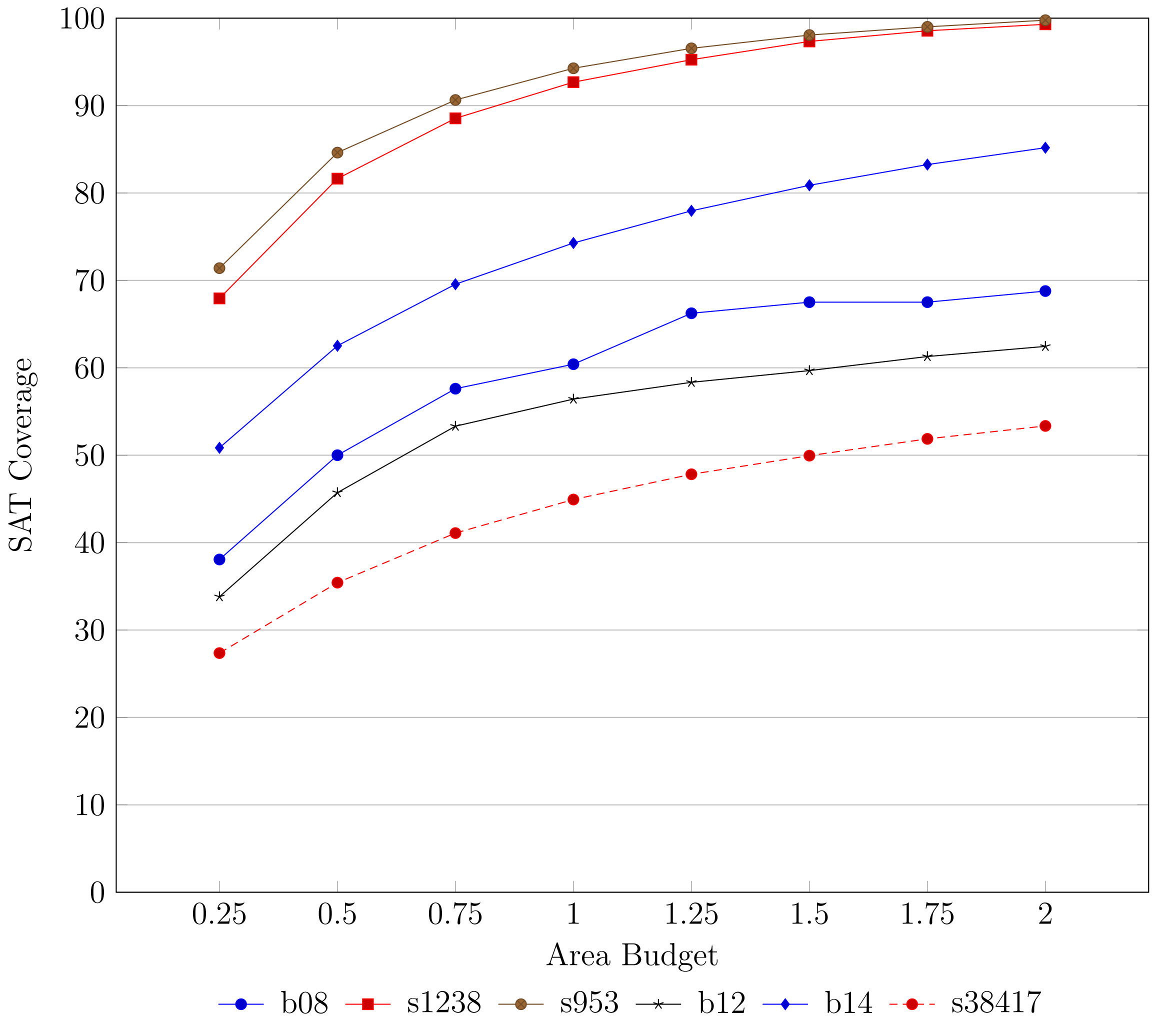


Figure 5.4: Effect on SAT coverage by varying the area budget using ISCAS89 [4] and ITC99 circuits [5]

We were also interested to see how varying the area budget can effect the performance of the checker circuit. Varying this area budget allows the greedy assertion selection algorithm to select more assertions from the assertions simulated by differential fault simulation. Figure 5.3 shows the effect of increasing the area budget on a subset of the circuits from Figure 5.2. Initially the DSIM coverage increases rapidly, but eventually we see diminishing returns. Figure 5.4 shows the effect of increasing the area budget on SAT coverage. We can observe that for each circuit, the SAT coverage provides an upper bound on the coverage.

