Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer Engineering (36^{th}cycle)

Formal Methods for Test and Reliability

By

Nikolaos Ioannis Deligiannis

******

Supervisor(s):
Prof. Matteo Sonza Reorda, Supervisor
Prof. Riccardo Cantoro, Co-Supervisor

Doctoral Examination Committee:
Prof. Chrysovalantis Kavousianos, Referee,
Prof. Görschwin Fey, Referee,
Prof. Wolfgang Kunz,
Prof. Marcello Traiola,
Prof. Ernesto Sanchez

Politecnico di Torino
2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Nikolaos Ioannis Deligiannis
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To my grandmother ***Parthena Deligianni*** *and*
in the heartfelt and enduring memory of my beloved grandfather, ***†Nikolaos Deligiannis****...*

Acknowledgements

First and foremost I would like to deeply thank my two supervisors, Matteo Sonza Reorda and Riccardo Cantoro who invested in me and allowed me to grow with their continuous support, care, and guidance. I would like to acknowledge Bernd Becker and my friend and colleague Tobias Faller further for their contribution and the paramount assistance they provided. Furthermore, I deeply thank my friends, for their selfless support and presence especially in the toughest times throughout my journey. Last but certainly not least, Father and Mother, nothing could have been achieved without you.

Abstract

In the rapidly evolving landscape of nanotechnology, where innovations promise groundbreaking advancements in various industries, ensuring the reliability and safety of digital circuits becomes paramount. While applicable to the broad category of commercial off-the-shelf products, this becomes significantly more evident when examining domains or industry sectors, such as automotive, aviation, railways, and the biomedical sector, that fall into the category of safety-critical applications. In these cases, the probability that a fault may activate an error and propagate to a failure that would endanger human lives or cause environmental damage should be carefully evaluated and kept under predefined thresholds. To achieve this result, the manufacturers must comply with strict safety standards and procedures that mandate rigorous coverage thresholds and comprehensive testing protocols. From end-of-manufacturing up until the in-field phase, each integrated circuit (IC) is subjected to several testing procedures to ensure that it meets stringent quality standards, functions reliably within specified parameters, and remains resilient to various environmental conditions throughout its operational lifespan. Design-for-testability (DfT) techniques are incorporated during the design phases of electronic circuits to enhance the testing process.

However, despite the presence of powerful electronic design automation (EDA) utilities, such as automatic test pattern generation (ATPG) tools, intended for use alongside DfT-compliant designs during testing, the relentless evolution of technology brings about faster, smaller, and denser circuits. This evolution renders certain utilities inadequate as the complexity of the test procedure significantly increases, as seen with Burn-In (BI) test. Burn-In, an omnipresent step in the test chain for products intended for use in safety-critical domains, was, until recently, conducted in its traditional static format. Notwithstanding its effectiveness, static BI test became less effective for newer, denser technologies as in its static form it was found not to fully exercise all internal parts of the ICs. Hence, it evolved into new dynamic

forms, where stress stimuli are applied in an internal manner on top of the external temperature and voltage increase. However, the generation of appropriate stress-inducing stimuli is a costly and arduous task for the test engineers, due to the lack of automation to aid the generation process.

Another test domain that could substantially benefit from automation is the in-field test. Continuous in-field testing enables the detection of faults or anomalies that may occur over time. Early detection of potential issues allows for proactive maintenance or corrective measures, reducing the risk of system failures in critical situations. However, the task of developing appropriate software test libraries (STLs) for such scenarios is typically a task that requires a lot of manual effort from the perspective of the test engineer. In fact, not only the test must consider parameters such as application time and memory footprint but it must also avoid targeting untestable faults of the design. This means that the test should only focus on those faults that are able to produce a failure in the operating scenario, ignoring those that can not produce any (critical) failure.

This PhD thesis proposes solutions, based on Formal Methods (FMs), addressing the aforementioned test topics. The manuscript is organized in three main parts.

The initial part provides an introduction and overview of the imperative need for testing and reliability in the modern digital era. It delves into the distinct testing areas that form the focus of this thesis.

The second part includes the three main contributions of the thesis. A section proposing FM-based solutions for dynamic BI test stress stimuli generation is first presented. These methods consider various switching activity metrics, and their effectiveness is showcased by applying them on scalar pipelined processors. The second contribution regards FM-based solutions targeting the identification of functionally untestable faults under the stuck-at and the cell-aware fault models. Lastly, the final contribution regards methodologies that aid the generation of STLs for microprocessors and GPUs.

The last part provides the conclusions of the overall work.

Contents

List of Figures

List of Tables

Nomenclature

Roman Symbols

H	High. Logic value of 1
L	Low. Logic value of 0

Acronyms / Abbreviations

ALU	Arithmetic and Logic Unit
ASIL	Automotive Safety Integrity Level
ATPG	Automatic Test Pattern Generation
BI	Burn-In
BIST	Built-In Self-Test
CAT	Cell-Aware Test
CNF	Conjunctive Normal Form
CTM	Cell Test Model
DFA	Deterministic Finite Automaton
DfT	Design-for-Testability
DUT	Device Under Test
EDA	Electronic Design Automation
FA	Full Adder

FC Fault Coverage

FF Flip Flop

FIT Failures In Time

FMECA Failure Mode, Effects and Criticality Analysis

FMs Formal Methods

IC Integrated Circuit

ISA Instruction Set Architecture

LFM Latent Fault Metric

MaxSAT Weighted Maximum Satisfiability

MIMD Multiple-Instructions Multiple-Data

MSB Most Significant Bit

PDF Path Delay Fault

PI Primary Input

PMHF Probabilistic Metric of Hardware Failures

PO Primary Output

PPI Pseudo-Primary Input

PPO Pseudo-Primary Output

SBST Software-Based Self-Test

SE Stress Efficiency

SIMD Single-Instruction Multiple-Data

SM Streaming Multiprocessor

SoC System-on-Chip

SPFM Single Point of Fault Metric

STL Software Test Library

SWA Switching Activity

TA Test Alternative

TF Timeframe

TP Test Program

UDFM User Defined Fault Model

VCM Validity Checker Module

Chapter 1

Introduction

1.1 The need for test and reliability

In the dynamic realm of digital technology, ensuring the reliability and effective testing of digital circuits emerges as a fundamental requirement. As our reliance on electronic systems continues to grow, ensuring digital circuits' seamless and error-free operation becomes crucial for various applications, ranging from consumer electronics to safety-critical systems in aerospace and healthcare. The rapid advancement of integrated circuits (ICs) and the increasing complexity of digital designs have accentuated the need for rigorous testing methodologies to identify and fix potential faults and vulnerabilities. Designers and IC manufacturers continue pushing the boundaries of Moore's law [1] by advancing to new technology nodes. This progression results in the emergence of newer, faster, and more powerful circuits as the number of transistors increases geometrically within the same silicon area.

Despite achieving significant performance gains, the escalating challenge lies in the exponential increase in the effort needed to test the latest generations of ICs adequately. This surge in testing complexity is further compounded by a proportional rise in the likelihood of faults, given the concurrent increase in transistor counts and the sensitivity of advanced technology nodes to new types of defects and weaknesses. As performance leaps forward, navigating the intricate landscape of testing becomes more critical, demanding heightened diligence to ensure the resilience and robust functionality of cutting-edge IC designs. Thus, from the manufacturer's standpoint, the testing process must be meticulously orchestrated and enhanced to ensure that

the newer ICs meet the same quality standards as their predecessors, despite the challenges posed by advancing technological nodes.

1.2 The safety-critical domain

Safety-critical systems represent a paramount category in the realm of digital technology, where the reliability of circuits takes on heightened significance. These systems, pervasive in aerospace, healthcare, and automotive industries, are tasked with functions directly impacting human safety and well-being. In safety-critical applications, the consequences of circuit failure can be severe, ranging from life-threatening incidents to environmental disasters. As our reliance on digital technology expands, integrating electronic systems into safety-critical domains underscores the critical need for rigorous testing and uncompromised reliability. The stringent requirements of safety standards, such as ISO-26262 [2] in the automotive sector and DO-254 [3] in aerospace, demand that digital circuits undergo meticulous testing to ensure their resilience in the face of diverse operational scenarios. In this context, pursuing technological advancements must be balanced with an unwavering commitment to maintaining the highest reliability standards, given the direct impact on human lives and the integrity of critical infrastructures.

From the manufacturers' standpoint, this necessitates a meticulous and comprehensive approach at every step of the digital circuit development process. Beginning with the initial design phase, manufacturers must anticipate and address potential fault scenarios, considering the critical nature of the applications in which their circuits will operate. This includes implementing robust fault-tolerant design strategies and ensuring that the circuit can detect, isolate, and recover from faults. The manufacturing process itself requires rigorous quality control to minimize defects and variations that could compromise reliability. As the design progresses to fabrication, testing methodologies become increasingly sophisticated to account for the intricacies of advanced technology nodes. Manufacturers must navigate the delicate balance between optimizing performance and maintaining reliability, adapting testing procedures to accommodate the growing complexity of integrated circuits. Post-production, ongoing monitoring, and maintenance become imperative, as even the most well-designed circuits may face challenges in real-world operating conditions. Ultimately, manufacturers must orchestrate a seamless integration of design, testing,

and quality assurance processes from concept to deployment to uphold the stringent standards demanded by safety-critical applications.

1.3 Open problems & State of the Art

Despite the incorporation of numerous Design-for-Testability (DfT) techniques in the design process of modern ICs and the wide variety of commercial electronic design automation (EDA) utilities, achieving the high thresholds imposed by safety standards remains an open problem. This task is far from being optimally solved due to its diverse and complex nature. Hence, in the absence of a global and uniform solution manufacturers identify and develop their own, in-house testing flows (in a "heuristic" manner) that are fine-tuned on their own products.

Among the various testing procedures modern ICs undergo, especially in the safety-critical domain, the importance and significance of switching activity maximization for dynamic Burn-In (BI) test, untestable fault identification, and in-field testing cannot be underestimated. These three pivotal tasks collectively address critical challenges in ensuring the robustness and dependability of integrated circuits (ICs). Switching activity (SWA) maximization during dynamic BI test not only accelerates the detection of latent defects but also enhances the overall reliability of ICs by subjecting them to realistic operational conditions. On the other hand, functionally untestable fault identification is instrumental in pinpointing elusive faults that traditional testing methodologies may overlook, thereby fortifying the comprehensive quality assessment of semiconductor devices. Finally, in-field testing stands at the forefront of proactive reliability maintenance, enabling the continuous monitoring and evaluation of ICs in real-world environments. Together, these contributions advance the state-of-the-art in semiconductor testing and contribute significantly to the longevity and performance assurance of electronic systems.

This thesis is organized into three distinct sections according to the contribution of each of the respective test domains. The first section regards the BI test. The second regards the safety-critical domain and in-field test and is linked with the identification of functionally untestable faults. The latter regards the automation of Software Test Library (STL) generation.

1.3.1 Burn-In test

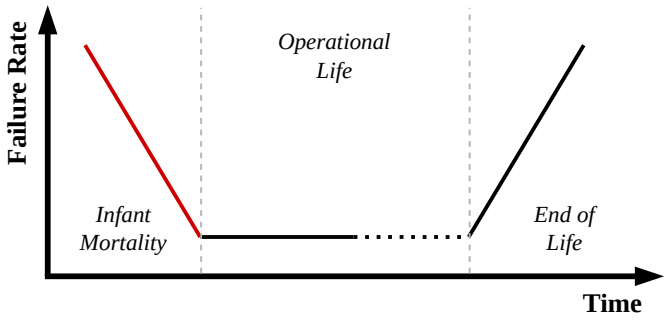

Fig. 1.1 The bathtub curve.

The BI test [4] is a crucial step typically conducted just before the final System-Level Test (if present), and it is a standard procedure in safety-critical domains. This test is pivotal as the primary countermeasure against the "Infant Mortality" phenomenon [5]. The *Infant Mortality* phase, which is characteristic of the bathtub curve as shown in Figure 1.1, is marked by the failure of a significant number of devices in the early stages of their life cycle. The BI test stands as a proactive measure, effectively mitigating issues and ensuring the reliability of systems in safety-critical applications.

During BI testing, the devices under test (DUTs) are placed within climatic chambers. Operating under higher than nominal power, thermal, and frequency conditions, the DUTs are exposed to a range of external and internal stresses. This rigorous process pushes the devices to their specification limits, sustaining the stress for extended periods, ranging from hours to days. This procedure aims to induce accelerated aging of the DUTs, equivalent to weeks or even months of regular operation. The multifaceted stress induces potential weaknesses in components to surface as observable defects. Any DUT showing such defects is promptly identified and removed from the testing process. By subjecting the DUTs to this intensive BI regimen, the testing effectively eliminates the time period associated with Infant Mortality. Through BI testing, these latent defects are brought to light, ensuring that only robust and reliable devices proceed to subsequent phases of production or deployment.

Up until recently, the most commonly applied BI procedure was *static* BI, during which the DUTs are exposed to a fixed and elevated temperature (and often voltage)

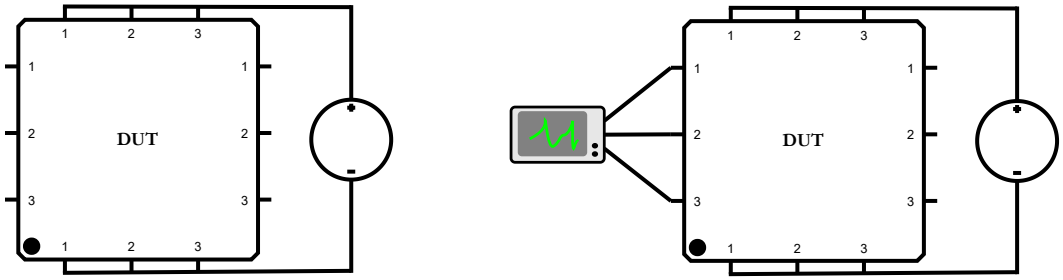

Fig. 1.2 Static BI (left) and Dynamic BI (right) formats.

for an extended period of time without the application of any kind of additional stress-inducing stimulus during the test. However, a major drawback of this approach is that nets of the circuit are not exercised [4] and the induced stress is uniform over all of them, differently than during the operational phase. Moreover, as device feature sizes continue to scale down and their structural and architectural complexity increases, the complexity of BI testing becomes prohibitive. Simultaneously, process variations introduce substantial uncertainty. Errors or misconfigurations in BI parameters can lead to catastrophic consequences, as the testing conditions may damage the DUTs, resulting in yield losses. For instance, an elevation in the junction temperature during BI testing can escalate leakage currents, potentially causing thermal runaway. Concerning test application time, all forms of accelerated aging methods prove time-consuming, with durations reaching tens or hundreds of hours, especially for emerging technologies. Consequently, these lengthy testing procedures can become bottlenecks in the overall manufacturing process.

To amend these obstacles, BI is evolving into new forms [6] where the stress is generated and induced to the DUTs with less dangerous and more controllable actions by also resorting to internal stress [7]. If the DUTs are equipped with Design for Testability (DfT) infrastructures (e.g., scan chains), internal stress can be easily generated by relying on them. However, in such an approach, the DUTs would operate in test mode, and thus, they would age in a way that could be different than in operational mode. Hence, this approach to the problem can possibly introduce unnecessary test escapes or even overstress the DUTs and potentially cause yield loss. On the other hand, by using functional stimuli to stress the DUTs, it is not possible to cause damage to the devices unless there exists a fatal design flaw, since the circuits are executing code as intended. Furthermore, it has been shown [8], that creating proper temperature gradients between different parts in a circuit, or cores

in a System-on-Chip (SoC), may enable the detection of defects that could hardly be targeted in other ways. Evidently, we can conclude that it is important to devise strategies able to generate purely functional test stimuli that maximize the switching activity (SWA) while the DUT works in normal mode.

1.3.1.1 State of the Art

Having emphasized the significance of subjecting DUTs to both external and internal stress during BI, a common industry practice involves internally stressing the device by applying stimuli with the aim of toggling every internal net at least once [9]. Given the time-consuming nature of the BI test, this is typically accomplished by leveraging DfT infrastructures, such as scan, and gradually applying stress vectors [10].

However, a drawback of this approach is that the DUTs operate in test mode, leading to aging behavior that differs from the operational mode. Consequently, this method may introduce unnecessary test escapes or even overstress the DUTs, potentially resulting in yield loss. To address this issue, manufacturers typically closely monitor the DUTs during stress application, ensuring that the stress profile aligns with nominal values and legacy readings by extrapolating from previous design and test iterations of the DUTs.

1.3.1.2 Main Contributions

This thesis proposes solutions in the domain of internal stress stimulus generation for dynamic BI test purposes. Initially, a methodology based on Formal Methods (FMs) is presented to generate functional stress sequences designed to maximize the constant and repeatable SWA of a sequential circuit [11, 12]. The effectiveness of this method is further demonstrated through the presentation of results for stress metric maximization using evolutionary techniques [13, 14]. Additionally, an alternative stress metric is considered, involving the incorporation of layout information obtained from a place-and-route procedure. This information is employed to derive topological insights and calculate minimum distances between neighboring parts of a design, to dissect the design in neighborhoods, and to generate functional sequences to optimally stress them [15].

1.3.2 Functionally untestable faults identification

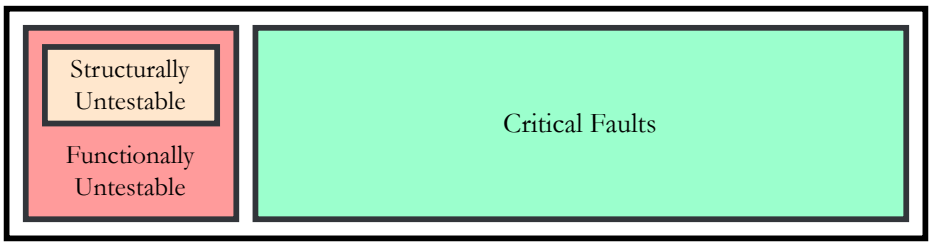

Fig. 1.3 Fault universe.

During the structural tests performed at the end of manufacturing but also during the functional tests performed in the mission cycle of the ICs sufficient fault coverage (FC) thresholds must be reached. This becomes more evident when considering the safety-critical application sector where the respective safety standards mandate high FC thresholds to be met during each test. The fault coverage is traditionally computed as:

$$\text{FC} = \frac{\text{Tested Faults}}{\text{Total Faults}} \tag{1.1}$$

The denominator represents the fault universe for the DUT. The fault universe is depicted in Figure 1.3. The vast majority of the faults (Critical Faults) are *testable*. This means that a test stimulus (i.e., an input stimulus able to activate the fault and produce a failure) can be generated by resorting to a given test methodology (e.g., via automatic test pattern generation (ATPG)). However, there exist faults for which no test can be found [16]. These faults are called *untestable faults* and are distinguished into two main categories.

The *structurally* untestable faults correspond to fault sites that cannot be forced to a specific logic value (0/1) due to either being unconnected (redundant) or due to the fault site being tied to a specific logic value. These types of untestable faults are further called *uncontrollable*. Furthermore, the set of structurally untestable faults also contains faults that although being controllable, i.e., can be forced to a specific value, their effect cannot propagate to an observable point of the circuit (e.g., any primary output) due to a blockage or masking of the fault effect. These faults are called *unobservable*.

As a natural extension, a superset of structurally untestable faults is identified as *functionally* untestable faults. Although these faults can be excited and propagated to an observable point in the circuit, they are incapable of causing any failure under the considered operational scenario or application profile. In the terminology of ISO-26262, such faults are called *safe*.

For instance, consider a standard processor equipped with a debugger module. While the module may be fully testable through DfT techniques, such as scan, the nature of the debugger module is such that it always remains inactive during the system's functional mode. Consequently, it becomes impossible to excite and/or propagate any fault within it.