What we can observe in Figure 5.3 is that circuits s953 and s1238 are able to eventually approach 100% coverage while others like b12 and s38417 get stuck below 30%. This may be an indication that although we have increased the number of assertions the greedy algorithm is able to select from its input set, there are simply not enough good assertions in that input set.

In early investigations using SAT queries, we observed that the assertion pool just before filtering was usually sufficient to cover almost every sensitizable bitflip. For each sensitizable bitflip, we would check if there existed at least one candidate assertion that would detect it. Obviously we cannot include millions of hardware assertions in our checker circuit, so we must perform some sort of filtering. Thus if we have mined sufficient assertions to cover almost all bitflips, and we also observe that increasing the area budget of greedy assertion selection does not yield a large improvement then this might mean that we did not hand off enough assertions to differential fault simulation, or that the assertion filtering step performed poorly. However, it is also possible that the length or number of simulations was not enough to fully stress all the potential bitflips in the circuit.

Regardless of not being able to approach 100% coverage, what we do notice is that even for a very small area budget, we are able to get a reasonable amount of coverage. This indicates that there exist assertions which provide a high amount of coverage relative to their hardware area cost.

5.6.2 Assertion Timing in Relation to Outputs

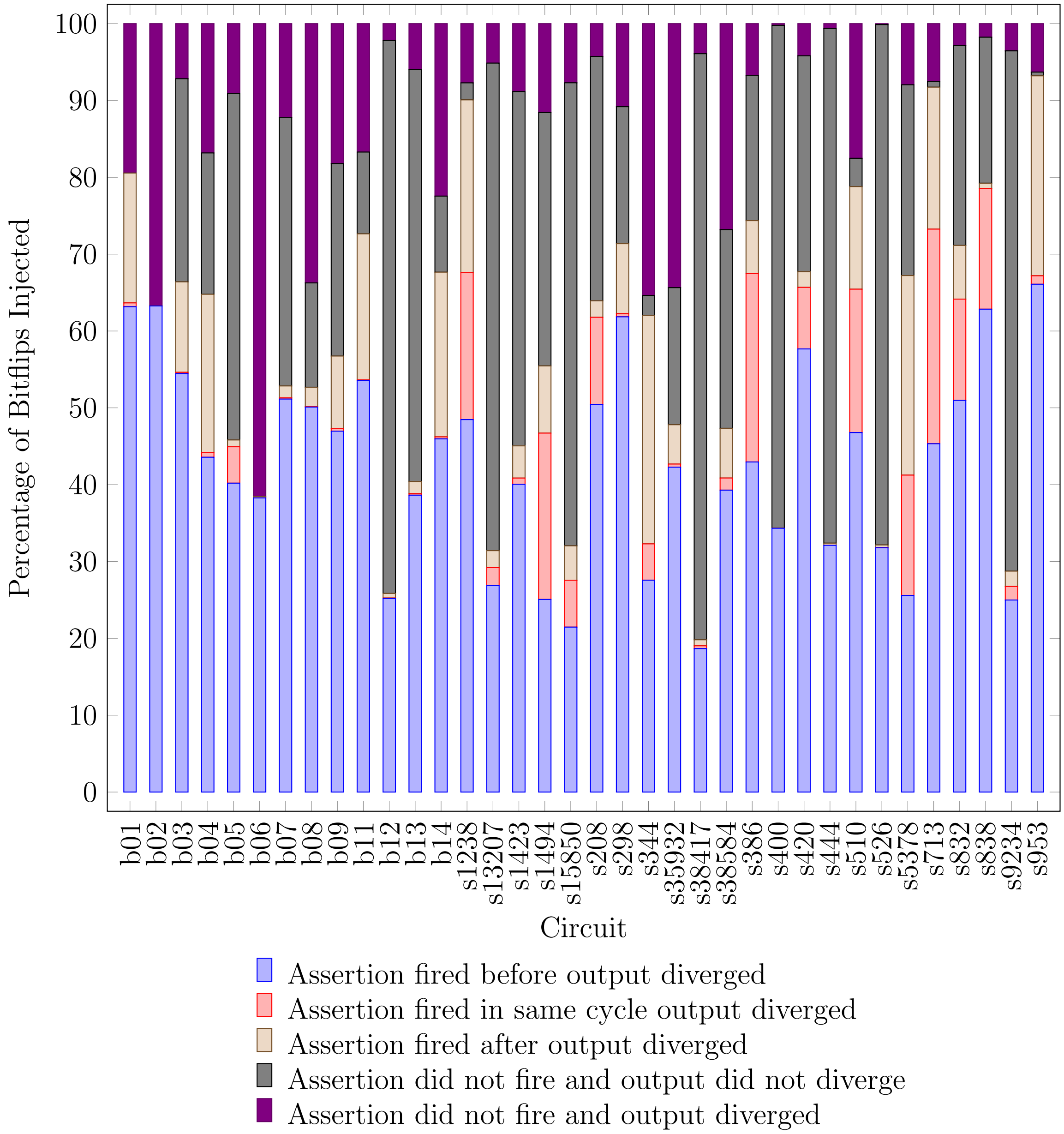


Figure 5.5: Assertion versus output divergence timing using ISCAS89 [4] and ITC99 circuits [5]

One possible benefit to assertions is that they can detect the presence of a fault before the fault manifests as an error. Thus in Figure 5.5 we kept track of the timing of assertions firing in relation to outputs diverging. We observe that circuits like b12 and s38417 that performed poorly in Section 5.6 are dominated by the category of neither assertions firing nor outputs diverging. This again suggests that the simulation did run for long enough to produce a random input sequence that would activate the effect of these bitflips.

Arguably the only catastrophic scenario is when an assertion did not fire but the outputs diverged. If this scenario occurs often for a particular circuit then the performance is indeed poor. Circuits such as b06 and b02 are very small, with the area budget only affording a few assertions, so the results here varied compared to the experiments performed for Figure 5.2.