As the untestable faults contribute to the number of total faults for a DUT, their presence has a negative impact on the computation of the FC (Equation (1.1)). This can be a serious issue, especially in the safety-critical domain where during the mission phase a functional test that is periodically performed must achieve a high FC threshold. As an example, according to ISO-26262, a functional FC of 98% must be achieved for every critical system of the car (e.g., airbags). However, as researchers have showcased [17, 18], given the mission profile of a system the number of functionally untestable faults can reach percentages $>$10% at times. Thus, unless these faults are identified and eliminated from the fault list of the DUT, compliance with the safety standards may be impossible. Hence what is of interest for the test engineers is the *testable fault coverage*, which is computed as:

$$\text{Testable FC} = \frac{\text{Tested Faults}}{\text{Total Faults} - \text{Untestable Faults}} \tag{1.2}$$

In the safety-critical domain, among the several possible solutions, the Failure Mode, Effects, and Criticality Analysis (FMECA) [19] is in charge of identifying which faults are not able to produce any failure. Since FMECA is barely automated, the issue of identifying such faults is a major concern for the industry.

1.3.2.1 State of the Art

Regarding the identification of structurally untestable faults, the ATPG engines are able to deduce certain uncontrollable faults by performing toggling checks on the DUT. As for the unobservable faults, the branch and bound algorithms used either

abort them after reaching a predefined wall clock timer per fault or given enough time they are able to identify them as untestable.

When considering functionally untestable faults the dominant practice is for the test engineers to have an in-depth knowledge of the underlying architecture of the DUT and to perform manual analyses on the circuitry in combination with the mission/application profile in order to deduce which circuit regions can be considered redundant. There exists a lack of automation since the EDA tools lack the utility to formulate complex functional constraints and consider them during the pattern generation phase [17, 20, 21].

1.3.2.2 Main Contributions

This thesis proposes solutions in the area of uncontrollable fault identification [22] and also for functionally untestable fault identification under the cell-aware fault model [23]. The methods are based on FMs and showcase how elaborate operational constraints can be incorporated in a test pattern generation phase to accurately circumscribe the search space and enable tools such as Bounded Model Checking (BMC) solvers and Satisfiability (SAT) solvers to deduce whether lines or faults in a sequential circuit are untestable under the considered functional scenario.

1.3.3 In-Field test

In the realm of safety-critical applications, where reliability is paramount, the introduction of in-field testing emerges as a pivotal strategy to ensure ongoing operational integrity. As technological advancements and complexities continue to define our critical systems, the need for real-time monitoring and evaluation becomes increasingly crucial. In-field testing not only addresses the challenges unique to the dynamic operational environment but also provides an essential layer of assurance, allowing for proactive identification and mitigation of potential issues. This approach stands at the forefront of safeguarding safety-critical applications, offering a responsive and adaptive methodology to uphold stringent safety standards throughout the entire lifecycle.

This challenging target is normally achieved using a mix of different solutions, including redundancy, DfT, and functional self-test. When it comes to the latter,

which involves conducting tests by manipulating functional inputs and observing corresponding outputs without relying on DfT, test engineers frequently turn to *Software-Based Self-Testing* (SBST). This approach helps them generate STLs tailored to specific fault models [24, 25].

The SBST strategy is a flexible and non-intrusive method that relies on carefully crafted test programs (TPs) and utilizes the Instruction Set Architecture (ISA) to apply test patterns. These patterns are designed to activate potential faults within a targeted unit of a device and propagate their effects to some visible location(s). TPs are primarily developed at the assembly level and in practice, an STL is a collection of TPs. These TPs effectively allow the detection of hardware faults in a device. Their development is up to the manufacturing company, which then passes them to the system company, which integrates them into the application software. STLs can be activated during the operational phase, e.g., during the application idle times. The dominant fault models currently supported by the safety standards are the stuck-at and the transition delay models. STLs have been used extensively in the past for end-of-manufacturing [25] and in-field test in the case where the DUT is a processor or controller [26, 27] and are now used for in-field test of GPUs as well [28].

While STLs offer numerous advantages, their development is frequently a time-consuming and expensive process, necessitating the expertise of a skilled developer familiar with the intricacies of the specific processor's micro-architecture. The manual creation of the program involves ensuring hardware faults are detectable, requiring an understanding of the micro-architecture's behavior under fault influence, which is a non-intuitive and time-consuming task. Furthermore, the SBST program must be tailored to its environment (e.g., memory mapping and peripheral configurations), making the creation of SBST a complex process that, up until now, had to be repeated for each new design.

1.3.3.1 State of the Art

The current approach to STL development heavily relies on manual efforts by test engineers, primarily due to the unique nature of these tests. The requirement for periodic execution during idle cycles in a device's mission phase introduces application-specific parameters such as test application time, memory footprint, and firmware configuration. Unfortunately, there is limited room for abstraction or generalization in addressing these considerations. Another contributing factor to the

lack of automation in functional pattern generation is the complexity and difficulty of formulating operational constraints. These constraints arise from the combination of system specifications and the system's mission profile.

One obstacle to automation lies in the intricate nature of these operational constraints, making their formulation challenging. Although traditional ATPG tools provide a certain level of constraint formulation, their focus is typically limited to the primary (PIs) and pseudo-primary inputs (PPIs) and outputs (POs, PPOs) of the DUT, falling short of addressing the broader operational context.

Despite these challenges, in specific scenarios, a certain level of automation can be applied. For instance, when examining an arithmetic circuit within the Arithmetic and Logic Unit (ALU) of a microcontroller, a degree of automation becomes feasible. Applying a pseudo-random approach or employing ATPG on an isolated unit, such as the ALU, can be advantageous. This is particularly true when the arithmetic instructions are known in advance, facilitating the generation of a functional test routine. Random functional approaches [29] as well as evolutionary techniques-based approaches [30] have also been employed in the past, which offer a significant degree of automation.

1.3.3.2 Main Contributions

This thesis introduces innovative solutions leveraging FMs for the streamlined generation of STLs tailored to both CPUs [31] and GPUs [32]. Specifically, in the context of CPUs, it demonstrates that with a modular structure and comprehensive documentation, as exemplified by the RISC-V architecture [33], a substantial portion of STL generation can be automated. Conversely, for more intricate and densely designed GPUs, the proposed method systematically addresses challenging-to-test faults, thereby enhancing the efficiency of an existing STL in terms of testable FC.

1.4 Formal Methods

Formal methods are rigorous mathematical techniques that are used in a variety of applications spanning specification up to test and validation of software and hardware [34]. FMs are composed of well-formed statements in a mathematical logic that are used to produce rigorous deductions in that logic, which means that

each step follows a rule of inference. The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design and establish a correctness or safety property that is true for all possible inputs. However, due to the extremely large search spaces that characterize modern systems, FMs are used in a "divide and conquer" fashion either on isolated components of the system or on the most critical parts of it.

A design and test area where FMs are dominant is formal verification. In today's systems where validation by mere simulation means is not sufficient a mathematical proof of correctness is essential. Errors that occur during the translation of a specification into the final IC implementation if not detected will cause all produced chips to be erroneous. Hence for each step from the RT-level all the way to the layout level of a design formal verification is performed.

Furthermore, it has long been known that an ATPG problem can be reduced to a Satisfiability instance and solved using an SAT solver [35]. However, this approach was not widely adopted as the so-called structural ATPG approaches generally provide better scalability. More recently, significant improvements in the underlying SAT solvers in conjunction with extended solving capabilities specifically developed and tailored to ATPG led to an increased interest in such techniques [36, 37].

The following subsections introduce fundamental concepts of FMs, which serve as foundational elements for all contributions in this thesis. All proposed methods presented in this thesis were developed in an FM framework named `FreiTest`. `FreiTest` is an ATPG framework that is derived from PHAETON [38]. The framework has been fully rewritten and redesigned primarily for STL generation. This includes the circuit import, functional constraint handling, fault simulation, data export, and visualization, as well as Conjunctive Normal Form (CNF) formula generation and the whole ATPG process itself.

1.4.1 Reduction of ATPG to Boolean satisfiability

The main goal of reducing the ATPG problem to the problem of Boolean Satisfiability is comprised of multiple steps. Assuming a sequential circuit, initially the circuit is unrolled in time. This means, that initially the circuit's structure is duplicated for a discrete and finite amount of times. Each instance of the circuit is called a *timeframe* (TF) and can be interpreted as a clock cycle. The PPIs of the n^{th} instance of the

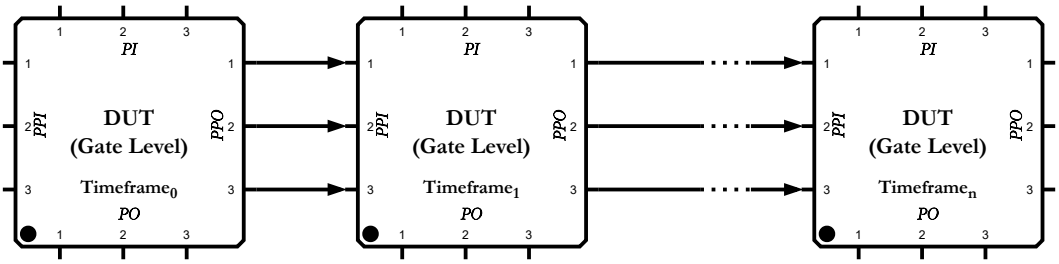

Fig. 1.4 Circuit unrolling technique.

circuit are driven from the PPOs of the n-1$^{\text{th}}$ instance. The initial state of the circuit can be freely specified. This process is depicted in Figure 1.4.

Table 1.1 Representation of different Boolean operators as CNF sub-expressions.

Operator	**Boolean Expression**	**CNF Sub-Expression**
AND	$C = A \cdot B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
NAND	$C = \overline{A \cdot B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$
OR	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
BUF	$C = A$	$(\overline{A} \vee C) \wedge (A \vee \overline{C})$
NOT	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
XOR	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$
XNOR	$C = \overline{A \oplus B}$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee \overline{C})$

The next step, is to derive the Boolean formula of the unrolled circuit in CNF in order for an underlying SAT solver engine to be able to handle it effectively. While tools like Satisfiability Modulo Theory solvers are proficient in managing Boolean formulas expressed in traditional Boolean algebra format, their scalability is comparatively limited when compared to conventional SAT solvers. In order to extract the Boolean formula in CNF we perform *symbolic simulation* on the unrolled circuit. During symbolic simulation we traverse the circuit structure in topological order i.e., starting from the PIs towards the POs and we perform variable assignment to each primitive cell of the circuit we encounter. Each primitive is identified and its Boolean formula is replaced with its equisatisfiable formula coming from the Tseitin transformation [39] as shown in Table 1.1. Finally, all generated sub-formulas are “merged" together with conjunctions ($\wedge$). A simplified example is shown in Figure 1.5.

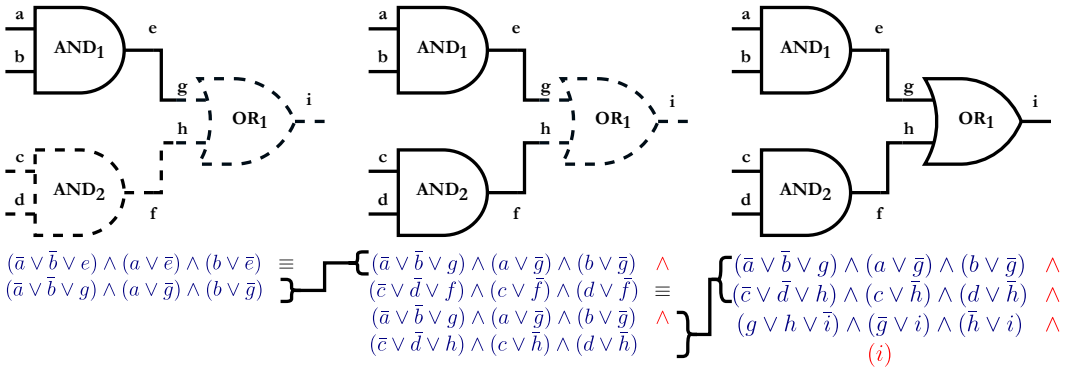

Fig. 1.5 Step-by-step CNF generation via symbolic simulation.

In the CNF, each set of disjunctions is termed a *clause*, and each variable within a clause is referred to as a *literal* and can be either positively (`true`) or negatively (`false`) assigned. It is important to highlight that in the concluding step, a *unit clause* (i.e., a clause comprised of precisely 1 literal) is incorporated into the CNF, featuring the output of the circuit. By doing so, we implicitly instruct the SAT engine to identify a literal assignment that renders the circuit output equal to 1 (*model*). After the CNF of the circuit is generated, we can optionally enforce functional constraints and ask an SAT solver to generate a model. Assuming that this is feasible, then by identifying the literals mapped to the PIs and PPIs of the circuit, we can perform a conversion to 0/1 logic and extract an input pattern.

1.4.1.1 ATPG for combinational circuits via formal methods

In Figure 1.6, a miter circuit of a combinational circuit is shown. While considering permanent hardware faults (stuck-at), we will see in a step-by-step way how we can rely on SAT solving to perform automatic test pattern generation.

The stuck-at fault site is shown on the red circuit which represents the faulty machine (FM). The green circuit represents the fault-free golden machine (GM), and the outputs of the two circuits drive the final XOR gate. Note that the PIs of the circuits are the same. The first step is to perform the symbolic simulation of the miter circuit to generate the Boolean formula in CNF (as shown in Figure 1.5). The resulting CNF is the following:

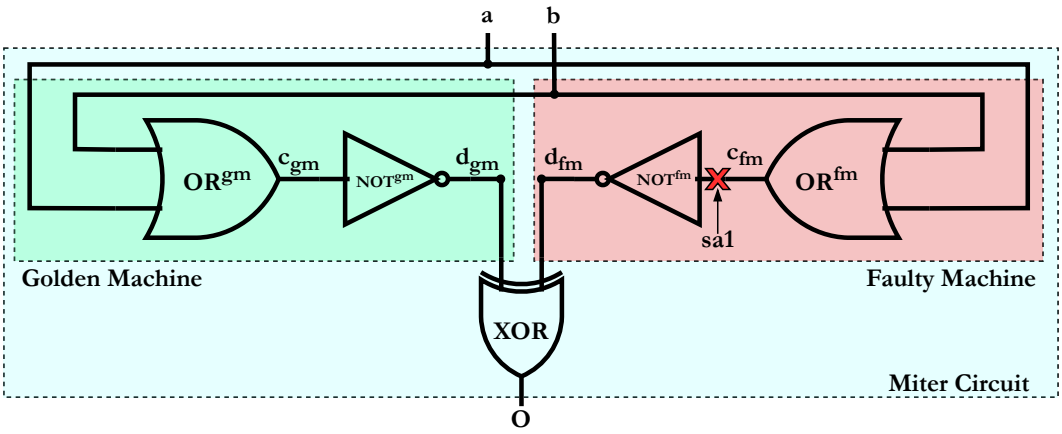

Fig. 1.6 SAT-based ATPG example for combinational circuit and stuck-at 1 fault.

$$
\begin{aligned}
\mathrm{CNF} &:= \mathrm{CNF}_{\mathrm{gm}} \wedge \mathrm{CNF}_{\mathrm{fm}} \wedge \mathrm{CNF}_{\mathrm{Miter}} \\
\mathrm{CNF}_{\mathrm{gm}} &:= \overbrace{(a \vee b \vee \overline{c_{\mathrm{gm}}}) \wedge (\overline{a} \vee c_{\mathrm{gm}}) \wedge (\overline{b} \vee c_{\mathrm{gm}})}^{\mathrm{OR^{gm}}} \wedge \overbrace{(\overline{c_{\mathrm{gm}}} \vee \overline{d_{\mathrm{gm}}}) \wedge (c_{\mathrm{gm}} \vee d_{\mathrm{gm}})}^{\mathrm{NOT^{gm}}} \\
\mathrm{CNF}_{\mathrm{fm}} &:= \overbrace{(a \vee b \vee \overline{c_{\mathrm{fm}}}) \wedge (\overline{a} \vee c_{\mathrm{fm}}) \wedge (\overline{b} \vee c_{\mathrm{fm}})}^{\mathrm{OR^{fm}}} \wedge \overbrace{(\mathrm{sa1})}^{\text{stuck-at 1}} \wedge \overbrace{(\overline{\mathrm{sa1}} \vee \overline{d_{\mathrm{fm}}}) \wedge (\mathrm{sa1} \vee d_{\mathrm{fm}})}^{\mathrm{NOT^{fm}}} \\
\mathrm{CNF}_{\mathrm{Miter}} &:= \overbrace{(\overline{d_{\mathrm{gm}}} \vee \overline{d_{\mathrm{fm}}} \vee \overline{\mathrm{O}}) \wedge (d_{\mathrm{gm}} \vee d_{\mathrm{fm}} \vee \overline{\mathrm{O}}) \wedge (d_{\mathrm{gm}} \vee \overline{d_{\mathrm{fm}}} \vee \mathrm{O}) \wedge (\overline{d_{\mathrm{gm}}} \vee d_{\mathrm{fm}} \vee \mathrm{O})}^{\mathrm{XOR}} \wedge (\mathrm{O}) \\
&\Longleftrightarrow \\
\mathrm{CNF} &= (a \vee b \vee \overline{c_{\mathrm{gm}}}) \wedge (\overline{a} \vee c_{\mathrm{gm}}) \wedge (\overline{b} \vee c_{\mathrm{gm}}) \wedge (\overline{c_{\mathrm{gm}}} \vee \overline{d_{\mathrm{gm}}}) \wedge (c_{\mathrm{gm}} \vee d_{\mathrm{gm}}) \wedge \\
&\quad (a \vee b \vee \overline{c_{\mathrm{fm}}}) \wedge (\overline{a} \vee c_{\mathrm{fm}}) \wedge (\overline{b} \vee c_{\mathrm{fm}}) \wedge (\mathrm{sa1}) \wedge (\overline{\mathrm{sa1}} \vee \overline{d_{\mathrm{fm}}}) \wedge (\mathrm{sa1} \vee d_{\mathrm{fm}}) \wedge \\
&\quad (\overline{d_{\mathrm{gm}}} \vee \overline{d_{\mathrm{fm}}} \vee \overline{\mathrm{O}}) \wedge (d_{\mathrm{gm}} \vee d_{\mathrm{fm}} \vee \overline{\mathrm{O}}) \wedge (d_{\mathrm{gm}} \vee \overline{d_{\mathrm{fm}}} \vee \mathrm{O}) \wedge (\overline{d_{\mathrm{gm}}} \vee d_{\mathrm{fm}} \vee \mathrm{O}) \wedge (\mathrm{O})
\end{aligned}
\tag{1.3}
$$

Note that for the faulty machine, an additional literal (sa1) has been introduced to represent the presence of the stuck-at 1, affecting the output of the OR gate (c_{fm}). This extra literal is necessary to mark the beginning of the fault cone. Furthermore, with the Boolean formula of the miter circuit, the test engineer can incorporate any functional or structural constraints into the CNF, not limited to the observable parts of the circuit (i.e., its primary inputs). Further discussion on constraint formulation will be provided later in the text. Lastly, the unit clause (O), linked to the output of the miter circuit, is added to the CNF. This is to explicitly guarantee that a difference

will be detected between the FM and the GM, i.e., the fault effect will propagate to a primary output.

At this point, we can ask an SAT solver to search for a model for Equation (1.3). Given that the circuit is trivial, we can easily identify the input pattern required to test the indicated stuck-at 1 fault. That pattern is <a,b> := <0,0>.

Most SAT-solvers are based on the CDCL algorithm [40, 41], which draws inspiration and augments the DPLL (or DLL) algorithm [42, 43]. A DPLL SAT solver employs a systematic backtracking search procedure to explore the (exponentially sized) space of variable assignments looking for satisfying assignments. In Chapter A we show how an SAT-solver would identify the aforementioned test pattern.