5.7 Flow Runtime Measurements

Flow Step	b14	s13207	s15850
Finding Sensitizable Bits	35817.050	142.513	250.957
Assertion Mining	2013.994	14510.008	22566.477
Assertion Filtering	1515.633	258.754	1229.041
Assertion Ranking	8154.746	1147.192	1984.762
Greedy Assertion Selection	1582.353	228.626	378.016
Final DSIM Coverage	2108.490	908.272	1006.162
SAT Coverage	1369.026	29.493	75.446

Table 5.2: Runtime measurement in seconds of hardware assertion selection flow using ISCAS89 [4] and ITC99 circuits [5]

Table 5.2 lists the runtime of all the steps of the hardware assertion selection flow for one of the larger circuits. The runtime of each step in our flow depends not only on the size of the circuit, but also on the various runtime arguments that were defined in Section 5.3.

5.7.1 Finding Sensitizable Bits

It is evident that the runtime of b14 in Table 5.2 is dominated by the process of finding sensitizable bitflips. Since, this step is SAT-based, the runtime will increase exponentially as the size of the SAT instance increases, which depends on both the size of the circuit and the number of timeframes for which it is unrolled.

If this flow is expected to be run multiple times on the same FPGA netlist, this initial step can be extracted out of the flow. Information about which bitflips are sensitizable will remain valid across multiple runs, so this step is only needed to be run once.

5.7.2 Assertion Mining

The runtime of assertion mining is usually controlled by user arguments. There are arguments for how many clauses to learn and there is a timeout value which will stop the assertion mining if the frequency of learning new clauses descends below some threshold. We aim to select the number of learned clauses and the timeouts such that we will have a reasonable runtime. However, if this step is extracted out and the learned clauses are saved to a persistent database, then this runtime could be arbitrarily long.

The runtime of circuits s13207 and s15850 are dominated by this step. We speculate that because these circuits are around half the size of b14, that is more difficult to discover the same number of learned clauses as for b14, thus, the process of assertion mining is slower when the target number of learned clauses is the same.

5.7.3 Assertion Filtering

Because we have chosen to use selector difficulty potential as our filtering method, the runtime of this step will increase with the number of learned clauses and with the number of filter iterations. In every filter iteration we must iterate through each literal of each learned clause and then finally sort the learned clauses.