Finally, after several steps, the SAT-solver manages to identify a model for the CNF. By isolating the literals corresponding to the PIs of the circuit, we extract the test pattern <*a*, *b*> := <0,0>.

1.4.1.2 ATPG for sequential circuits via formal methods

Let us now consider the case where a sequential circuit is our DUT, and we wish to perform functional ATPG. That is, we do not assume full control over the PPI of the circuit via the presence of any kind of DfT infrastructure, such as scan. In that case, it may be possible that the fault excitation and propagation may not be feasible to happen in one clock cycle due to the fact that we also have to consider the circuit states. For this reason, we cannot rely on plain SAT-solving. The tool we will employ for this task is Bounded Model Checking.

Bounded model-checking algorithms unroll the state machine of the circuit for a fixed number of steps, k, and check whether a property violation can occur in k or fewer steps. This typically involves encoding the restricted model as an instance of SAT. The process can be repeated with larger and larger values of k until all possible violations have been ruled out.

During bounded model checking, we embed a desired property (P), i.e., the desired switching activity is reached, into the CNF formula by defining an upper bound (k); we ask the underlying solver to check whether this property can be reached up to a maximum depth k starting from a well defined initial state (I) by

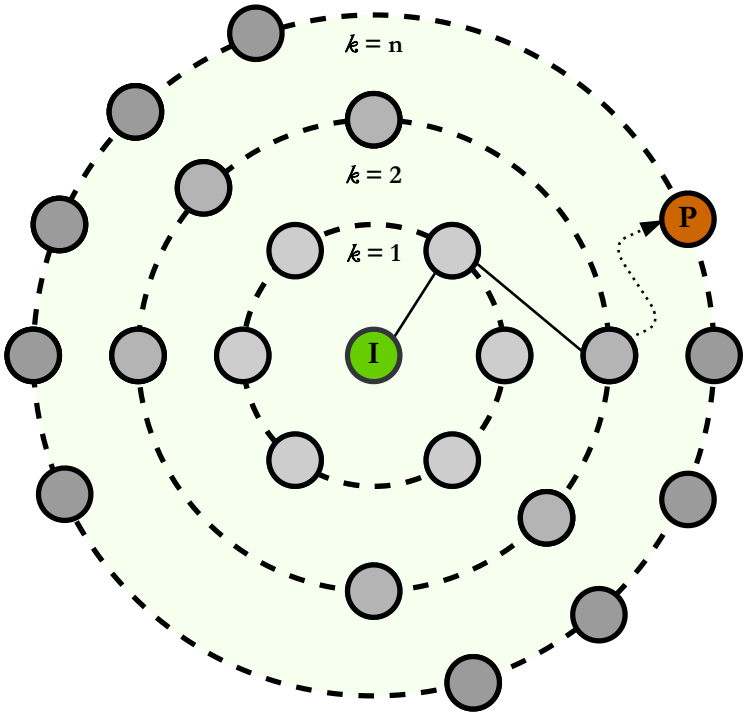

Fig. 1.7 Bounded Model Checking.

following a transition relation (T). This concept is depicted in Figure 1.7, whereas the general CNF formula for an arbitrary step k is built as:

$$\text{CNF}_k = I^0 \wedge \bigwedge_{i=0}^{k-1} T^{i \rightarrow i+1} \wedge P^k \tag{1.4}$$

Let us consider the circuit of Figure 1.8. We assume a stuck-at fault is present in the output of the D Flip-Flop, and we need to generate a test vector for it. Also, we assume that the initial state of the Flip-Flop is 0. Given that the circuit is relatively trivial, we can infer that the pattern is a = <1, X> to test for the aforementioned stuck-at 0 fault.

In Chapter B, we show an example of how a BMC-solver would identify the pattern a = <1, 1> after 2 unrolls (k = 2) of the circuit.

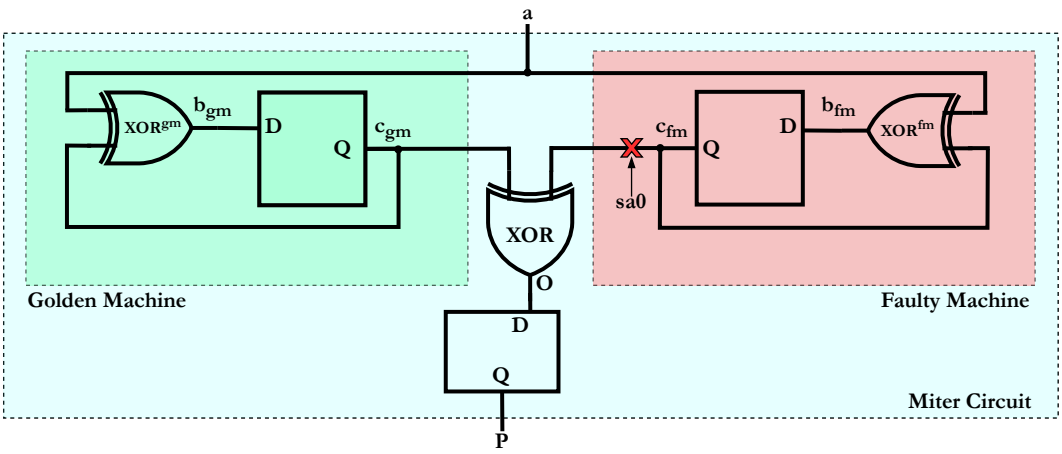

Fig. 1.8 BMC-based ATPG example for sequential circuit and stuck-at 0 fault.

1.4.2 Validity checker modules

One key aspect in which formal methods excel over traditional structural EDA tools is the elegance in which they allow for elaborate and accurate constraint formulation through rigorous propositional logic statements. As shown in the example of Figure 1.5, by adding a positively assigned literal that is mapped to a PO of the circuit, we instruct the underlying solver to find a model that respects this constraint i.e., a pattern for which the circuit output is 1. However, considering that after the symbolic simulation, every single net of the design is mapped to at least 1 unique literal[1] we can create arbitrary statements for every internal part of the design.

For example, considering the example of Figure 1.5 let us assume that we want to inquire further that the generated pattern for which the circuit's output is set to 1 also sets the output of the AND_1 gate to 1. In order to do so, we can simply add the unit clause (e) to the CNF. In that way, in order for the overall CNF to be satisfied, the added unit clause must hold true.

To formulate more complicated functional constraints, however, a general mechanism for constraint application is required. This is achieved by the Validity Checker Module (VCM) [38]. The VCM is a small circuit, with respect to the DUT, written in a Hardware Description Language (HDL) like SystemVerilog and later synthesized into a gate-level representation using the same technology library as the DUT. The

[1]For encoding 0/1 logic 1 literal is enough. To support more logic types (e.g., 0/1/X and 0/1/X/Z), more than 1 literal is required to encode each net of the design.

DUT and VCM are encoded into a single *miter circuit* as shown in Figure 1.9. The VCM's circuit inputs are connected to the miter circuit and thus enable access to every part of the faulty and fault-free DUT. The VCM's logic computes if the state and behavior of the DUT match the constraints that have been encoded inside the VCM. The VCM's validation result is indicated via its outputs. A Boolean value of 1 indicates that the constraint is held, while a 0 indicates that the constraint is not held and the miter circuit shows an invalid behavior.

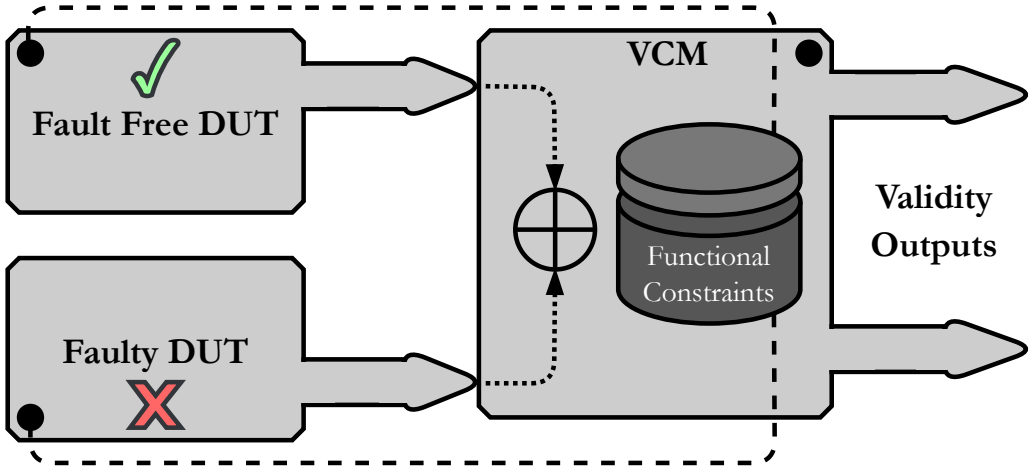

Fig. 1.9 Miter circuit and VCM interaction.

By performing a symbolic simulation on the DUT and the VCM gate-level descriptions the Boolean formula of the combined circuit is generated in a CNF. In this way, constraints are embedded in the CNF, hence making the underlying FM engine aware of the desired functional scenario. Thus one is able to formulate complex functional constraints via circuit logic inside the VCM and apply those constraints to the DUT.

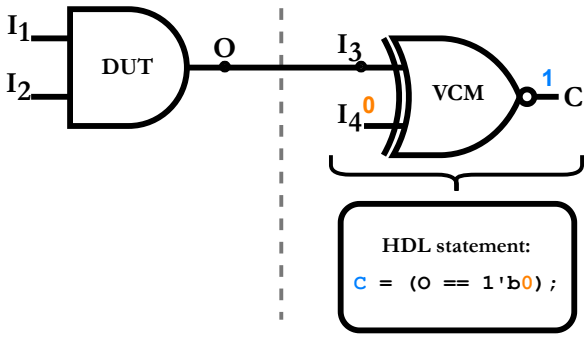

Fig. 1.10 Constraint application through the VCM.

A very simplified example of this interaction is shown in Figure 1.10. As the concept of the VCM is applicable to constraining any generic circuit, for brevity we replace the miter circuit with a single AND gate (called CUT in the following) in this example. We wish to constrain the output of the AND gate always to be 0. Note, however, that the constraints encoded in the VCM are typically of complex nature instead. The final CNF of the circuit is the conjunction of the two sub-CNFs, for the CUT and the VCM respectively, i.e., $\text{CNF} = \text{CNF}_{\text{cut}} \wedge \text{CNF}_{\text{vcm}}$. The two independent CNFs CNF_{cut} and CNF_{vcm} are constructed using the Tseitin transformation (see Table 1.1) of the logic AND and XOR gates respectively. For the VCM we additionally add a unit clause for each VCM output, i.e. C here, to enforce the constraint always to hold.

$$
\begin{aligned}
\text{CNF}_{\text{cut}} &:= (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\text{I}_1 \vee \neg \text{O}) \wedge (\text{I}_2 \vee \neg \text{O}) \\
\text{CNF}_{\text{vcm}} &:= (\neg \text{I}_3 \vee \neg \text{I}_4 \vee \ \ \text{C}) \wedge (\ \ \text{I}_3 \vee \ \ \text{I}_4 \vee \ \ \text{C}) \wedge \\
&\quad\ (\ \ \text{I}_3 \vee \neg \text{I}_4 \vee \neg \text{C}) \wedge (\neg \text{I}_3 \vee \ \ \text{I}_4 \vee \neg \text{C}) \wedge \\
&\quad\ (\neg \text{I}_4) \wedge (C)
\end{aligned}
$$

Additionally, a unit clause is created with the constraint that forces the VCM's output to 1, which must be evaluated as true, simplifies CNF_{vcm} to the unit clause $(\neg \text{I}_3)$ (step i.) which is equal to the unit clause $(\neg \text{O})$ since the input is driven by the AND gate's output (step ii.). Given that the unit clause must be positively assigned in order for the CNF to be satisfied too, two extra clauses are eliminated (step iii). This sequence of events is shown below:

$$
\begin{aligned}
\text{CNF} &= \text{CNF}_{\text{cut}} \wedge \text{CNF}_{\text{vcm}} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\text{I}_1 \vee \neg \text{O}) \wedge (\text{I}_2 \vee \neg \text{O}) \wedge (\neg \text{I}_3) && \text{i.} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\text{I}_1 \vee \neg \text{O}) \wedge (\text{I}_2 \vee \neg \text{O}) \wedge (\neg \text{O}) && \text{ii.} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\neg \text{O}) && \text{iii.} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2) \wedge (\neg \text{O}) && \text{iv.}
\end{aligned}
$$

1.4.3 Initial state extraction & application

In all proposed methods discussed in this thesis, all circuits are considered to be initialized to a well-defined state. That means that there are no X values within the pipeline, i.e., each Flip-Flop (FF) holds a 0/1 logic value. Traditionally, when considering processor circuits, this can be achieved either by asserting the processor's `RESET` signal before any code or firmware is executed on the system (general case), or given the presence of an application profile it may be that a certain initialization instruction sequence must be applied first. No matter the case, the result is that the sequential circuit, after the application of an initialization process, is driven to an initial state.

As previously stated, formal methods enable the modeling of arbitrary initial states during, e.g., an SAT-based ATPG process. In the methods presented in this thesis, this is achieved by utilizing the VCM circuit. The initial state of an arbitrary sequential circuit comprised of n FFs can be considered as a vector with each FF value concatenated as:

$$\text{init} = < \text{FF}_i >_{i=0}^{n-1} = < \text{FF}_0, \text{FF}_1, \cdots, \text{FF}_{n-1} > \tag{1.5}$$

Assuming that such a vector is available, then it is possible to map each logic value to the corresponding FF literals and during the very first timeframe constrain them to hold these specific values. Figure 1.11 depicts a strategy for extracting the initial state of a sequential circuit, stemming from the activation of the global `RESET` signal. Assuming that the signal is asserted for a total of m clock cycles, then on the very last clock cycle, the circuit is driven to the state s_m colored in yellow. By using a logic simulator and the gate-level of the respective circuit it is possible to extract the initialization vector (Equation (1.5)) with a few lines of HDL code in the testbench circuit used for the simulation. For the purpose of the initial state extraction, the framework `R4VES` [44] was used. Lastly, the extracted initialization vector is passed into the formal methods framework and the VCM circuit is responsible for embedding it in the CNF generation process.

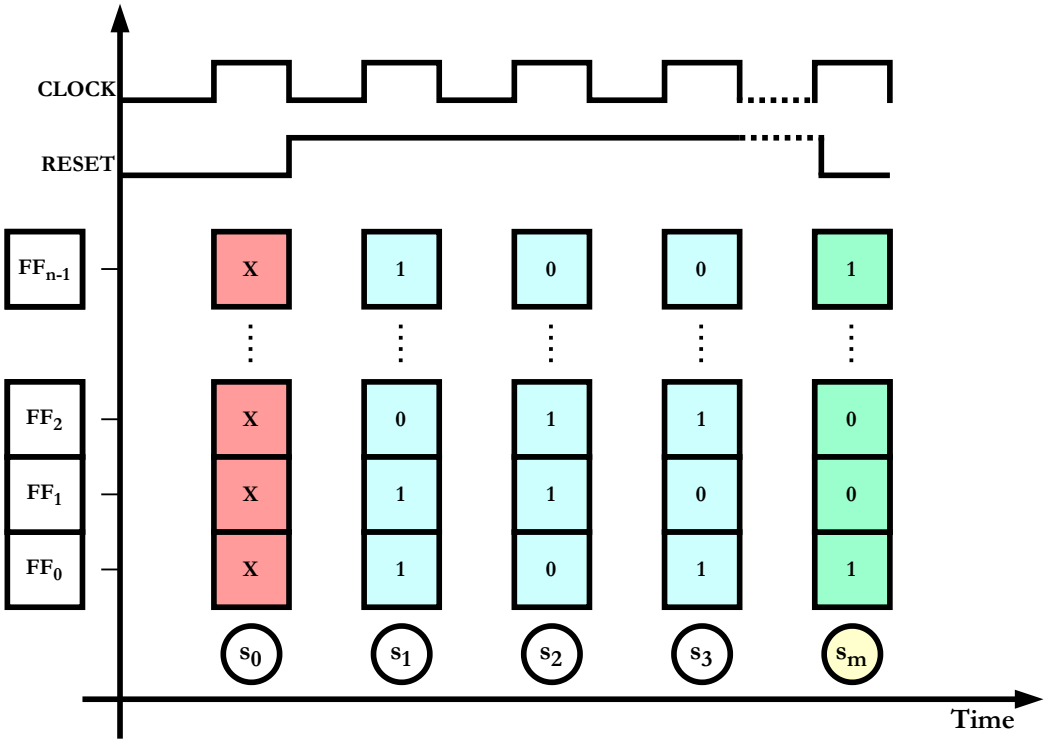

Fig. 1.11 Initial state extraction stemming from the synchronous RESET signal activation for arbitrary sequential circuit.

1.5 Thesis organization

This thesis offers formal methods-based solutions to the testing areas presented in Section 1.3. The solutions are based on two tools of the formal methods' set of utilities. Namely, Boolean Satisfiability and Bounded Model Checking [45]. The rest of this manuscript is organized as follows.

In Chapter 2 we focus on the BI test and while considering two stress metrics we provide methodologies to generate purely functional, stress-inducing stimuli that maximize them optimally. In Chapter 3 we focus on the problem of the identification of functionally untestable faults. We consider the stuck-at and the cell-aware fault models and while targeting processors as DUTs we propose methods that identify a considerable amount of such faults. In Chapter 4 we target in-field test and we propose BMC-based methods that are able to generate functional test code (e.g., in assembly) while considering permanent hardware faults and processors and GPUs as

the DUTs. Lastly, in Chapter 5 we draw some conclusions and reflect on the overall thesis contribution.

Chapter 2

Burn-In Test

2.1 Background

Various types of Burn-In (BI) testing rely on applying internal stress to devices under test (DUTs), ranging from simple stimuli at DUT inputs (Dynamic BI) to output capturing (Monitored BI) and full evaluation of DUT responses in conjunction with functional test patterns (Test-In BI). It has been demonstrated that such approaches significantly reduce test application times without compromising test reliability and quality [46].

However, generating appropriate stress stimuli for DUTs during BI testing requires careful consideration of two key factors. First, selecting a stress metric is crucial to quantify stress levels in absolute terms and establish clear thresholds. Inducing internal stress involves finding the right set of patterns to increase the switching activity (SWA) within the DUT, with the specific approach determined by the chosen stress metric [47].

The second key factor involves achieving sufficient internal stressing of DUTs. Common industry practice employs *structural* techniques, assuming the presence of Design-for-Testability (DfT) infrastructure on the DUTs (e.g., scan). However, stress generated through DfT mechanisms, if not used in a controlled and monitored way, may damage the devices since the induced stress is not guaranteed to match the operational constraints. Conversely, *functional* means regard "ad-hoc" generation of routines in a low-level language (e.g., assembly) to maximize the employed stress metric. In contrast to structural methods, the application of functional stress

stimuli does not pose any risk to the DUTs, as the circuit aging is coherent with the functional specifications.

Comparing the benefits of structural versus functional approaches, a recent study [48] highlighted distinct differences. Experimental results showed that stressing a 32-bit processor (used as a case study) with scan led to a uniform temperature increase across various modules. In contrast, applying functional stimuli (i.e., stress programs) at speed resulted in significantly higher thermal activity in the processor unit. The authors concluded that purely functional code applied at speed to the DUT outperforms DfT-based approaches like scan in terms of induced stress.

The subject of SWA maximization has been extensively studied in the past, especially in the domain of combinational circuits. What follows is an overview of previous works, distinguished into two categories: combinational and sequential circuits.

2.1.1 Previous works on combinational circuits

The reliability of integrated circuits is strongly linked with the problem of the identification of the maximum current consumption during the devices' testing phases. The research community has studied the problem extensively in the past. For instance, in [49] the authors address the problem of the maximum current estimation in MOS integrated circuits (ICs). They state that unrestricted voltage drops in the circuit's power and ground lines can lead a system to misbehave and cause a degradation in the switching speed. Such current estimates play a crucial role in the accurate timing analysis of the circuit and can further be used to enhance the reliability of the circuit's power and ground buses. In [49] the authors also propose a heuristic approach that can be applied to small combinational IC blocks to identify their maximum static and transient current by dividing the circuit into groups of combinational interconnections of logic gates and acting on each group independently. In [50], the authors approach the same problem, namely the estimation of the maximum current in CMOS ICs, by modeling it as an optimization problem. They search for stimuli (i.e., test patterns) that maximize the circuit's current value waveform. They further state that identifying such estimates plays a vital role in the reliability and performance of the circuits since such knowledge can be used to enhance the circuit's design, rendering it resilient to soft errors and overheating scenarios. In [51] the authors propose algorithms for

probabilistically estimating the average switching activity in combinational circuits of moderate size. They link the switching activity of a circuit with estimates that concern the circuit's power and heat dissipation.

The authors of [52] further highlight the importance that accurate power consumption estimates have for the performance and the reliability of VLSI chips. More importantly, they state that to obtain such accurate estimates, one has to identify a pair of **two** consecutive input patterns (test vectors) that induce as many logical switches within the device as possible. That is, to maximize the circuit's switching capacitance. Moreover, they state that the complexity of such an exhaustive search for the identification of the appropriate pattern pair on a combinational circuit with n primary inputs is $\mathcal{O}(4^n)$. Their approach to compute such power estimates is based on an automatic test generation technique, while a Monte Carlo methodology is also described. Although the proposed method is effective, it is applied to relatively small-sized combinational circuits. Likewise, in [53], the authors approach the problem of power dissipation in combinational CMOS ICs by relying on formal methods. They model the circuit power dissipation as a Boolean function of the circuit's primary inputs. They also link the power estimation problem to the identification of the appropriate pair of two consecutive test vectors that maximize the gate switching of the device. Their solution reduces the problem to a weighted max-satisfiability (MaxSAT) problem. But, once again, the method was applied to small combinational circuits due to the high complexity imposed to obtain the circuit's objective function and optimize it.

Overall, maximizing the SWA of a circuit can be proven beneficial from a designer's perspective since it allows for extracting important information that enhances the overall reliability of the devices by fine-tuning specific design parameters. However, SWA maximization can be advantageous in the context of device testing as well. During the multiple test steps that are adopted during the device manufacturing, it may happen that faults do not manifest themselves during testing and escape the whole set of test steps. These latent defects are the prime suspects behind the Infant Mortality behavior, which is resolved via BI. In [54] the authors present a probabilistic approach (i.e., random excitation of internal nodes) to generate input patterns that maximize the SWA of a combinational block further to achieve the maximization of power dissipation during BI testing. In [55] the author relies on a genetic algorithm to develop a method to maximize a combinational circuit's SWA and maximize the circuit's heat dissipation during BI testing.

Having acknowledged the importance and the benefits that stem from the SWA maximization on ICs and the acquisition of switching information of a circuit in general, the researchers focused deeper into the SWA maximization problem and proposed various methodologies to solve it effectively. In [56] the authors present a technique to extract estimates about the maximum switching activity of a combinational circuit's nets while also considering delays within the circuit. Although the method is applied solely to combinational circuits the authors claim that such an approach can also be employed in sequential circuits by isolating the combinational blocks and applying the method in a divide-and-conquer fashion. In [57] a methodology based on automatic test pattern generation (ATPG) tools is presented to identify the pair of test vectors that maximize the weighted SWA of a combinational IC. This is achieved by appropriately modifying the gate-level description of the circuit and generating vectors while targeting a selected set of faults on the modified netlist while considering a variable delay. In [58] the authors present a simulation-based approach for identifying the appropriate combination of two test vectors for combinational ICs that maximize the switching activity and thus the circuit's power consumption. Although highly effective, the generated pairs of test vectors for a combinational IC block with many inputs cannot be proven to be the absolute best due to the exhaustive simulations needed for an exponentially growing number of vectors. In [59] another approach based on formal methods is presented targeting combinational circuits. The authors present an integer linear programming approach that utilizes satisfiability (SAT) solvers as an underlying technology to effectively compute the maximum weighted switching activity of combinational ICs that can be used to derive information about the circuits' power dissipation.

2.1.2 Previous works on sequential circuits

Regarding sequential circuits, and specifically processors (or processor cores), the authors of [60] propose a method based on formal methods for the generation of patterns that maximize the SWA on various parts of the processor in a uniform manner by isolating the functional units of the core and encoding them in conjunctive normal forms (CNFs). By considering each unit separately, they define simpler functional constraints (i.e., transitions maximization) per unit flexibly and evaluate each formula's satisfiability individually. This enables the construction of functional

stimuli that can be used during BI to assist the aging process of various parts of the circuit.

In [61] the authors present a methodology based on evolutionary techniques that builds stressful assembly programs for a target processor by extracting characteristics (i.e., code segments) that were found to be causing high switching activity when executed. These segments, in turn, are used to compose a final stress program that gets refined during the evolution process and can induce high stress within the functional units of the core. In [13, 14] we present another method based on evolutionary techniques, which is able to generate from the ground-up (i.e., without dependencies to pre-existing code) assembly programs that induce high stress amounts within specific units of the processor in a repeatable manner. Although experimental results prove the method's effectiveness, there is no guarantee that the generated programs are the best possible.

In [48] the authors, while targeting a 32-bit processor intended for mission-critical usage, present a comprehensive methodology and propose metrics for the comparison and the evaluation of stress procedures that are applied to the circuit during BI. The processor is used in two variants in their case study. On the one hand, the core is equipped with DfT infrastructures (e.g., scan); on the other, it is not. Their experimental results demonstrate that while the processor with scan is being stressed, a uniform elevation of its temperature is observed inside the various modules. On the other hand, when functional stimuli (i.e., stress programs) are applied at-speed to the processor, a much higher thermal activity is observed in the respective processor unit. The authors conclude that a purely functional code applied at-speed to the DUT has a notably better performance in terms of induced stress and, thus, renders it more suitable than DfT-based approaches such as scan.

2.2 Constant & repeatable SWA maximization

To summarize, the SWA maximization problem has been approached with various methodologies in the past, both for combinational and sequential circuits. In this section, while covering the case where the DUTs are sub-modules of a pipelined microprocessor, the aim is to present a methodology based on SAT solving for generating stimuli that maximize the repeatable constant switching activity within the targeted module of the core. In this case, the generated functional stimuli

correspond to assembly programs. The proposed algorithm optimally solves the problem using formal methods. Although the considered DUT is a microprocessor, there are no limitations to the method's applicability method to other sequential circuits.

2.2.1 Problem definition

The goal is, given the gate-level description of a pipelined processor as input and a sub-module of the processor as a stress target, to generate a functional sequence (i.e., assembly instructions) that induces the highest possible switching activity within that target. More precisely, to identify a pair of **two** instructions (or vectors) (I_1, I_2) that can maximize the repeatable constant switching activity of a certain processor module (i.e., the processor module for which the test engineer is required to generate stress stimulus) when executed in sequence. Specifically, the sequence induces the maximum possible *constant* switching activity within the module. This means that each instruction of the two-vector pair is able to maximize the number of logical switches performed from the nets of the targeted module from High to Low (HL) and from Low to High (LH) manner when they are applied in sequence. Low corresponds to the logic value 0 and High to the logic value 1. More specifically, that means that if the application of I_1 forces n nets to toggle, then I_2 forces the same n nets to make the inverse transition.

Note that the objective is to maximize the constant switching activity, i.e., induce the same stress amounts in the processor unit for an arbitrarily long time period (in clock cycles); hence, it is required for the generated sequence of instructions to be a *repeatable* one. Let us assume that for a specific processor module T the optimal sequence $\widetilde{s}$ of two instructions has been found. It is further assumed that the whole processor has been initialized properly, either via the activation of its `RESET` signal or via the execution of a synchronization sequence. The predefined initialization phase drives the core to a well-defined and legal functional state s^a. The first of the two stress-inducing instructions of the sequence $\widetilde{s}$, drives the processor to a new state s^b. Lastly, the second instruction drives the processor to the same initial state s^a in order for the generated sequence $\widetilde{s}$ to be a repeatable one. By satisfying the aforementioned equivalence that guarantees a repeatable sequence to be generated, one can maintain a maximum gate switching activity within the processor module T for an arbitrarily long period of time.

Assuming that the optimal stress-inducing pair of instructions ($\widetilde{s}$) has been generated, it can be used to stress the processor module T (e.g., to create a hot spot within the core) in a constant manner. This can be achieved by repeating the sequence $\widetilde{s}$ for as many times as required (e.g., $\widetilde{s}\widetilde{s}...\widetilde{s}$). After the repetition of the generated pair of instructions for a sufficiently large number of times, an unconditional jump instruction can be issued to transfer the code execution back to the start of the stress segment. The loop can be forcibly stopped, e.g., by issuing an interrupt call. the whole process can be interpreted as a deterministic finite automaton (DFA) as depicted in Figure 2.1.

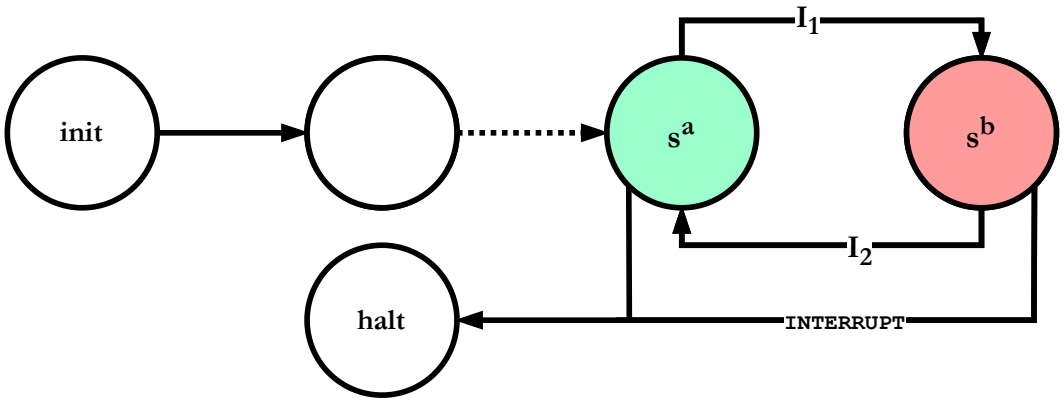

Fig. 2.1 DFA representation of constant and repeatable switching.

2.2.2 Stress evaluation metric

In order to measure the effectiveness of the generated instruction sequences in terms of stress induced to the target processor module, we use the average induced stress percent metric. Given that the generic target processor module T consists of m nets and the generated sequence $\widetilde{s}_n$ is composed of n instructions, we calculate the average induced stress percentage as:

$$\overline{\text{stress}}_{\%} = \frac{\sum_{i=1}^{m}[HL(i)+LH(i)]}{n \times m} \times 100 \tag{2.1}$$

The numerator of the fraction represents the total amount of HL and LH transitions that were performed by every net out of the m existing in the target processor module T. The denominator of the fraction represents the maximum achievable value, corresponding to the case where every net of the module makes a transition

(either from HL or LH) when every instruction of the sequence $\widetilde{s}_n$ is executed. For example, assuming a sequence of 2 instructions, then the maximum achievable value would be $2 \times m$, where the first instruction forces all nets to switch, and the second instruction forces the same m nets to perform the inverse transitions. When considering a pipelined processor, in the first phase we assume that every instruction of the sequence is executed by the target module over 1 clock cycle, unless it is stated otherwise. We will relax this assumption later. Obviously, the denominator value of the fraction should be interpreted as a theoretical maximum, which can never be achieved in practice, e.g., due to the presence of uncontrollable lines within the module T. But most importantly, it is clearly not given that all nets can be toggled in the same clock period in a functional manner.

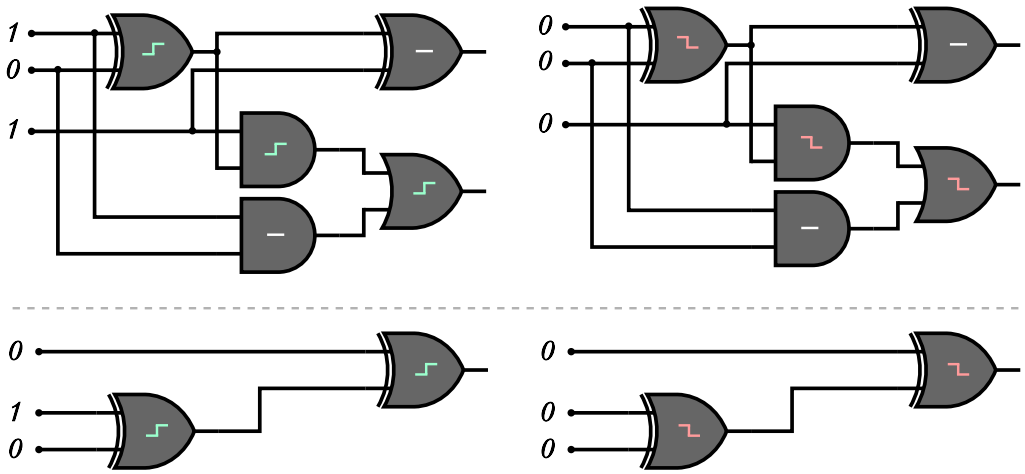

Fig. 2.2 1-bit full adder (top) and 3-bit odd parity checker (bottom) maximum switching for sequences $\widetilde{s}_{FA}$ and $\widetilde{s}^{odd}_{parity}$, respectively

For example, let us consider the two combinational circuits depicted in Figure 2.2: a 1-bit Full-Adder (FA) at the top and a 3-bit message odd parity checker below. Further, let us assume that they are initialized in such a manner that all nets hold the logic value 0. For the case of the FA one optimal sequence that maximizes the consecutive gate switching is $\widetilde{s}_{FA} := < I_1, I_2 > = < 101, 000 >$. This sequence forces 3 out of 5 nets to toggle, and there exists no pair of vectors that when applied to the FA circuit's inputs can force a higher number of HL and LH transitions than 3. On the other hand, considering the parity checker, there exists a sequence that forces all nets to toggle e.g., $\widetilde{s}^{odd}_{parity} := < I_1, I_2 > = < 010, 000 >$.

2.2.3 Search space analysis

The identification of the instruction pair that leads to the repeatable constant maximization of the SWA of a processor module is a non-trivial task, with an intricacy that depends on the size and characteristics of the sub-module we are targeting.

For example, let us consider as a target the 32-bit adder of the processor's arithmetic and logic unit. In this case, the search space of the problem can be shrunk significantly, since we know beforehand which instructions have to be considered in the search, namely, the `add`/`sub` instructions of the processor's instruction set architecture (ISA). Thus, the problem can be reduced to the identification of the appropriate pairs of operands that maximize the SWA of the adder unit.

On the other hand, assuming that the stress target is the processor's instruction decode unit then the algorithm should not only consider potential operands like in the case of the 32-bit adder, but a combination of instructions and operands in order to generate a stress effective instruction sequence. In fact, the search space should include the majority of the set of instructions supported by the processor's ISA. So, for the case of the decoding unit, the task is harder since the search space is considerably larger than in the case of the adder.

For example, assuming that there is only one integer addition assembly instruction in the ISA (to simplify) then the search space (S_{adder}) can be described as a set given by the Cartesian product:

$$\begin{aligned} S_{\text{adder}} &= S_{\text{instruction}_1} \times S_{\text{instruction}_2} \\ S_{\text{instruction}_1} &= I_{\text{N-bit}} \\ S_{\text{instruction}_2} &= I_{\text{N-bit}} \end{aligned}$$

where $I_{\text{N-bit}}$ is the set of all N-bit integers (e.g., if the processor supports 32-bit registers then this would be the set of all 32-bit integers). In the case of the decoder being the stress target, then the search space (S_{decoder}) can be described as a set given by the Cartesian product:

$$\begin{aligned} S_{\text{decoder}} &= S_{\text{instruction}_1} \times S_{\text{instruction}_2} \\ S_{\text{instruction}_1} &= S_{\text{ISA}} \times S_{\text{op}} \\ S_{\text{instruction}_2} &= S_{\text{ISA}} \times S_{\text{op}} \end{aligned}$$

where S_{ISA} is the set of all instructions supported in the processor's ISA and S_{op} is the set of all operands that are supported by every instruction of the ISA. It is clear that the size of the set S_{decoder} is much greater than the size of the set S_{adder}.

Thus, as mentioned earlier, one can safely assume that the scalability of the method depends on the complexity of the targeted processor module. Also, the structure of the targeted module has an impact on the overall run-time of the method.

2.2.4 Proposed method

The problem is modeled as a MaxSAT problem. The reason for this selection is the fact that the problem of the maximization of the SWA of a processor's module can be seen as an optimization problem. The core idea is to enforce constraints that encode differences (transitions) between two consecutive clock cycles and thus, maximize the SWA over those two cycles. Then, the corresponding MaxSAT solver evaluates these constraints in order to determine the satisfiability of the CNF formula. Lastly, if the formula is satisfiable it is possible to extract the two input vectors that stress the maximum by examining the model identified by the solver.

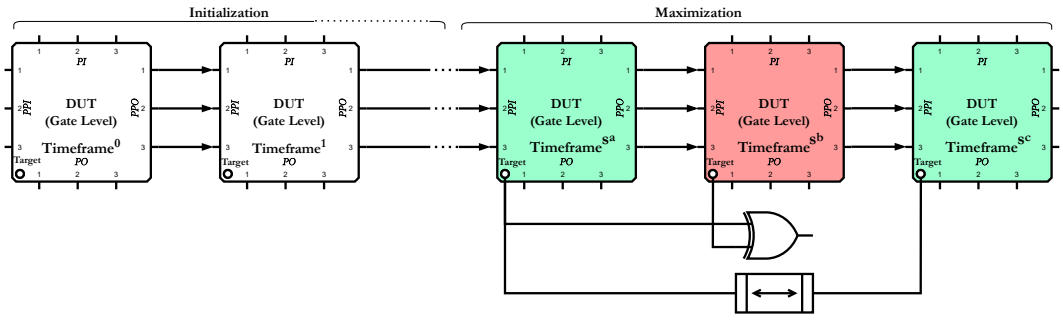

Fig. 2.3 MaxSAT model.

One crucial parameter to the accuracy and effectiveness of the method is the unrolling depth, i.e., the total number of times the circuit has to be replicated. As shown in Figure 2.3 (which can be correlated with Figure 2.1) the unrolling depth is the sum of the initialization phase timeframes plus the maximization phase timeframes. We have previously explained that we begin from a valid and well-defined initial state, i.e., there are no `DON'T CARE` values in the pipeline and the processor state is a functional one. The initialization step is vital since the underlying solver is ignorant about the architectural constraints of the pipeline (e.g., a boot

address must be set before starting the execution of code in the program counter bits). Thus, if the initialization step is not done, in an attempt to satisfy the formula the solver may end up with an assignment, which although satisfying the CNF, forces the processor to a non-functional state. In our case, we assert the processor `RESET` signal and unroll the circuit for the minimum number of times that is required for the effects of the signal to take effect and propagate to the whole pipeline. This number is architecture-specific, i.e., for a different processor a different number of initialization timeframes may be required, and this information can be derived either from the specification of the architecture (if available) or by inspecting the RT-level description of the DUT. Following the initialization phase, we must define the duration of the maximization phase, i.e., the three states encoded by the rightmost timeframes of Figure 2.3. For most of the processor units, the duration (in terms of clock cycles) for each instruction is one clock cycle. Yet, there are specific cases (e.g., for the multiplication unit) where the instruction remains active for more than one clock cycle. Specifically, the exact number of timeframes required for the maximization phase can be derived from the Equation (2.2) below:

$$\mathrm{TF}_{\max} = (2 \times \mathrm{duration}) + 1 \tag{2.2}$$

As mentioned earlier, the parameter duration is architecture-specific and represents the number of clock cycles that the instruction requires to be executed in the module of the pipeline that we intend to stress. Hence, we need twice the duty cycle duration of the module in order to fully account for the states TF^{s^a} and TF^{s^b} (see Figure 2.3), respectively. Lastly, we need one extra timeframe in order to force the equivalence of the resulting state after the execution of the second instruction of the sequence with the initial state, i.e., $\mathrm{TF}^{s^c} \equiv \mathrm{TF}^{s^a}$.