5.7.4 Assertion Ranking

The second largest time sink is the assertion ranking. We expect this to be the case. The runtime of assertion ranking step is proportional to the size of the circuit, the number of candidate assertions to simulate, the number of simulations, and the length of each simulation so we aim to select runtime arguments to our flow that will balance the quality of the generated violation matrix with the runtime. We assume that expanding the scope of the assertion ranking by adding more assertions, more simulations, and making the simulations longer will improve the violation matrix and thus improve the performance of greedy assertion selection.

5.7.5 Greedy Assertion Selection

The major contributor to the runtime of the greedy assertion selection is the fact that every time an assertion is selected, all other rows in the violation matrix have to be updated. Thus, the runtime of this step increases with the number of candidate assertions that passed the assertion filtering step, the area budget, and the number of sensitizable bitflips.

5.7.6 Final DSIM Coverage

The final DSIM coverage experiment is simply an assessment of the hardware assertions using the same differential fault simulation technique as in assertion ranking. Assuming all DSIM parameters are kept the same for this step as for assertion filtering, this step will take less time because the circuit we are simulating will have a smaller number of assertions included.

5.7.7 SAT Coverage

In this step we use a SAT query to check that every sensitizable bitflip can be detected by one of the hardware assertions we've selected. Thus the runtime will be proportional to the number of sensitizable bitflips. The worst case runtime will be exponential with regards to the size of the SAT instance, which not only depends on the size of the circuit, but also on the number of timeframes for which it is unrolled.

5.8 Summary

In this chapter, we have assessed the performance of our proposed hardware assertion selection flow. We defined the user-provided arguments and we showcased various coverage metrics across a variety of standard benchmark circuits. We found that the performance of the generated checker circuits was not competitive with DWC, and we discussed reasons as to why that is the case. Even though compared to DWC we were not able to cover the same number of bitflips with similar area, we were still able to show that many bitflips will be detected before their effect will be propagated to the circuit outputs. In the next chapter, we will provide some concluding remarks and discuss possibilities for future research.

Chapter 6

Conclusion and Future Work

In this thesis, we discussed a flow for generating checker circuits than can detect SEUs in FPGAs. While the most common way to mitigate SEUs is with hardware redundancy in the form of DWC and TMR, we sought a different approach to hardware redundancy. We sought to use the hardware invariants that can be generated from the CDCL mechanism of modern SAT solvers. By creating a CNF representation of the compiled FPGA netlist, hardware invariants can be automatically extracted. Then, out of a large pool of these hardware invariants, we attempt to cover as many bitflips as our area budget allows.

We expected that using the hardware assertion selection flow, we would be able to approach 100% bitflip coverage for most circuits when constraining the maximum area of the checker circuit to the size of the original circuit. However, while this was possible for a few circuits, most of our benchmark suite did not achieve this goal. Thus we concluded that this approach to hardware redundancy would not be competitive against more traditional approaches like DWC and TMR.

In some cases bitflips can be detected before they reach the output as discussed in

Section 5.6.2. In a hardware redundancy solution such as TMR or DWC, if comparison or voting is only used on the outputs, then it would only be possible to detect faults once they have manifested as errors on the outputs. Here we show that there are bitflips that can be detect before they affect the outputs. Thus, one benefit of hardware assertions in contrast to hardware redundancy such as TMR or DWC is that we can sometimes detect the state of the circuit becoming corrupted before this manifests as an error.

An approach like TMR does not have to be mutually exclusive to the hardware assertion approach. Regulations may mandate that an approach like TMR be used. For example, to reach the highest safety integrity level (SIL), redundancy may be a requirement [59]. The additional detection ability and circuit visibility provided by assertions can still provide a benefit. TMR allows a single fault to be detected and masked out. However, in an FPGA, upsets will continue to accumulate, so continuous scrubbing of the configuration memory is necessary for reliable operation. In Section 5.6.2 we have shown how assertions can detect many bitflips before they have an effect on the outputs. Thus, assertions could be added to a TMR implementation to improve detection latency, and thus allowing the scrubbing period to be decreased.

6.1 Future Work

6.1.1 Exploring the Effect of Multiple Bit Upsets

In the presence of multiple upsets, the approach of N-modular redundancy is no longer as reliable. Faults in multiple copies of the circuit mean a voter can no longer be guaranteed to mask out those faults. In contrast, we speculate that the greater the number of faults, the more likely it will be that an assertion will be violated. Thus it would be valuable to experimentally assess how the assertion approach can compare to traditional redundancy approaches in the presence of multiple upsets.

6.1.2 Exploring Different Methods of Generating Assertions

In Section 5.6.1 we showed the there exist assertions providing high bitflip coverage relative to their size. However, we simply rely on the CDCL mechanism to provide us with a plethora of candidate assertions by randomly exploring the search space.

Although the CDCL mechanism of a SAT solver conveniently produces assertions as a byproduct, we have little control over which area of the circuit is searched. While CDCL allows us to find a vast number of assertions, it does not allow us to narrow down the search space to target bitflips that have weak coverage. It would be beneficial to construct a search algorithm for finding assertions that can also target finding assertions that are candidates for specific LUT bits.

6.1.3 Exploiting Unsensitizable Bits in FPGA LUTs

A large portion of the SRAM cells comprising LUTs end up being unsensitizable as seen from Table 5.1. These unsensitizable bits can be exploited to increase fault masking [45]. In the presence of an SEU, a bit that was previously considered unsensitizable, might become sensitizable, and the authors seek to assign these bits such that fault masking is maximized. One area for future consideration is to try the opposite approach: assign all such unsensitizable bits such that fault masking in the presence of an SEU is minimized. As such we would increase the likelihood that such a fault would propagate to a hardware assertion, and the hardware assertion would detect it.

6.1.4 Faults in Routing and Other FPGA Components

This thesis focused only on SRAM cells that define LUTs. Other SRAM cells control the other functions of the PLBs (e.g. carry chains, flip-flops). There are also SRAM cells that control routing. Because routing architecture is usually not detailed in FPGA documentation, a simple custom routing architecture can be developed, or an FPGA architecture that has been completely reverse engineered like the Lattice iCE40 series [60].

6.2 Concluding Remarks

This thesis discussed novel methods of generating invariants and separating useful invariants to use as hardware assertions. The biggest lesson we learned is that even though the SAT CDCL mechanism allows us to generate a large pool of candidate assertions, the big challenge lied in how to separate the useful assertions for hardware checkers. Even though we did not attain our goal of being able to detect all bitflips, we developed software and methods for begin able to deal with a large pool of candidate assertions. The platform we developed will enable future research where different methods of generating candidate assertions can be substituted.

Bibliography

[1] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. USA: Kluwer Academic Publishers, 1999.

[2] B. Pratt, M. Fuller, M. Rice, and M. Wirthlin, "Reduced-precision redundancy for reliable FPGA communications systems in high-radiation environments," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 1, pp. 369–380, Jan. 2013.

[3] P. Taatizadeh and N. Nicolici, "An automated SAT-based method for the design of on-chip bit-flip detectors," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 101–108.

[4] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems*, 1989, pp. 1929–1934 vol.3.

[5] L. Basto, "First results of ITC'99 benchmark circuits," *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 54–59, 2000.

[6] T. R. Oldham and F. B. McLean, "Total ionizing dose effects in MOS oxides and devices," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 483–499, June 2003.

[7] C. M. Hsieh, P. C. Murley, and R. R. O'Brien, "A field-funneling effect on the collection of alpha-particle-generated carriers in silicon devices," *IEEE Electron Device Letters*, vol. 2, no. 4, pp. 103–105, April 1981.

[8] G. Bruguier and J. M. Palau, "Single particle-induced latchup," *IEEE Transactions on Nuclear Science*, vol. 43, no. 2, pp. 522–532, April 1996.

[9] P. E. Dodd and F. W. Sexton, "Critical charge concepts for CMOS SRAMs," *IEEE Transactions on Nuclear Science*, vol. 42, no. 6, pp. 1764–1771, Dec 1995.

[10] R. Koga, S. H. Penzin, K. B. Crawford, and W. R. Crain, "Single event functional interrupt (SEFI) sensitivity in microcircuits," in *RADECS 97. Fourth European Conference on Radiation and its Effects on Components and Systems (Cat. No.97TH8294)*, 1997, pp. 311–318.