Up until this point, we have enforced the constraint for the activation of the `RESET` signal in the very first timeframe of the unrolled circuit. In general, the constraints are encoded as clauses and are appended to the circuit's CNF formula. In the MaxSAT terminology, two types of clauses exist. The so-called *hard* clauses, which must all be satisfied in order for the CNF to be evaluated as satisfiable, and the *soft* clauses, whose satisfiability does not affect the satisfiability of the CNF formula. Each soft clause is also correlated with an integer weight value and so, the goal of the MaxSAT solver is to find an assignment that satisfies all the hard clauses of the CNF formula while maximizing the sum of weights for the satisfied soft clauses at

the same time. A uniform weight distribution is used and thus, each soft clause is associated with the same integer value as a weight.

In Figure 2.3 we have an abstract representation of the proposed MaxSAT model. The initialization phase consists of the encoding of a hard clause on the core's RESET signal that corresponds to the signal activation. For every one of the following timeframes, clauses are inserted for the RESET signal in order to render it inactive and thus, to prohibit the solver from using it for the sensitization of lines in the targeted module. Although it is true that the activation of the RESET signal in certain cases can cause a significant amount of concurrent transitions, such behavior is abnormal and thus it must be prohibited in order to enforce a functional scenario within the pipeline. Moreover, we have previously introduced the idea of repeatable instruction sequences that maximize the constant SWA of a given module. In order to guarantee repeatability, every net of the module must have the same value in TF^{s^a} and in TF^{s^c}. For this reason, every literal that encodes a net of the targeted processor module during TF^{s^a} ($l^{s^a}_{net_i}$), along with every literal that encodes a net of the targeted module during TF^{s^c} ($l^{s^c}_{net_i}$) are linked together by inserting the following logic implications as hard clauses to the CNF formula, which guarantee their equivalence during these two states:

$$\omega^{hard}: \; l^{s^a}_{net_i} \longleftrightarrow l^{s^c}_{net_i} \equiv (\neg l^{s^a}_{net_i} \vee l^{s^c}_{net_i}) \wedge (l^{s^a}_{net_i} \vee \neg l^{s^c}_{net_i})$$

Hence, given that the solver finds a satisfying assignment for the CNF formula, it is guaranteed that all of the nets of the module will hold exactly the same logic values during TF^{s^a} and TF^{s^c} and thus, the generated sequence is indeed repeatable. Besides the states' equivalence, we also have to encode the corresponding switching of the module's nets between the timeframes TF^{s^a} and TF^{s^b}. This switching corresponds to soft clauses, i.e., we request from the solver to satisfy as many switching transitions as possible. For every net i of the targeted processor module we encode an XOR gate on the CNF as a hard clause, whose inputs are the corresponding literals for the net i in TF^{s^a} and TF^{s^b}, respectively. In order for the switching requirement to take effect, we further encode soft clauses ($l^{\oplus}_{\text{diff}_i}$) that correspond to the output of the XOR gate and require it to be 1 as shown below:

$$\left.\begin{array}{lll}
\omega^{hard}: & (l^{s^a}_{\text{net}_i} & \vee \neg l^{s^b}_{\text{net}_i} \vee l^{\oplus}_{\text{diff}_i}) \wedge \\
& (\neg l^{s^a}_{\text{net}_i} & \vee l^{s^b}_{\text{net}_i} \vee l^{\oplus}_{\text{diff}_i}) \wedge \\
& (l^{s^a}_{\text{net}_i} & \vee l^{s^b}_{\text{net}_i} \vee \neg l^{\oplus}_{\text{diff}_i}) \wedge \\
& (\neg l^{s^a}_{\text{net}_i} & \vee \neg l^{s^b}_{\text{net}_i} \vee \neg l^{\oplus}_{\text{diff}_i}) \\
\omega^{soft}: & (l^{\oplus}_{\text{diff}_i}) &
\end{array}\right\} l^{s^a}_{\text{net}_i} \oplus l^{s^b}_{\text{net}_i} \text{ in CNF}$$

The first four clauses correspond to the Tseitin transformation of the XOR $(l^{s^a}_{\text{net}_i} \oplus l^{s^b}_{\text{net}_i} \Leftrightarrow l^{\oplus}_{\text{diff}_i})$ gate in order to be represented as a CNF. The very last unit clause requests that the output of the XOR gate must be evaluated as 1. Thus, if satisfied it corresponds to a difference between the two literals that encode net i in TF^{s^a} and in TF^{s^b} i.e., the corresponding net performed an HL or an LH transition. Due to the state equivalence enforced by the aforementioned hard clauses, it is redundant to enforce another set of switching constraints for the target module's nets between TF^{s^b} and TF^{s^c} since it is already implied due to the equivalences enforced for TF^{s^c} and TF^{s^a}.

When it comes to the application of functional constraints (e.g., ISA encoding, memory mapping, program counter limits), a VCM circuit is employed which is responsible for embedding them in the CNF in the manner presented in Section 1.4.2.

Assuming that the solver managed to satisfy the CNF, we can now extract the input vectors in the corresponding timeframes and compose the stress-inducing sequence. In order to do so, we need to identify the literals that are used to encode the PIs of the module and extract the 0/1 logic values from the found model. For instance, considering the FA circuit of Figure 2.2, we can extract the first vector $v_1 := <1, 1, 0>$ from the first timeframe and the second vector $v_2 := <0, 0, 0>$ from the second timeframe by examining the assignments of the corresponding input literals. In the general case, we can extract and disassemble the N-bit instructions from the respective pipeline instruction register and use it in combination with the previously decoded vectors in order to compose an assembly program that effectively stresses the target processor module. In the general case, one can always use the instruction bus of the processor as a probing point and disassemble the N literals back to N-bit instructions (while always respecting the bit order).

2.2.5 Experimental results

The proposed method has been applied to two single-issue, scalar, pipelined RISC processors, namely to the OR1200 [62] and to the RISC-V processor RI5CY [63]. The OR1200 is a 32-bit scalar RISC processor using the Harvard micro-architecture. It is mainly intended for embedded, portable and networking applications. The processor RI5CY is a 4-stage in-order 32-bit RISC-V processor core. The ISA of the processor was extended to support multiple additional instructions including hardware loops, post-increment load and store instructions, and additional ALU instructions that are not part of the standard RISC-V ISA. RI5CY has become a popular core for a huge variety of applications and especially for Internet-of-Things designs. Both processors' RT-level descriptions were synthesized using the Silvaco 45nm Open Cell Library [64] via Design Compiler by Synopsys.

The experiments were performed on a machine using an Intel i9-9900 processor running at 3.10GHz. For every sequence generated for the targeted modules within the considered processor cores, a logic simulation environment was set up using QuestaSIM by Mentor Graphics. Specifically, during the simulation of the generated stress-inducing sequences, the targeted processor module was isolated and a toggling evaluation was performed for every clock cycle. The reported results from every processor module were then aggregated and post-processed in order to calculate the stress induced via Equation (2.1).

Table 2.1 Experimental Results

Processor	Stress Program Generation Approach	Average Induced Stress			CPU Generation Time		
		Adder	Decoding Unit	Load Store Unit	Adder	Decoding Unit	Load Store Unit
OR1200	MaxSAT	82%	91%	65%	3^{sec}	15^{min}	8^{sec}
	Stuck-At Test Program	24%	44%	57%		-	
RI5CY	MaxSAT	76%	36%	34%	5^{sec}	14^{min}	7^{sec}
	Stuck-At Test Program	73%	30%	32%		-	

For both cores, the units that we targeted to generate stress-inducing stimuli are: (i) the 32-bit Adder of the processors' ALU (ii) the Instruction Decode unit (iii) the Load and Store unit.

In order to have a mean of comparison with our results, both cores we use for hand-written test programs that reach a high stuck-at fault coverage (>85%). These test programs are simulated in the same manner as the generated sequences and their effects on the targeted processor units in terms of induced switching activity

are investigated clock cycle per clock cycle. The results are then post-processed in pairs of two consecutive instructions in order to effectively compare them with the instruction pair of the MaxSAT-generated sequences. The highest stress-inducing pairs are considered in the comparison presented in Table 2.1.

Experimental results show that for both processor cases, for all considered functional units, the proposed method generated sequences that outperform the respective segments found in the functional stuck-at test programs. Most importantly, there is a notable difference in the case of the Decoding Unit and the Load and Store Unit of the RI5CY core. One can see that in fact, the maximum sustainable switching activity is lower than that of the respective units in the OR1200 processor. Although, one cannot directly compare two completely different (structurally) circuits this deviation is fully justified by the complexity imposed by the respective units in the case of the RI5CY processor. It is safe to assume (as can also be seen in Figure 2.2) that as the complexity of the module increases, the percentage of possible concurrent logical switches within a circuit decreases, i.e., they are related in an inversely proportional manner. Regarding the two units, in the case of the RI5CY processor, it is true, that they cover a significantly larger set of instructions than those of the OR1200 processor. Lastly, it can be seen that there exists no clock cycle, during the application of the stuck-at test program on the RI5CY core, in which a higher number of concurrent toggling was found than the number of concurrent toggling induced by any of the generated instructions.

Table 2.2 Supplementary Comparisons

Generation Approach	**Average Induced Stress**			
	Adder	Multiplier	Decoding Unit	Load Store Unit
MaxSAT	82%	62%	91%	65%
Evo	61%	55%	63%	58%
Test Program	24%	6%	44%	57%

In order to provide the reader with a further comparison, we have also developed stress programs for the OR1200 processor (considering the same modules) via μgp [65], which is an evolutionary optimizer that was developed to produce assembly programs maximizing a given fitness function for a variety of processors. The implemented evolutionary algorithm is similar to the one described in [13, 14]. We also considered the special case of the 32-bit multiplier unit. The results are reported in Table 2.2 below.

We can clearly see that although the evolutionary-based stress program generation yields sequences of instructions that are better in terms of induced stress in the considered processor units than the test programs, they are sub-optimal solutions and thus, the proposed method outperforms them in all cases. For the case of the 32-bit multiplier though, the required CPU time was quite large (approximately 85 hours), and considerably greater than for the rest of the considered units. It is well known that the arithmetic multiplier circuits represent an arduous task for formal methods [66, 67]. In order to overcome the complexity imposed by the multiplier circuit, a heuristic sampling approach was employed. Specifically, instead of generating a switching constraint for every net of the multiplier unit, we sampled a portion of high fan-out nets of the circuit and enforced switching constraints only on them. This implies that since the circuit has not been fully considered (in terms of nets being constrained for maximization), the generated solution is still a sub-optimal one, although better than the one produced with other approaches.

2.3 2-Multi-Point SWA maximization

In this section, while also covering the case where the DUTs are sub-modules of a pipelined microprocessor, the aim is to present a methodology, based on bounded model checking (BMC), for generating stimuli that maximize the SWA of a targeted module of the core while considering topological, layout information about neighboring nets of the design. That is, the DUT is dissected into neighborhoods consisting of 2 nets, and for each group, we aim to generate a functional sequence (i.e., an assembly program) that enforces all possible transitions. IC manufacturers steer towards the usage of such stress-inducing approaches due to their effectiveness in generating targeted heat gradients within the DUTs. This is achieved by incorporating information derived from the layout of the designs. The proposed algorithm optimally solves the problem by relying on formal methods. Although the considered DUT is a microprocessor, there are no limitations to the applicability of the method to other sequential circuits.

2.3.1 Problem definition

Given the gate-level description of a processor and assuming that the layout L of the target processor's sub-module M is available, then it is possible to know the pairs of nodes which are neighboring, i.e., they are placed within a minimum specified distance from one another. Given a list that contains pairs, with each pair consisting of **two** neighboring nodes, the goal is to generate functional stimuli (i.e., snippets of assembly code) that each force the maximum switching activity for a pair of nets within the targeted unit. In other words, for every pair of two nodes, to generate a sequence of instructions that is able to induce both transitions to each pair starting from a well defined initial state. Furthermore, given that the resources (e.g., tester memory) are limited during the BI test, the case where the generated instruction sequence length is minimal is considered. This means the transitions of the nodes must happen over a short period of clock cycles.

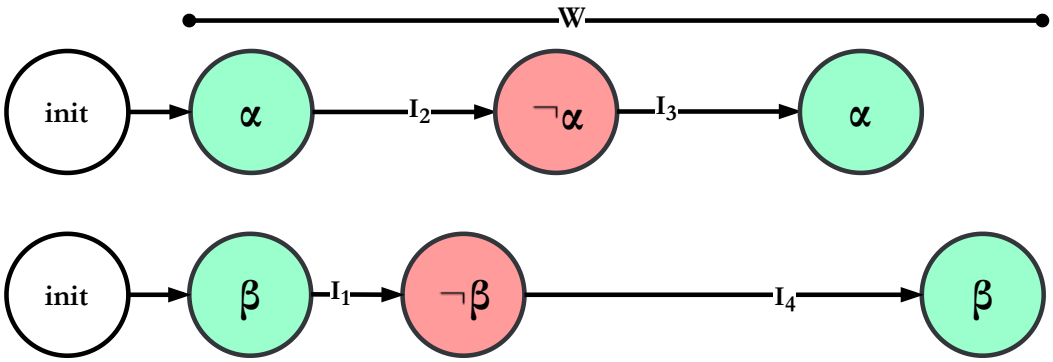

Fig. 2.4 DFA representation of 2-multipoint switching for a neighborhood of 2 nets.

The process can be visualized as a DFA as shown in Figure 2.4. Considering a pair of two neighboring nodes (α, β), they are initially assigned a logic value stemming from the initialization phase. The initialization phase can be either the result of an initialization sequence or the state that follows the activation of the global `RESET` signal of the processor. Thus, after both nets are assigned a logic value $\alpha, \beta \in \mathbb{B}$, the application of the optimal switching sequence begins with the execution of instruction I_1 causing the second net to toggle. Following that, the execution of the second instruction of the sequence I_2 forces the first net to toggle, and consequently, instruction I_3 sets it back to the logic value imposed by the initialization phase. Lastly, the second net follows which is also set to its initial logic value by the instruction I_4.

In this scenario, which presents the ideal case, the neighborhood has been fully sensitized since both nets toggled twice. Also, as it is required the sequence of transitions occurs over a small time window (W). This parameter is crucial given that for each net neighborhood of the stress target a sequence will be generated. Thus, the total memory footprint for storing the stress-inducing stimuli has to be minimal. It is of course possible for both nets to perform a transition at the same time by the execution of a single instruction. Clearly, it is also not given that this switching pattern will be possible for every pair of nodes, due for example to uncontrollable lines or to the inability of both nodes of a pair to perform both transitions from their initial states as shown in Figure 2.4.

2.3.2 Stress evaluation metric

When it comes to the evaluation of the stress induced in the DUT by the application of stress stimuli, the test engineers may resort to *single-point* metrics which evaluate how many times each net of the DUT toggle. Such metrics are useful for computing extended statistics for the DUTs such as the number of times a net toggles during the test stimuli application or the average toggling frequency of the circuit. However, the proposed method is based on the concept of the *multi-point* switching metric [47].

In order to compute the effectiveness of a generated instruction sequence (*seq*) of a given pair (α, β) of nets, the stress efficiency metric (SE) is introduced which is defined as follows:

$$\mathrm{SE}(seq) := \sum_{i \in \{\alpha,\beta\}} T(i, seq) \mid_{init} \tag{2.3}$$

where T is a function that computes the number of states (transitions) reached from the initial *init* state of node *i* during the application of the sequence *seq*. Thus, the range of values that can be held by the function T used to compute Equation (2.3) is $\{0,1,2,3,4\}$.

For every pair of nodes, we are interested in generating functional sequences for which the SE function gives the maximum value (4), meaning that both nodes were forced to perform both transitions. Note that our metric differs from the multi-point stress metric introduced in [47]. Namely, we do not only consider the case where the nodes of each pair hold opposite initial values. Instead, we generalize by considering

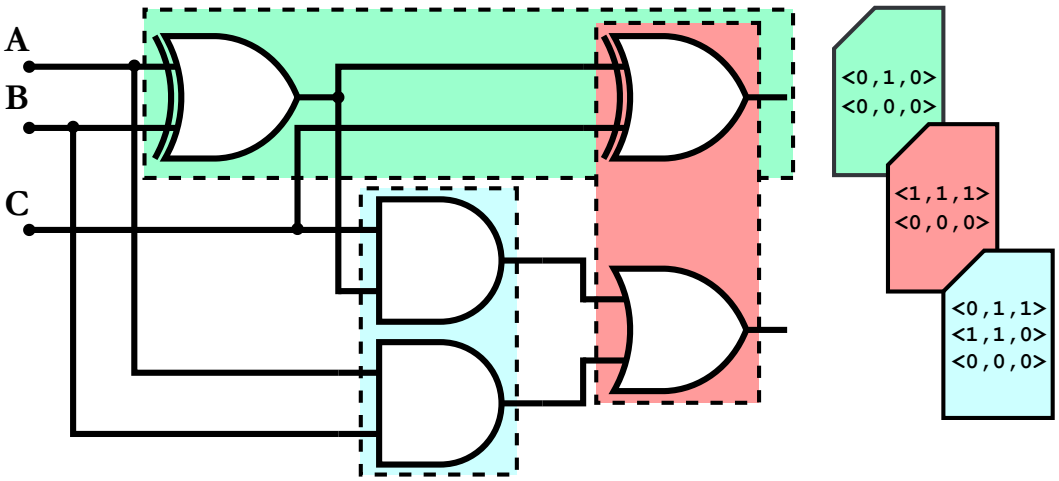

Fig. 2.5 Maximum stress efficiency sequences for a net pairing on a full adder.

whichever initial value for both nodes. In Figure 2.5, a full adder is presented as an example that has been dissected into neighborhoods of size 2. Then, assuming that all gates' outputs are initialized to logic 0, an optimal sequence that enforces both transitions starting from the initial state is computed. Note that as mentioned earlier, for the group depicted with green color and the group depicted with red color, it is possible with just 2 functional vectors to obtain a SE of 4. This is the optimal scenario as not only is the maximum SE value achieved but also with the minimal amount of vectors. However, clearly, this is not always possible. As can be seen, with the group depicted in blue, the maximal SE value of 4 can be achieved with a minimum of 3 functional vectors. Lastly, by considering Equation (2.3) as an objective function, the goal can be defined as an optimization problem:

$$\forall\ pair \in L : \max_{seq_{pair}} \{SE(seq_{pair})\}|_{\#seq \leq W}$$

2.3.3 Search space analysis

Similarly to the explanation provided in Section 2.2.3, the problem's search space depends once again on the characteristics and also the size of the sub-module of the processor that we are considering as a stress target. The size of the stress target affects the total number of nets, and thus the total amount of neighborhoods for which an optimal stress-inducing functional sequence must be generated. Thus, the size of the search space increases proportionally with the size of the stress target.

Furthermore, the complexity of the targeted processor module affects the search space. For instance, once again by considering as a stress target the processor's 32-bit adder, we know beforehand the assembly instructions to be used. Hence, the search space is limited to the total combination of operands. Conversely, when considering the processor's decoding unit as a stress target, then the search space is also affected by the total combination of opcodes (i.e., assembly instructions) on top of the combination of operands. One key parameter that differentiates this approach from the approach presented in Section 2.2, is the usage of the parameter W. As we have seen in the example of Figure 2.5, the total number of instructions may exceed two in order to obtain the optimal SE from a functional sequence.