[11] C. D. Norton, T. A. Werne, P. J. Pingree, and S. Geier, "An evaluation of the Xilinx Virtex-4 FPGA for on-board processing in an advanced imaging system," in *2009 IEEE Aerospace conference*, March 2009, pp. 1–9.

[12] N. Montealegre, D. Merodio, A. Fernández, and P. Armbruster, "In-flight reconfigurable FPGA-based space systems," in *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2015, pp. 1–8.

[13] K. Røed, "Single event upsets in SRAM FPGA based readout electronics for the time projection chamber in the ALICE experiment," Ph.D. dissertation, Bergen U., 2009.

[14] B. Bylsma, D. Cady, A. Celik, L. Durkin, J. Gilmore, J. Haley, V. Khotilovich, S. Lakdawala, J. Liu, M. Matveev, B. Padley, J. Roberts, J. Roe, A. Safonov, I. Suarez, D. Wood, and I. Zawisza, "Radiation testing of electronics for the CMS endcap muon system," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 698, pp. 242 – 248, 2013.

[15] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, April 1962.

[16] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, Oct 2004.

[17] B. Pratt, M. Caffrey, J. F. Carroll, P. Graham, K. Morgan, and M. Wirthlin, "Fine-grain SEU mitigation for FPGAs using partial TMR," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 2274–2280, Aug. 2008.

[18] K. S. Morgan, D. L. McMurtrey, B. H. Pratt, and M. J. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2065–2072, Dec 2007.

[19] N. A. Touba and E. J. McCluskey, “Logic synthesis of multilevel circuits with concurrent error detection,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 7, pp. 783–789, July 1997.

[20] J. H. Patel and L. Y. Fung, “Concurrent error detection in ALU’s by recomputing with shifted operands,” *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 589–595, July 1982.

[21] W. J. Townsend, J. A. Abraham, and E. E. Swartzlander, “Quadruple time redundancy adders [error correcting adder],” in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov. 2003, pp. 250–256.

[22] Y. M. Hsu and E. E. Swartzlander, “Time redundant error correcting adders and multipliers,” in *Proceedings 1992 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, Nov. 1992, pp. 247–256.

[23] F. G. de Lima Kastensmidt, G. Neuberger, R. F. Hentschke, L. Carro, and R. Reis, “Designing fault-tolerant techniques for SRAM-based FPGAs,” *IEEE Design Test of Computers*, vol. 21, no. 6, pp. 552–562, Nov 2004.

[24] A. Jacobs, G. Cieslewski, and A. D. George, “Overhead and reliability analysis of algorithm-based fault tolerance in FPGA systems,” in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 300–306.

[25] R. W. Hamming, “Error detecting and error correcting codes,” *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950.

[26] *Virtex-5 FPGA Configuration User Guide*, Xilinx. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[27] A. Stoddard, A. Gruwell, P. Zabriskie, and M. J. Wirthlin, “A hybrid approach to FPGA configuration scrubbing,” *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 497–503, Jan 2017.

[28] S. Satoh, Y. Tosaka, and S. A. Wender, “Geometric effect of multiple-bit soft errors induced by cosmic ray neutrons on DRAM’s,” *IEEE Electron Device Letters*, vol. 21, no. 6, pp. 310–312, June 2000.

[29] R. K. Lawrence and A. T. Kelly, “Single event effect induced multiple-cell upsets in a commercial 90 nm CMOS digital technology,” *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 3367–3374, Dec 2008.

[30] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, “Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule,” *IEEE Transactions on Electron Devices*, vol. 57, no. 7, pp. 1527–1538, July 2010.

[31] A. Sánchez-Macián, P. Reviriego, and J. A. Maestro, “Enhanced detection of double and triple adjacent errors in Hamming codes through selective bit placement,” *IEEE Transactions on Device and Materials Reliability*, vol. 12, no. 2, pp. 357–362, June 2012.

[32] A. Neale and M. Sachdev, “A new SEC-DED error correction code subclass for adjacent MBU tolerance in embedded memory,” *IEEE Transactions on Device and Materials Reliability*, vol. 13, no. 1, pp. 223–230, March 2013.