Hence, the search space for the optimal sequence required to stress a pair of the modules can be approximated by the Cartesian product:

$$S_{\text{seq}} \leq \prod_{i=1}^{W} S_{\text{instruction}_i}$$

The reader should note that the set $S_{\text{instruction}_i}$ differs according to the functional unit we target. It holds that $S^{\text{adder}}_{\text{instruction}_i} \ll S^{\text{decoder}}_{\text{instruction}_i}$ since the decoder is responsible for handling every instruction of the processor's ISA, which means that the search space for the case of the decoder is much larger than the search space for the case of the adder.

2.3.4 Proposed method

The method used to model the problem is BMC. The reason for this is due to the nature of the maximization problem. For an arbitrary group comprised of 2 nets, the requested sequence must be generated at most within a specified window of timeframes (W) rather than a strict, predefined static amount of timeframes. This means, that the generated process may be generated, e.g., in two or in three clock cycles (as shown in the example of Figure 2.5). This behavior points towards the usage of incremental SAT solving, hence the usage of a bounded model checker.

As explained in Section 1.4, During bounded model checking, we embed a desired property (P), i.e., the desired switching activity is reached, into the CNF formula by defining an upper bound (k) we ask the underlying solver to check whether this property can be reached up to a maximum depth k starting from a well

defined initial state (I) by following a transition relation (T). The general CNF formula for an arbitrary step k is built as:

$$\mathrm{CNF}_k = I^0 \wedge \bigwedge_{i=0}^{k-1} T^{i \rightarrow i+1} \wedge P^k \tag{2.4}$$

Note, that the so-called classical BMC aims to show that an invariant (safety property) P can be falsified i.e., $\neg P$ can be reached in at most k steps [45]. We use the equivalent $\neg P^k := P^k$, where P is equal to "the target switching activity is reached".

During each step, a call to an SAT solver is issued. If CNF is satisfiable then this means that the property is satisfied. Otherwise, no conclusion is drawn and the next iteration begins. If however, during the last step (i.e., maximum unrolling depth k is reached) the CNF is proven unsatisfiable then no valid assignment exists that can satisfy the desired property within k steps.

Considering the problem at hand, the maximum depth k is the parameter W of the method. As an initial state I, the process stemming from the activation of the reset state of the processor is considered and thus what is missing is the definition of the property. What we want to embed as a property to the BMC problem is the following: "For a pair of neighboring nets (α, β), does there exist a sequence of input vectors that can force both transitions starting from I?". However, (i) a mechanism of encoding this property into CNF must be formulated and (ii) it is not given that the maximum toggling i.e., that both nets can switch twice is given. Let us consider the sequential circuit of Figure 2.6 as an example of our DUT.

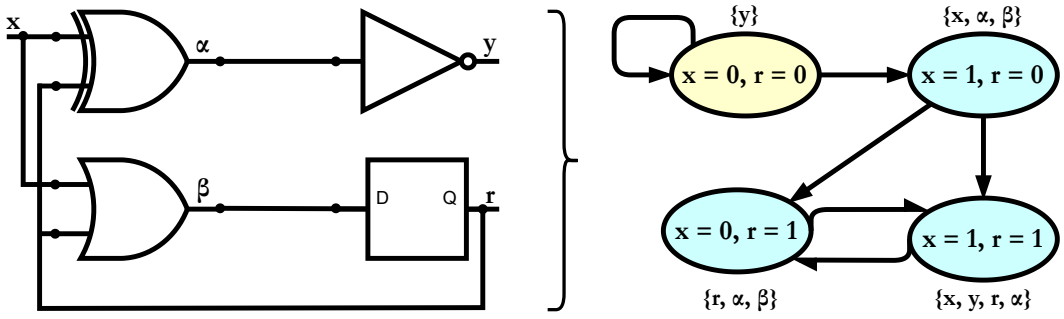

Fig. 2.6 Kripke structure for example sequential circuit as DUT.

The group of nets to be stressed is α and β. The initial state for the DUT is considered to be r = 0. Hence, the Kripke structure [68] of the sequential circuit can be generated as shown on the right side of Figure 2.6. The initial state I is highlighted in the yellow color and the transition states $T^{i \rightarrow i+1}$ are highlighted in blue. Each state shows the set of circuit points that evaluate to logic 1 enclosed in brackets. Furthermore, the combination of PIs/PPIs that drive the circuit to the corresponding state is shown within each state. As we can see, the maximum switching that can be achieved for the pair (α, β) starting from I is 3 since there is no way for the net α after switching from L to H to switch back to L whereas for net β this is possible by switching back and forth from the bottom states of the Kripke structure.

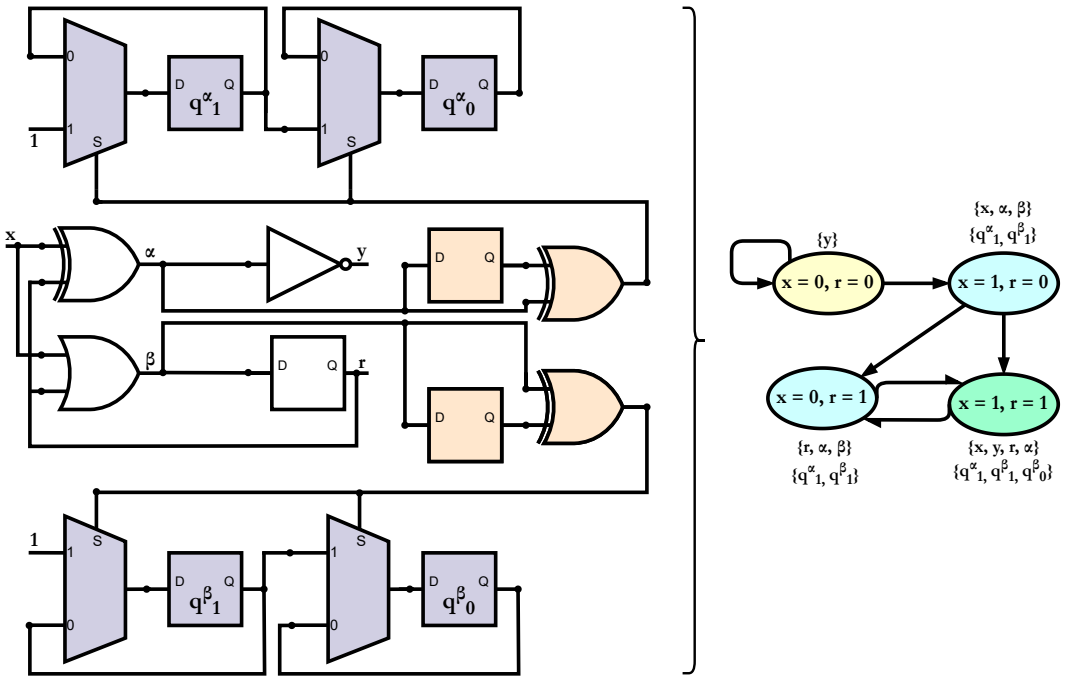

Fig. 2.7 Proposed method concept.

However, what is missing from the example, is the target property P. The method's mechanism that implements the embedding of the multi-point switching property is presented in Figure 2.7. Initially, each net of interest (i.e., α and β) is intercepted and difference detectors are added (colored orange) for each one. Albeit being internal circuit parts which by traditional electronic design automation (EDA) tools are inaccessible, in the domain of propositional logic we know which literals are mapped to each gate, and thus we can easily access and constrain them (see Figure 1.5). The purpose of the difference detector circuits is to detect whether a

toggle has been performed in the current clock cycle by comparing it with the latched value that each net holds. The D Flip-Flop (FF) holds the previous value of the net and in each timeframe this value is compared against the current value the net has. Extra care is taken to assign each FF of the difference detectors the initial value the nets α and β hold rather than initializing them to 0. This is because it is not always given that every net of a design is initialized to 0.

The circuits colored in purple are *unary* counters. These counters use the unary system (base-1) which represents each natural number by a corresponding number of 1-symbols. They count the number of transitions for each net. Since each net, starting from the initial state can perform at most two transitions, two-bit counters are used which are initialized at 0 as indicated by the Kripke structure. The Kripke structure of the enhanced version of the DUT has been updated with a second set of signals that are positively assigned on each state. The signals q_y^x correspond to the net x and the bit index y of the respective counter. In the current state, we can define the BMC target P as $(q_0^a \wedge q_0^b)$. This means for the circuit to reach a state where both unary counters to hold the value 11.

However, as we observed from Figure 2.6, the neighborhood (α,β) cannot achieve a SE of 4. Hence the current BMC problem, after reaching the maximum unrolling depth k=W, would yield `Unreachable`, indicating that there exists no stress sequence that can achieve a SE of 4. However, it is possible to achieve a SE of 3. In Figure 2.7 this corresponds to the state of the Kripke structure which is colored in green.

To account for all possible switching count permutations the following strategy shown in Algorithm 1 is employed:

Initially (lines 1 to 7), we generate all switching permutations for the neighborhood. Considering that each net's switch count can be 0, 1, or 2 and that there are 2 nets per neighborhood, the total number of permutations is $3^2 = 9$. However, out of these permutations, we do not care for the permutation (0,0) i.e., for neither net to switch. After the permutations are sorted in a descending order of their switching sum starting from (2,2) down to (1,0) and (0,1) a call to a BMC solver is issued. If the solver responds with `Reachable` the generation stops and the stress sequence is extracted by decoding the PI literals to 0/1 logic for each TF. The loop continues until either a `Reachable` state is found or no further permutations are left to consider. In the latter case, this means that the current neighborhood is uncontrollable since no

Algorithm 1 Stress sequence generation for arbitrary neighborhood (α, β).

```
1: switching_permutations ← {}
2: for i ← 0,2 do
3:     for j ← 0,2 do
4:         switching_permutations ← (i, j)
5:     end for
6: end for
7: SortSumDescending(switching_permutations)        ▷ (2,2), (2,1), (1,2) ... (0,0)
8: perm_index ← 0
9: S ← NIL
10: repeat
11:     S ← Solve(switching_permutations[perm_index])
12:     perm_index ← perm_index + 1
13: until S = Reachable || perm_index == len(switching_permutations) - 1
```

sensitization pattern exists under the considered functional scenario. Lastly, to incorporate functional constraints in the stimuli generation process a VCM is employed (see Section 1.4.2) which is responsible for prohibiting any kind of non-functional state from being reached during the BMC process. Note that the presented algorithm can be extended to maximize the SWA for arbitrary groups of N nets, e.g., N=3 (triplets) instead of N=2 (pairs), resulting in a complexity of $\mathcal{O}(3^N)$ for each group.

2.3.5 Experimental results

The proposed method has been applied to the `RI5CY` processor, synthesized using the Silvaco 45nm Open Cell Library [64] via Design Compiler by Synopsys. Since we did not have the actual layout of the processor available, we generated an artificial layout-derived mapping of the nets of the functional unit to be stressed instead. Without loss of generality, for every unit, we grouped the internal nets and divided them into unique pairs by using a uniform distribution.

As stress target modules within the `RI5CY` core we used:

- the 32-bit adder, consisting of 538 internal nets that were grouped into $\frac{538}{2}$ = 269 node pairs
- the decoding unit, consisting of 616 internal nets that were grouped into $\frac{616}{2}$ = 308 node pairs.

Table 2.3 Experimental Results

Stimulus Type	Generation Approach	Adder						Decoding Unit					
		Runtime	Stress Efficiency (SE)					Runtime	Stress Efficiency (SE)				
			0	1	2	3	4		0	1	2	3	4
Functional	BMC	8 min	0.74%	0.00%	14.58%	0.00%	84.75%	15 min	0.65%	0.00%	12.34%	4.87%	82.14%
	STL	-	1.12%	0.00%	14.87%	0.00%	84.01%	-	0.32%	0.32%	19.16%	2.60%	77.60%
Non Functional	SCAN	-	0.37%	0.00%	1.86%	1.86%	95.91%	-	0.65%	0.00%	13.27%	3.56%	82.52%

The results of the experiments are shown in Table 2.3. The table shows the percentage of pairs of nodes that achieve a given stress efficiency (from 0 to 4). All sequences generated had a length of W $\leq$ 10. The ideal window size W is dependent on the underlying micro-architecture. In the case of `RI5CY`, we experimentally verified that the optimal stress efficiency achieved by both methods does not increase if the window size is bigger than the aforementioned value. The results of both methods were compared with those produced by an STL achieving 95% of functional stuck-at fault coverage on the whole processor with a duration of approximately 130k clock cycles. In order to accurately compare our results with the STL we performed a coverage profiling logic simulation. Namely, we calculated the number of transitions for all pairs for both units every W=10 clock cycles. The chunk of instructions achieving the maximum value of Equation (2.3) is used for comparing with our results.

As a reference, we performed a further comparison with DfT-based stimuli. After converting the processor into its scan equivalent, we launched an ATPG process, using TestMAX by Synopsys, and considered the generated vectors. By using a test-bench written in SystemVerilog we applied the patterns and computed the maximum stress efficiency they induced to the DUTs in a manner identical to the SBST program (i.e., by using a window of W=10 capture cycles). For the case of the adder, we can see that with scan it is possible to optimally stress pairs of nodes that no functional method managed to toggle. The same can be seen in the case of the decoder. This is expected, since as it has been showcased in [48] DfT approaches enable a better stress distribution since they simultaneously exercise many cells in the circuit. Yet, for both cases, we can see that the stress efficiency of the proposed methods is not that far off from the case of scan-induced stress. On the other side, functional stimuli can be applied at-speed to the DUTs (in contrast to scan), thus can be better for exciting possible weak points in the circuit. Moreover, functional stimuli are guaranteed not to stimulate the circuit differently than in the operational mode (thus avoiding any form of overtesting).

Table 2.4 Supplementary Comparisons

Stress Target	**Runtime**	**Stress Efficiency**				
		0	**1**	**2**	**3**	**4**
Adder	6 h	0.74%	0.00%	14.58%	0.00%	84.75%
Decoding Unit	171 h	0.97%	0.32%	23.70%	2.92%	72.07%

In order to provide an additional comparison, we have also implemented in μgp an evolutionary-based approach to generate stress-inducing sequences maximizing the employed stress metric [15]. The results are presented in Table 2.4. For the evolutionary approach, we constrained all sequences to have a fixed length of W=10 instructions. For the case of the adder, we can see that both methods converged to the same results by achieving to force 84.75% of the unit's pairs to all 4 combinations of values in the target time window. Both methods were found to slightly outperform the stuck-at STL in terms of induced stress. For the case of the decoder, the BMC-based approach outperformed the evolutionary-based one and also the STL program by achieving a notable stress efficiency of 82.14% while the rest achieved 72.07% and 77.60% of optimal switching, respectively. Furthermore, the pair switching is also achieved in a notably shorter period of time than in the case of the STL. For instance, if we concatenate the generated sequences by the evolutionary-based approach for a given unit into a test program the final size would be $10\times$ *total_pairs* in terms of instructions whereas for the BMC-based approach, this would be $\leq 10\times$ *total_pairs*. It is also clear that the latter dominates the former in terms of CPU runtime since for both test generation procedures it converged faster by orders of magnitude, most notably for the case of the decoder.

The justification is the following. Firstly, the way the two methods approach the problem and generate solutions is different. The evolutionary algorithm starts from a completely random (yet valid) set of sequences that are refined in every iteration. Thus, the initial solutions may in fact be far off from the optimal point, and hence longer times are required for the algorithm to converge. On the other hand, the BMC-based algorithm starts by searching for a solution sensitizing both nodes at the same time. As most node pairs are shown to be fully sensitizable, it is rare that multiple BMC problems need to be solved and fast reasoning is achieved. Additionally, the BMC algorithm increases the number of timeframes gradually starting by searching for short, easy-to-compute sequences aiding in reducing the runtime. In theory, through the k-induction [69] and Craig interpolation [70] implemented in the BMC

solver, unsensitizable node pairs can be found before reaching the maximum depth W, which increases the convergence speed of the optimization loop.

Chapter 3

Functionally Untestable Faults Identification

3.1 Background

From the design phase of an integrated circuit (IC) up to the end of the manufacturing phase, but also during the operational phase of the circuits, meticulous testing procedures are applied in order to evaluate and in certain cases (e.g., the safety-critical domain) guarantee that each design works as it was specified and intended. However, in certain end-of-manufacturing tests (e.g., during the structural test) but also during the in-field functional tests, especially in the safety-critical domain, the stringent reliability safety standards mandate extremely high fault coverage (FC) thresholds to be met, in order for an electronic product to be compliant, and thus considered as a part of a safety-critical system.

To provide some perspective, considering the ISO-26262 safety standard regarding the functional safety of electrical and electronic systems in road vehicles, each application of the system is characterized by an automotive safety integrity level (ASIL). As an example, let us consider the entertainment system of a modern automotive vehicle against the subsystem responsible for the airbag control. Obviously, a failure in the entertainment system of the vehicle is less severe than a failure in the airbag control system. To quantify the severity of each ASIL level classes have been devised ranging from quality management (QM), indicating that all assessed risks are tolerable from a safety perspective minor severity, and ASIL A, which is

the lowest rating of the functional safety, up to ASIL D which mandates the most stringent level of safety measures to apply. For each category, the safety standard introduces the probabilistic metric of hardware failures (PMHF), which reports the robustness of the system against generic random hardware faults and is expressed in an absolute number called failures in time (FIT), which is the number of failures expected in one billion hours of operation. Furthermore, the standard introduces the single point of fault metric (SPFM) to quantify the resilience of a given component against single-point faults, while resilience against latent faults is expressed by the latent fault metric (LFM).

Table 3.1 Failure metrics per ASIL according to ISO-26262

ASIL	**PMHF (FIT)**	**SPFM (%)**	**LFM (%)**
A	< 1000	-	-
B	< 100	≥ 90	≥ 60
C	< 100	≥ 97	≥ 80
D	< 10	≥ 99	≥ 90

Table 3.1 contains all of the aforementioned metrics as they are defined in the ISO-26262 standard per ASIL. These metrics, especially the SPFM and LFM, are evaluated by FC percentages. Similar classification and quantification are mandated by other safety standards as well, e.g., DO-254 for aviation, IEC-62279 for railways, and IEC-62061 for industrial machinery.

However, to reach such percentages by means of either structural or functional tests a major requirement is that the vast majority of the DUT structures are controllable and testable. That is, for almost all faults of the design we can generate a test vector that is able to excite and propagate the fault effect to an observable point of the design. However, in reality, this is rarely the case. As new generations of ICs are following an iterative design approach from which parts are either omitted or refined, it is typically the case that certain parts of the circuits are either unused (thus uncontrollable) or given the application profile they become functionally uncontrollable or unobservable, or both. As explained in Section 1.3.2 the presence of such untestable faults has a negative impact on the computation of the FC. Hence, they must be identified and excluded from the final coverage computation.

The research community has put a lot of effort into developing methodologies for identifying untestable faults in the past under various fault models. What follows is an overview of such works, classified according to the employed fault model.

3.1.1 Previous works referring to the stuck-at fault model

The authors of [71, 72] propose an efficient method to identify sequential redundant faults when adopting the stuck-at fault model. Their approach focuses on identifying Flip-Flops (FFs) that cannot be initialized and circuit lines that cannot be controlled to definite values. They classify redundant faults into four types and use this classification to improve the efficiency of test generation systems. This method involves a simple procedure to find uncontrollable lines, aiding in the identification of invalid states and redundant faults, thereby enhancing test generation system efficiency. While their method efficiently identifies sequential redundant faults, it has a disadvantage in terms of computational complexity for large circuits. This is because the process involves an exhaustive search to identify uncontrollable lines and invalid states, which can become computationally intensive as the size of the circuit increases.

In [73] the authors of the FIRE algorithm [74] propose as an extension a novel fault-independent algorithm for identifying untestable faults in sequential circuits under the stuck-at fault model, termed FUNTEST. This method avoids exhaustive searches by utilizing simple implications to identify conflicts in the circuit that indicate untestability, requiring no global reset state or state transition information. This approach significantly outperformed the traditional exhaustive search methods in identifying untestable faults, offering a speed-up of up to three orders of magnitude at that time. However, it is noted that while FUNTEST efficiently identifies a large subset of untestable faults, it does not guarantee the identification of all such faults in the circuit.

In [75] the authors introduce two theorems for identifying untestable faults in sequential circuits using combinational automatic test pattern generation (ATPG). The first theorem focuses on single faults within a combinational array, indicating that if a fault is untestable in this context, it remains untestable in the broader sequential circuit. The second theorem extends this concept to multi-fault scenarios, suggesting that untestable multi-faults in a combinational array correlate to untestable

single faults in sequential circuits. This approach leverages combinational ATPG to simplify and enhance the efficiency of identifying untestable faults in sequential circuits, offering a practical method without the need for an exhaustive search or reset state. However, the effectiveness of their methods might be reduced in highly sequential circuits or in circuits where sequential behavior significantly influences fault testability. This implies that while the method is powerful for a wide range of applications, its utility could be constrained in circuits with complex sequential interactions or minimal combinational components.

In [76] the FILL and FUNI algorithms are presented, which focus on identifying illegal states and sequentially untestable faults in synchronous sequential circuits without assuming a global reset mechanism. FILL uses binary decision diagrams for efficient identification of a large subset of illegal states, while FUNI identifies untestable faults that require some of these illegal states for detection, working faster and identifying many untestable faults without exhaustive search required by automatic test generation procedures. However, FILL cannot detect illegal states forming cycles of length two or more, trading completeness for simplicity and efficiency. In [77], the MUST algorithm is presented as another extension of the FIRES algorithm for identifying sequentially untestable faults in circuits without exhaustive search. MUST extends the FIRES algorithm by performing multiple-stem analysis, identifying additional untestable faults that single-stem analysis misses. It significantly reduces the run-time compared to sequential ATPG by using a more efficient computational approach, although it does not guarantee the identification of all untestable faults. The method's limitation lies in its focus on faults requiring multiple stem assignments, potentially overlooking complex conditions for fault detection.

More recently, the authors of [78, 79] present innovative strategies for identifying untestable faults in sequential circuits. These works introduce fault-independent techniques, including a theorem for fault injection across any timeframe and a method for maximizing local impossibilities to detect multinode conflicting assignments. Together, these approaches advance the capability to identify untestable faults beyond the limitations of previous methods, offering more efficient and comprehensive tools for fault analysis in complex sequential circuits.

The authors of [80] introduce a novel approach for identifying untestable stuck-at faults at the RT-level using model-checking. This method, distinct from logic-level

analysis, efficiently identifies untestable faults by generating property specification language assertions. The technique allows for the formal verification of untestable faults in sequential synchronous designs, demonstrating effectiveness in reducing testing complexity and enhancing high-level test synthesis. However, it acknowledges that while it captures a large subset of untestable faults, it may not identify all such faults, indicating a trade-off between comprehensiveness and computational efficiency.

Lastly, in [17] the authors present methods for identifying on-line functionally untestable faults in embedded processor cores. Their approach focuses on the challenges of functional testing under on-line conditions, identifying faults related to debug/test circuitry and memory configuration constraints. They propose techniques for measuring the impact of these untestability sources on fault coverage, with experimental results showing a potential fault coverage loss of more than 13%. This work highlights the complexity of ensuring comprehensive fault coverage in embedded systems, particularly when functional tests must be executed on-line. Similarly, in [20] the authors present a semi-automated approach for identifying on-line functionally untestable faults in microprocessor cores within safety-critical systems. They highlight the significance of these faults, often overlooked yet crucial for achieving desired fault coverage in in-field tests. Their method, focusing on the fixed application code executed by the processor, demonstrates that a considerable number of faults can be untestable due to the specific operational conditions, contributing to more efficient testing and reliability analysis efforts.

3.1.2 Previous works referring to delay fault models

The research community has extensively focused on the identification of untestable faults, with particular emphasis on the prevalent stuck-at fault model, which effectively characterizes a significant majority of cases. Notably, safety standards such as ISO-26262 have introduced requirements for the consideration of delay faults. Consequently, the exploration of untestable faults has expanded to encompass the comprehensive analysis of delay faults, reflecting the evolving landscape of fault models in safety standards.

In [81] the authors propose an algorithm to identify untestable delay faults in digital circuits quickly. Utilizing pre-computed static logic implications, their fault-

independent approach can identify large sets of untestable faults without enumeration, offering a lower bound on the number of such faults. This method, applicable to both segment and path delay fault models, significantly speeds up the process by avoiding exhaustive searches when compared to typical ATPG tools. However, it's noted that the method provides only a lower bound on untestable faults due to its reliance on an incomplete set of logic implications.

In [82] the authors propose an efficient method to identify untestable path delay faults (PDFs). Their approach reduces the runtime and memory requirements of traditional methods by leveraging equivalence relations between sub-paths within fan-out-free regions of a circuit. This method dynamically prunes the search space to identify pairs of sub-paths that cannot be sensitized together, thus speeding up the identification of untestable paths. However, the authors acknowledge that the requirement to generate sensitization conditions for each sub-path only once can increase memory usage in large circuits and that the computation time may grow quadratically with the number of fan-out-free region sub-paths.

In [83] the authors introduce a novel approach to identifying functionally untestable transition faults in non-scan sequential circuits. They develop a new dominance relationship for transition faults and utilize it to identify a larger number of untestable faults efficiently. The method is structured in two phases: initially identifying a broad set of untestable faults through fault-independent logic implications across multiple time frames, then refining this set by applying dominance relationships to pinpoint additional untestable faults. This technique has been shown to significantly outperform previous methods in experimental evaluations, identifying more untestable faults with less computational effort. While the method efficiently identifies a significant number of untestable faults, it does not guarantee the identification of all such faults. The limitation arises because the approach, although advanced in utilizing logic implications and dominance relationships, may overlook some untestable faults due to the inherent complexity of the circuits or the method's reliance on specific fault models and assumptions.

The authors of [84] introduce a comprehensive methodology for pinpointing untestable transition faults within latch-based systems that incorporate multiple clock domains. This approach significantly enhances the identification process by considering architectural constraints and the varying sizes of defects, thereby facilitating a more accurate and efficient fault analysis in complex designs. However,

a limitation mentioned is the increased computational complexity and memory requirements due to the intricacies of handling multiple clock phases and the diverse defect sizes, which can complicate and lengthen the analysis process.

In [85] the researchers propose a method for identifying testable and untestable PDFs in circuits, utilizing decision diagrams. This approach enhances efficiency by focusing on partial paths or fanout-free segments, enabling the identification of all testable critical PDFs under the bounded delay fault model. The method is shown to be scalable and effective, particularly in handling very path-intensive benchmarks. However, there exist limitations in the processing complexity and the memory requirements for circuits with extensive path counts.

Lastly, in [86] the authors propose two incremental ATPG procedures aimed at enhancing the detection of invalidly tested transition faults caused by untestable defects. Their experimental results, derived from industrial circuits, demonstrate that the proposed methods can maintain or improve the quality of the transition fault test set with a minimal increase in the number of test patterns required.

3.1.3 Previous works referring to other fault models

The subject of the untestable fault identification has also been studied under other fault models such as the bridge faults and the gate-exhaustive fault models.

In [87] the authors propose an efficient implication-based method to identify untestable bridging faults in sequential circuits. Their approach uses sequential symbolic simulation as a pre-processing step to identify uncontrollable nets, followed by an implication-based analysis to determine the testability of each fault. The technique aims to quickly identify untestable bridges, reducing the computational effort required by ATPG tools. They demonstrated the effectiveness of their method through experiments on benchmark and industrial circuits, achieving significant reductions in the size of the bridging fault list for ATPG analysis.

In [88] the authors introduce an ATPG framework for testing both inter-gate and intra-gate bridging faults. They propose Satisfiability (SAT)-solving techniques to efficiently generate test patterns for IDDQ testing, which detects faults based on abnormal quiescent power supply current levels. Their methodology combines random simulations with deterministic stages, leveraging the SAT solver to confirm if a fault is testable or categorically untestable.

Lastly, in [89] the author focuses on the efficient identification of undetectable two-cycle gate-exhaustive faults. The method centers on using input necessary assignments to explore the vast number of necessary assignments characteristic of two-cycle gate-exhaustive faults. By efficiently identifying undetectable faults, the approach helps in reducing the computational effort required for test generation. It is highlighted that gates with a large proportion of undetectable faults are crucial targets for test generation. The method is shown to be effective through experimental results on benchmark circuits. However, while the procedure correctly identifies undetectable faults, it is not complete and may not identify all undetectable faults.

3.2 Uncontrollable lines identification

Efforts in developing methodologies for identifying untestable faults across various fault models have been extensive. Primarily, the focus has been on detecting structurally untestable faults. However, it has become evident, as illustrated in Figure 1.3, that structurally untestable faults constitute only a subset of a larger category: functionally untestable faults. These faults, particularly critical within the context of achieving a sufficient FC percentage given a mission profile, deserve attention. Furthermore, as it can also be inferred from the literature, there exists a lack of a systematic, "global" method that can guarantee the identification of **all** untestable faults of a circuit. This comes naturally, as the more complicated and highly configurable the IC designs become, the harder it becomes to systematically identify such faults.

In this section, we present two SAT-solving-based methods tailored for identifying sets of functionally uncontrollable lines, utilizing scalar, pipelined processors as a case study (without loss of generality). An *uncontrollable* circuit line corresponds to a circuit location for which no input pattern exists that can force the line to both logic values, namely logic 0 and 1 due to structural impediments such as, e.g., a blockage of the driver signal of the line. A *functionally uncontrollable* line, corresponds to a circuit location, that is uncontrollable under the imposed set of functional constraints for the circuit. As an example, let us consider permanent faults related to the most significant bits (MSBs) of a program counter of a 32-bit processor that is employed in a SoC that utilizes 64kB of instruction memory. Given that the memory requires

16 bits for indexing ($2^{16} = 65,536 \rightarrow$ 64kB), it follows that the upper 16 bits of the program counter correspond to functionally uncontrollable lines.

If a circuit line is proven to be uncontrollable, then it is safe to assume that the respective fault(s) (e.g., stuck-at) corresponding to that particular fault site are untestable. However, if a line is proven to be controllable, no verdict can be drawn as it may be that the fault (corresponding to that controllable line) albeit controllable, may be unobservable. The proposed methods target the case of controllability.

3.2.1 Basic idea

One basic solution to the uncontrollable lines identification problem corresponds to simply adding switching constraints for every line (in an iterative way and one at a time) of the circuit to the Boolean conjunctive normal form (CNF) formula by encoding them as clauses and then trying to satisfy it e.g., via an SAT-solver. For every line of the circuit, one CNF formula would have to be constructed and solved. This solution, although theoretically feasible, becomes computationally very critical for non-trivial circuits like CPUs, where each sequence corresponds to a piece of code with the related input data, causing the search space for the SAT solver to become extremely large. The SAT solver would have to find a sequence (if any), which satisfies all the clauses for the given toggling constraint. In this section, we propose to actually divide the problem into smaller ones by considering one instruction at a time. This corresponds to constraining the bits in the CPU instruction register to the values of each instruction's opcode. In this way, we simplify the problem by assigning a specific value to some of the formula clauses. Thus, we are trimming the search space of the solver. It is up to the SAT solver to identify possible values for the CPU inputs in any of the following clock cycles, able to satisfy the specified clauses. The general idea is first to identify the uncontrollable lines when each instruction is considered separately (which is a simpler task) and then to combine the results by identifying lines that are uncontrollable no matter the considered instruction. The final set of uncontrollable lines U includes the lines we can identify in this way. In detail, assuming n instructions are included in the instruction set architecture (ISA) of the target processor, we can state that:

$$U \supseteq \bigcap_{i=1}^{n} U_i = U_1 \cap U_2 \cap \cdots \cap U_n \tag{3.1}$$

where

- U is the total set of the processor's uncontrollable lines
- U_i is the set of uncontrollable lines considering instruction$_i$, only.

The reader should note that when computing U_i the SAT solver abstracts from the initial state of the CPU (i.e., on the previously executed instructions) when considering the generic instruction.

A well-known problem when dealing with test problems in a sequential circuit (hence, in a CPU, too) lies in the identification of the maximum number of timeframes (TFs) to be considered. Given a k-stages pipelined CPU, this number is k in the absence of stalls. However, under operational conditions, a processor's pipeline is prone to stalls originating either from data dependencies or from slow memory accesses. The latter can occur due to a miss in the instruction cache or the data cache of the processor. If we consider a traditional single-issue, scalar pipelined processor comprised of 5 pipeline stages, when a data dependency exists and data forwarding techniques are not sufficient, 2 extra clock cycles can resolve this issue [90]. For example, when there is a data dependency between two consecutive instructions (i.e., a `LOAD` followed by an `ADD` that needs the data loaded in the register from the `LOAD`) the pipeline has to stall for 2 clock cycles so that the results of the former instruction can be forwarded to the latter one. The information about the maximum latency (in clock cycles) for accessing the memory is normally available in the processor's or system's architecture manual/documentation. In other cases, the same information can be extracted by logic simulation or by analyzing the HDL code.

This micro-architectural information is crucial for setting a resource limit to the underlying SAT engine and to model the problem accurately. The resource limit is set by defining an overall amount of timeframes to be used by the solver. When working with a k-staged pipelined processor, we need to consider at least k timeframes. Extra timeframes are added according to the latencies values that were previously mentioned. For example, in a processor with 5 pipeline stages that accesses a memory with 2 clock cycles and needs 2 clock cycles to resolve a data dependency stall, we should model our SAT problem to consider $5+2+2$ timeframes, or $5+2+2+2$ timeframes if the model concerns instructions that access also the data cache (e.g., `LOAD`). In this way, we can model a possible cache

miss scenario and data dependency stalls. This is achieved by adding the appropriate constraints on the CNF formula and thus identifying the lines that cannot be toggled in such scenarios.

In both methods one extra timeframe (TF^0) is added which is used solely for initialization purposes [91]. We leave out constraints on this timeframe in order to consider all possible initial states and not a specific one e.g., an initialization stemming from the activation of the `RESET` signal. This initial timeframe is used to start from a given state, no matter which. In terms of switching activity, we only consider toggling arising in the following timeframes, excluding timeframe 0. No constraints are added in either of the following methods on this timeframe.

Furthermore, in both methods, the `RESET` signal of the processor is constrained to be inactive in all timeframes except timeframe 0 in order to prevent the solver from using it to toggle the circuit's lines, thus violating this typical constraint existing in the operational phase.

3.2.2 Method A

The idea behind Method A is to observe the switching effects that a specific instruction would produce on each pipeline stage **separately** (one at a time). We can analyze the effects of a specific instruction on a given pipeline stage by constraining the respective upstream pipeline registers. Most pipeline registers include some bits to store the opcode of the currently executed instruction. For a typical 5-stage pipeline the pipeline is divided into #pipeline_stages $- 1 = 4$ sections, i.e., the Decode stage (ID), the Execute stage (EX), the Memory stage (MEM), and the Write Back stage (WB). We do not include the Instruction Fetch stage because in that stage the instruction is not yet in the pipeline, but it is being fetched from the memory. Method A is simple and relatively fast since it uses just three TFs (one for initialization, and two others to check for possible toggling conditions). This results in smaller and easy-to-handle CNF formulas for the SAT solver.

The abstract model of Method A is shown in fig. 3.1. For each pipeline stage, one CNF formula is constructed, which is comprised of exactly three timeframes. In each timeframe, the rectangle colored in blue/green represents the respective stage's instruction register. It is assumed that each pipeline stage holds an instruction register that holds the opcode of the instruction that is currently being executed on

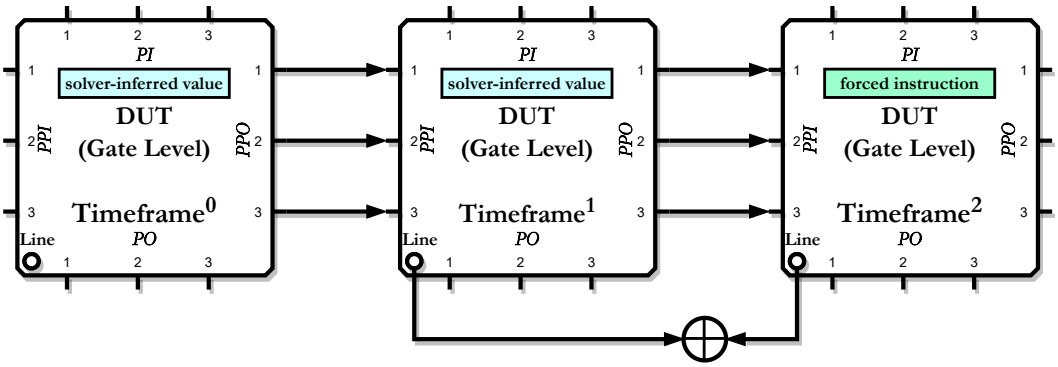

Fig. 3.1 Abstract concept of Method A applied to arbitrary pipeline stage.

the respective stage. However, in the case that a processor's architecture does not mandate for each stage to have an instruction register such as, e.g., the Memory and Write Back stage [92], extra architectural details are required for the test engineer to instead apply the appropriate control words and operands to these stages for every instruction rather than the instruction itself. This ensures the proper functioning of the processor despite the absence of dedicated instruction registers in certain stages.

For each instruction and pipeline stage the method compares the values of the circuit lines at timeframe TF^2, in which the instruction is forced into the respective stage's pipeline register and the timeframe TF^1, in which the instruction is not yet executed by the pipeline stage. Then, in the constructed CNF formulas (for each line separately) the toggling constraints for comparing the value of the selected line between the two respective timeframes are considered e.g., by intercepting the corresponding line's literals for each timeframe and encoding an XOR gate with them, for which we request that its output is 1. Furthermore, we assume that no exceptions are triggered during the computation of the controllability verdict for the circuit's lines.

For Method A, the sets U_i of Equation (3.1) are computed as:

$$U_i = U_{\mathrm{ID}_i} \cap U_{\mathrm{EX}_i} \cap U_{\mathrm{MEM}_i} \cap U_{\mathrm{WB}_i} \tag{3.2}$$

- U_i is the final set of uncontrollable lines for instruction i.
- $U_{\mathrm{ID}_i/\mathrm{EX}_i/\mathrm{MEM}_i/\mathrm{WB}_i}$ is the set of uncontrollable lines found when instruction i is considered for the Decode, Execute, Memory and Write Back stage, respectively.

Algorithm 2 Method A applied to a 5 stage scalar pipelined processor.

```
1:  uncontrollable_per_instruction ← {}
2:  CNF ← GenerateCNF(gate_level)
3:  for i ← 0,n do                                   ▷ ISA has n instructions
4:      opcode ← all_opcodes[I]
5:      not_toggling ← {}
6:      for S in {ID, EXE, MEM, WB} do
7:          ForceLogic(RESET, 0, TF^1)
8:          ForceLogic(RESET, 0, TF^2)          ▷ Assuming RESET active at logic 1
9:          ForceLogic(EXCEPT, 0, TF^1)
10:         ForceLogic(EXCEPT, 0, TF^2)     ▷ Assuming EXCEPT active at logic 1
11:         ForceLogic(instruction_register[S], opcode, TF^2)
12:         for all lines ∈ DUT do
13:             AddTogglingClause(TF^1, TF^2)
14:             result ← Solve(CNF)
15:             if result = UNSAT then
16:                 not_toggling[S] ← line
17:             end if
18:             ClearTogglingClause()
19:         end for
20:     end for
21:     uncontrollable_per_instruction[i] ← Intersect(not_toggling)
22: end for
23: method_A_result ← Intersect(uncontrollable_per_instruction)
```

The method, is presented in pseudo-code format in Algorithm 2. Initially, the circuit is unrolled for a fixed amount of 3 timeframes. Then, for each instruction opcode of the processor's supported ISA and for each of the pipeline stages targeted, the functional constraints (lines 6 to 9) are applied in the form of extra clauses to the CNF in a fashion similar to the one discussed in Section 1.4.2. Of course, given the application profile these constraints can become more elaborate. However, as it is typically up to the customer to select the mission profile, we consider the general case where a minimal set of functional constraints are applied to the DUT.