[33] A. Dutta and N. A. Touba, "Multiple bit upset tolerant memory using a selective cycle avoidance based SEC-DED-DAEC code," in *25th IEEE VLSI Test Symposium (VTS'07)*, May 2007, pp. 349–354.

[34] F. Brosser, E. Milh, V. Geijer, and P. Larsson-Edefors, "Assessing scrubbing techniques for Xilinx SRAM-based FPGAs in space applications," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 296–299.

[35] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 99–104.

[36] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. A. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of internal versus external SEU scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 2259–2266, Aug 2008.

[37] J. Lee, C. Chang, N. Jing, J. Su, S. Wen, R. Wong, and L. He, "Heterogeneous configuration memory scrubbing for soft error mitigation in FPGAs," in *2012 International Conference on Field-Programmable Technology*, Dec 2012, pp. 23–28.

[38] A. Sari and M. Psarakis, "Scrubbing-based SEU mitigation approach for systems-on-programmable-chips," in *2011 International Conference on Field-Programmable Technology*, Dec 2011, pp. 1–8.

[39] A. Sari, M. Psarakis, and D. Gizopoulos, "Combining checkpointing and scrubbing in FPGA-based real-time systems," in *2013 IEEE 31st VLSI Test Symposium (VTS)*, April 2013, pp. 1–6.

[40] H. R. Zarandi, S. G. Miremadi, D. K. Pradhan, and J. Mathew, "SEU-mitigation placement and routing algorithms and their impact in SRAM-based FPGAs," in *8th International Symposium on Quality Electronic Design (ISQED'07)*, March 2007, pp. 380–385.

[41] A. Das, S. Venkataraman, and A. Kumar, "Improving autonomous soft-error tolerance of FPGA through LUT configuration bit manipulation," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sep. 2013, pp. 1–8.

[42] J. Lee, Y. Hu, R. Majumdar, L. He, and M. Li, "Fault-tolerant resynthesis with dual-output LUTs," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2010, pp. 325–330.

[43] J. Lee, Z. Feng, and L. He, "In-place decomposition for robustness in FPGA," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2010, pp. 143–148.

[44] Y. Hu, Z. Feng, L. He, and R. Majumdar, "Robust FPGA resynthesis based on fault-tolerant Boolean matching," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2008, pp. 706–713.

[45] Z. Feng, N. Jing, and L. He, "IPF: In-place X-filling algorithm for the reliability of modern FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2226–2229, Oct 2014.

[46] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158.

[47] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commununications of the ACM*, vol. 5, no. 7, p. 394–397, Jul. 1962.

[48] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," pp. 530–535, 2001.

[49] J. P. Marques-Silva and K. A. Sakallah, "GRASP—a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '96. USA: IEEE Computer Society, 1997, p. 220–227.

[50] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, 1992.

[51] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, J. H. Siekmann and G. Wrightson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483.

[52] W.-T. Cheng and M.-L. Yu, “Differential fault simulation - a fast method using minimal memory,” in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, ser. DAC ’89. New York, NY, USA: Association for Computing Machinery, 1989, p. 424–428.

[53] “ABC: A system for sequential synthesis and verification.” [Online]. Available: https://people.eecs.berkeley.edu/~alanmi/abc/

[54] “Quartus II Web Edition.” [Online]. Available: https://fpgasoftware.intel.com/13.0sp1/

[55] N. Eén and N. Sörensson, “An extensible SAT-solver,” Berlin, Heidelberg, pp. 502–518, 2004.

[56] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers.” in *IJCAI International Joint Conference on Artificial Intelligence*, 07 2009, pp. 399–404.

[57] “Cyclone IV Device Handbook.” [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf

[58] R. H. Baer and H. J. Morrison, “Microcomputer controlled game,” U.S. Patent US4 207 087, Jun 10, 1980. [Online]. Available: https://worldwide.espacenet.com/patent/search/family/025267435/publication/US4207087A?q=pn%3DUS4207087

[59] International Electrotechnical Commission and others, “Functional safety of electrical/electronic/programmable electronic safety related systems,” 65A/550/FDIS, Standard, 2009.

[60] “Project Icestorm.” [Online]. Available: http://www.clifford.at/icestorm/