3.2.3 Method B

In contrast to Method A, Method B considers all pipeline stages together. However, for every instruction, we further divided the problem into sub-problems. The first sub-problem aims to identify uncontrollable lines in the presence of stalls due to misses in the instruction cache (IC)[1] only. The second one looks for uncontrollable lines in the presence of stalls due to misses in the data cache (DC) only. Hence, with respect to Equation (3.1), the sets U_i are now computed as:

$$U_i = U_{\mathrm{IC}_i} \cap U_{\mathrm{DC}_i} \tag{3.3}$$

- U_i is the final set of uncontrollable lines for instruction i.
- U_{IC_i} is the set of uncontrollable lines found by taking into consideration stalls due to data dependencies and misses in the IC for instruction i.
- U_{DC_i} is the set of uncontrollable lines found by taking into consideration stalls due to data dependencies and misses in the DC for instruction i.

A miss in the data cache typically concerns only the `LOAD` and `STORE` (LS) instructions of the processor's ISA. Hence, Equation (3.3) is applicable only for these instructions. For the rest of the ISA's instructions (non-LS instructions), there can only be misses in the instruction cache and so Equation (3.3) is simplified to:

$$U_i = U_{\mathrm{IC}_i} \big|_{\, i \notin \{\texttt{LOAD}, \texttt{STORE}\}} \tag{3.4}$$

While all instructions could be stalled due to a miss in the instruction cache, only the LS instructions are prone to data cache misses. Hence, two instances of Method B should be launched: one that concerns all the instructions and one that concerns only the LS instructions of the processor. The U_i sets are calculated according to Equation (3.4) for all instructions, apart from the `LOAD` and `STORE` instructions, for which we use Equation (3.3).

[1] Not to be confused with the abbreviation used in the rest of the text where IC stands for Integrated Circuit.

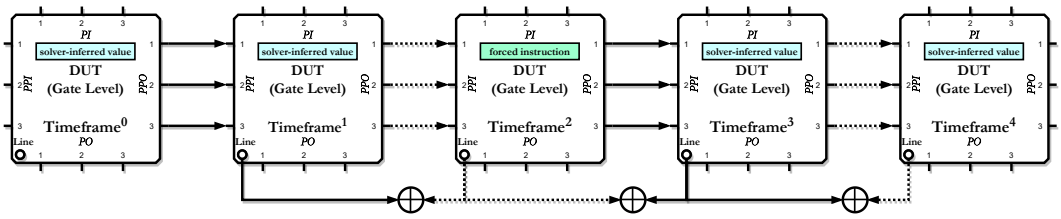

Fig. 3.2 Abstract concept of Method B.

Figure 3.2 depicts the abstract model of the Method. The instructions are now forced in the timeframe, which models the ID stage of the pipeline, which is colored green. Toggling constraints are added for each consequent timeframe for the considered line of the DUT. The total number of timeframes, in this case, depends on the number of stall cycles required from the DUT in the case of an instruction or a data cache miss and also on the number of stall cycles required to resolve a data dependency scenario. This information is derived typically from the documentation of the architecture of the processor. The formula by which the total timeframes are set is shown in the first lines of Algorithm 3 which presents Method B in a pseudo-code format.

As shown both in Figure 3.2 and in line 18 of the algorithm, the timeframe in which each instruction is forced to the pipeline is not a static one. In the case that instruction cache misses are artificially forced (non-LS mode) then the timeframe that models the Decode stage, which is also the timeframe in which the instruction must be forced into the pipeline is the one right after the cache miss is resolved. For instance, if the instruction cache latency lasts for 2 clock cycles, then this timeframe would be the fourth timeframe i.e., timeframe 3. That is one timeframe for the initialization (timeframe 0), two for the cache miss resolution (timeframe 1 and 2), and one for processing the instruction in the Fetch stage (timeframe 3). Whereas, for the LS mode where misses in the data cache are considered, then the timeframe that corresponds to the Decode stage would be the third timeframe i.e., timeframe 2.

Method A targets the lines that are uncontrollable when considering the combinational part of a CPU circuit by itself, assuming full control over its inputs. Clearly, this is a subset of the whole set of uncontrollable lines in the CPU, which can be computed with relatively low CPU time requirements. Method B targets a larger set of uncontrollable lines in the processor, taking into account the evolution of the pipeline in time.

Algorithm 3 Method B applied to a 5 stage scalar pipelined processor.

```
1: CNF ← GenerateCNF(gate_level)
2: if mode = non-LS then                          ▷ Models instruction cache miss
3:     TFs ← #pipeline_stages + insn_cache_latency + data_dep_latency
4: end if
5: if mode = LS then                              ▷ Models data cache miss
6:     TFs ← #pipeline_stages + data_cache_latency + data_dep_latency
7: end if
8: for i ← 0, n do                                ▷ ISA has n instructions
9:     opcode ← all_opcodes[I]
10:    not_toggling ← {}
11:    for all lines ∈ DUT do
12:        for j ← 0, TFs do
13:            ForceLogic(RESET, 0, TF^j)         ▷ Assuming RESET active at logic 1
14:            ForceLogic(EXCEPT, 0, TF^j)        ▷ Assuming EXCEPT active at logic 1
15:            if j > 1 then                      ▷ Avoid comparing with TF^0
16:                AddTogglingClause(TF^j, TF^(j−1))
17:            end if
18:        end for
19:        ForceLogic(instruction_register[ID], opcode, TF^Decode)
20:        if mode = non-LS then
21:            ForceInsnCacheMiss()
22:        end if
23:        if mode = LS then
24:            ForceDataCacheMiss()
25:        end if
26:        result ← Solve(CNF)
27:        if result = UNSAT then
28:            not_toggling[opcode] ← line
29:        end if
30:        ClearTogglingClauses()
31:    end for
32: end for
33: method_B_result ← Intersect(not_toggling)
```

However, Method B suffers from complexity issues that significantly increase the required computation time. Both methods do not start from a given initial state, but from whichever possible state. This leads to a result (i.e., a set of uncontrollable lines) which is an under-approximation of the total set of uncontrollable lines of the circuit, but can be computed with acceptable CPU time requirements.

3.2.4 Experimental results

The two methods presented in the previous sub-sections were applied to the `OR1200` [62] processor, which was synthesized using the Silvaco 45nm Open Cell Library [64] via Design Compiler by Synopsys. Two classes of experiments were performed. Those using Method A are fast in terms of computation time, but yield a relatively small subset of the circuit's uncontrollable lines. On the other side, experiments based on Method B are more time-consuming, but identify a higher number of uncontrollable lines. The 32 instruction opcodes supported by the ISA of `OR1200` were forced one at a time in the pipeline before combining the results.

Method A consists of three separate runs, differing in the stage considered in timeframe 2:

- The Instruction Decode stage.
- The Execute and the Memory (for LS instructions) stages.
- The Write Back stage.

The reason for running 3 different experiments rather than 4 since the `OR1200` implementation was a 5-stage pipeline is that the Execute and the Memory stage of the pipeline are merged into one stage. This happens because the processor implementation used for the experiments treats the Execute stage as a Memory stage for the LS instructions. Furthermore, the caches of the processor were disabled. All the necessary inputs of the Memory stage (i.e., the effective address of the LS instructions) are calculated by combinational modules (i.e., `operandmuxes`) which are located right after the Decode stage.

Three sets of uncontrollable lines are generated for every instruction (one for each stage), which are then processed according to Equation (3.2). In order to get

the final list of uncontrollable lines, we perform the intersections of the sets related to the different instructions, following Equation (3.1).

Method B, consists of 2 runs. Misses in the processor's caches are considered by constraining the instruction and data cache acknowledgment signal. By forcing this signal to 0 for a given number of timeframes, we manage to stall the pipeline, hence creating a scenario for the underlying SAT-solver to investigate uncontrollable lines that occur due to stalls originating from a cache miss. Data dependencies on the other hand are considered by suitably setting the number of timeframes. Specifically, data dependencies are taken into consideration in the experiments by considering a number of timeframes given by: $(1 + \#\text{pipeline\_stages} + 2)$. 2 extra timeframes are considered in our case study because the implementation of the `OR1200` processor we used handles all data dependencies within 2 clock cycles. Lastly, in order to also account for possible cache miss scenarios, which are also handled within 2 clock cycles in our processor, we add 2 extra timeframes to the previous formula (for every cache).

The experiments were performed on two machines using Intel's i9-9990 CPU running at 3.1GHz. Both Method A and Method B experiments are fully parallelizable. Thus we assigned one instance to every core in each machine.

The version of `OR1200` is comprised of $68,899$ lines. This number of lines comes after the removal/pruning of unconnected lines which is automatically performed by the FM framework by performing some topological analysis. In total, out of the $68,899$ lines, 315 were found to be uncontrollable by Method A and 1,592 by Method B. The 315 lines are a subset of the 1,592 lines. Table 3.2 reports the percentage of uncontrollable lines found by the two methods and the pipeline component that they belong to. We can see that in some cases (e.g., `dc_top`, `du`, `wbmux` etc.), Method A detects the same amount of uncontrollable lines as Method B. In other cases the better structured approach of Method B proves to be more effective, detecting a larger set of lines (e.g., `dc_fsm`, `ic_fsm`).

In order to validate the results yielded by the two methods, we performed a logic simulation of a test program, that reached 85% stuck-at fault coverage and we analyzed the resulting toggling circuit activity of the circuit. The logic simulation, performed in QuestaSIM by Siemens EDA, indicated that 8,229 lines of the circuit did not toggle. This set of lines includes the 1,592 lines identified as uncontrollable by Method B. Remarkably, a similar percentage (about 13%) of uncontrollable

Table 3.2 Experimental Results

Unit	#Lines	% of Uncontrollable Lines	
		Method A	Method B
dc_fsm	993	2.11	29.71
except	4,091	0.37	6.72
ic_fsm	734	0.68	36.92
iwb_biu	1,844	0.98	14.37
mult_mac	13,802	0.93	1.11
sprs	2,386	2.05	2.60
immu_top	604	8.44	8.44
operandmuxes	1,489	0.07	1.01
genpc	1,615	0.06	0.80
ctrl	3,045	0.03	0.33
rf	24,656	0.01	0.04
dwb_biu	1,110	0.09	0.72
freeze	60	1.67	13.33
lsu	1,117	0.36	0.63
alu	5,335	0.04	0.13
ic_top	1,112	0.54	0.54
if	1,520	0.33	0.33
wbmux	949	0.11	0.11
du	140	0.71	0.71
dc_top	1,239	0.08	0.08

faults was identified in a previous work [17] using a different version of the same processor, however with some additional operational constraints which may increase the number of uncontrollable lines. These results further demonstrate that the set of uncontrollable lines identified by the method is a correct under-approximation of the circuit's total uncontrollable lines.

To provide a further reference to compare our results with, we run a commercial ATPG on the processor in order to find the uncontrollable lines. We first ran the ATPG on the combinational part of the CPU (i.e., after converting it into its scan version), but the ATPG did not identify any uncontrollable line. Then, for every instruction (opcode) we launched three instances, one for each pipeline stage

(ID/EX/WB), of a combinational stuck-at ATPG, having the respective instruction register bits constrained, similarly to what Method A does. The ATPG identified 88 uncontrollable lines. Lastly, we also launched a sequential ATPG (mimicking the behavior of Method B), but after 48 hours the tool did not detect any uncontrollable lines and thus we stopped it. Results are reported in the 2 bottom lines of Table 3.3.

Table 3.3 Supplementary Comparisons

Method	**CPU time**	**#Uncontrollable Lines**
Method A	12hrs	315
Method B	90hrs	1,592
Combinational ATPG	21min	88
Sequential ATPG		-

The 1,592 uncontrollable lines identified by Method B represent about 2.3% of the total number of lines. By running the commercial ATPG tool on the same processor we identified 88 uncontrollable lines (all of them contained in the `ctrl` unit), corresponding to 0.1% of the total lines. This comparison proves the effectiveness of the proposed method. Lastly, all experiments were performed without enforcing any real constraint on the processor's behavior. By adding functional constraints (e.g., on the used instructions, or on the input data), the number of untestable faults would increase.

3.3 Identification of untestable cell-aware faults

Over the years, device manufacturers have utilized a diverse array of fault models. Apart from the well-established stuck-at model and transition delay model, prominent models include the bridging fault model [93–95], gate-exhaustive fault model [96], embedded-multi-detect fault model [97], and n-detect fault model [98]. Despite their effectiveness in end-of-manufacturing test phases, a concerning trend has emerged in recent years, with customers reporting an increasing number of faulty devices received from manufacturers [99]. This trend is noteworthy as the manufacturers' test flows have historically met the required thresholds for test coverage, as stipulated by the respective standards. The root cause of the failing devices was in some cases proved to be latent defects related to cell-internal faults [100].

These defects are not covered by the state-of-the-art fault models (e.g., stuck-at and transition delay faults) because, in these models, the common assumption is that a fault can only be present on the connections between cells (i.e., the ports of the technology library cells). The cell-aware fault model [99] has been proposed as a solution to systematically target all cell-internal defects of a technology library with great success. Researchers have presented results from high-volume production tests that showed a significant reduction in the defective parts-per-million rate.

Although these aforementioned benefits of cell-aware testing (CAT) regard the end-of-manufacturing test, the in-field test can also benefit from it. Most dominant fault models share the common assumption that a fault is not occurring in the internal part of the cells. Thus, the same latent defects (if not caught early) could still be a serious threat in a safety-critical system. CAT, being able to amend these defects, has the potential to be included as a complementary fault model in the upcoming safety standards revisions. Furthermore, in the area of advanced semiconductor technology manufacturing, where silicon aging is a prime concern, the cell-aware test seems to be a valuable solution.

However, the issue of the identification of the functionally untestable faults (i.e., those faults that will never be able to produce any failure in the considered operational scenario) under the cell-aware fault model remains an open topic. This issue is expected to become increasingly critical due to the growing importance of in-field test of systems including devices manufactured with the latest semiconductor technologies, as explained in the following section.

In this section, we face this issue for the first time to the best of our knowledge and present a method based on bounded model checking (BMC) to systematically identify the untestable static cell-aware faults in combinational cells in a microprocessor given a set of functional constraints. To demonstrate its effectiveness, our method was applied to a RISC-V processor with a minimum functional constraint set, i.e., the minimum constraints required to enable a functional behavior of the system. That is, no assumptions about the software executed on the core were made.

3.3.1 CAT and User-Defined Fault Models

The quality requirements imposed on the industry in recent years are becoming increasingly tougher. This means that the manufacturers have to refine their test

flow in order to improve the defect coverage of their products. However, as mentioned previously, an increasing number of failing devices has been reported by the semiconductor companies' customers to their suppliers although sufficient coverage percentages have been achieved under the state-of-the-art fault models. These numerous test escapes led to cell-aware and other fault models being developed. Instead of focusing only on defects outside the cell and only its interconnections with neighboring cells, the internal structure is also included in the fault model generation.

Typically, each cell's behavior is simulated via an electrical simulator (e.g., SPICE) under different defects and operating conditions. A cell-aware fault model is derived from the resulting defect injection campaign, called cell test model (CTM) [101], or user-defined fault model (UDFM) [102]. During ATPG, the cell-aware fault model is applied to the cells in the circuit which approximates the faulty behavior of the defective cell. Experimental results have showcased the effectiveness of CAT in industrial scale circuits in terms of reduced dppm figures [99]. Ultimately, test escapes related to internal-cell defects can be effectively targeted by CAT or even custom fault models [103]. Although it is true that the generation of the cell test models can be time-consuming due to SPICE simulations being computationally expensive, this process needs to happen only once for each new technology library.

3.3.2 Proposed method

The core concept of BMC is to extract the Boolean formula of the CUT and convert it into a conjunctive normal form. Then, custom requirements are encoded on top in the form of properties. For example, the textual definition of the property for identifying if a fault X is uncontrollable would be: "Can the fault site X be assigned to both logic values?". After the BMC instance is formulated, the BMC solver is started with the task of identifying a model (solution) for the BMC instance. The solver unrolls and solves the BMC problem incrementally by typically translating it to multiple SAT instances that are solved by an SAT-solver, up until a predefined maximum unrolling depth k is reached or a timeout is reached.

For test generation the cell-aware fault model specifies a set of faults per cell of the technology library in the form of defect matrices. Each fault can have one or multiple test alternatives that excite the fault. Table 3.4 contains a simplified defect matrix representing a defect in an AND gate with two inputs `A` and `B`, and one output

0. This fault model only contains the input combinations for which the cell's outputs differ from the fault-free outputs. In this example, the cell has two different entries, called *test alternatives* (TAs) in the following, with input combinations, called *fault conditions*, for which the fault effect of the respective combination is made visible to the gate's output port.

	Fault Conditions		Fault Effects
Test Alternative No.	A	B	O
1	1	0	1
2	1	1	0

Table 3.4 Example defect matrix for AND cell

For each fault in the cell-aware fault list an ATPG process is started. This process builds a BMC instance for the defect matrix and solves the BMC instance generating a test pattern or reports it to be unreachable. In the case that a test pattern could be generated, the fault is marked testable and a fault dropping simulation is performed to check for other detectable faults. In case no test pattern can be generated, the fault is marked untestable.

$$\begin{aligned} A \wedge \neg B &\rightarrow \quad O \\ A \wedge \quad B &\rightarrow \neg O \\ \neg(A \wedge \neg B) \wedge \neg(A \wedge B) &\rightarrow \underbrace{(A \wedge B \leftrightarrow O)}_{\text{fault-free behavior}} \end{aligned}$$

Fig. 3.3 Encoding of faulty AND cell for example model

In the case of the example from Table 3.4, a BMC instance encoding both test alternatives is generated. The faulty cell is encoded according to the formula shown in Figure 3.3, where the first two lines encode the faulty behavior according to the defect matrix; the third line encodes the behavior of the cell when none of the test alternatives is applied to the cell's inputs and is equal to the fault-free behavior.

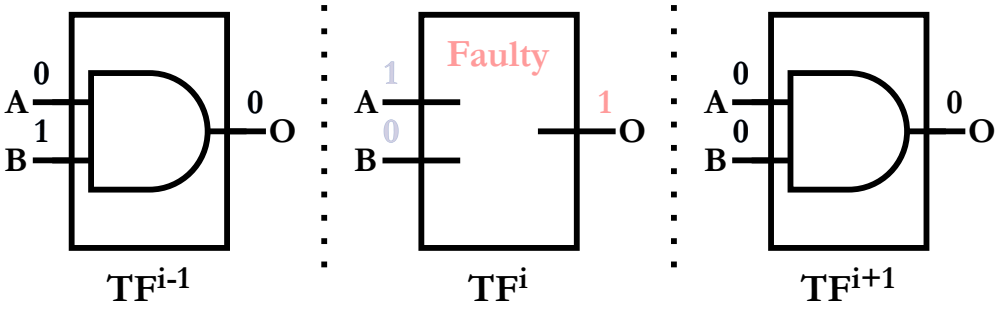

Fig. 3.4 Fault injection on AND cell for example model during BMC

Additionally to the fault propagation, for ATPG efficiency, the BMC instance is extended to enforce a fault activation in at least one timeframe. This is shown in Figure 3.4, where in TF^i the cell-aware fault is sensitized, while the first and last timeframes show no fault behavior. The propagation of the fault effect is then enforced to at least one primary output, which is encoded as the target property for the BMC problem.

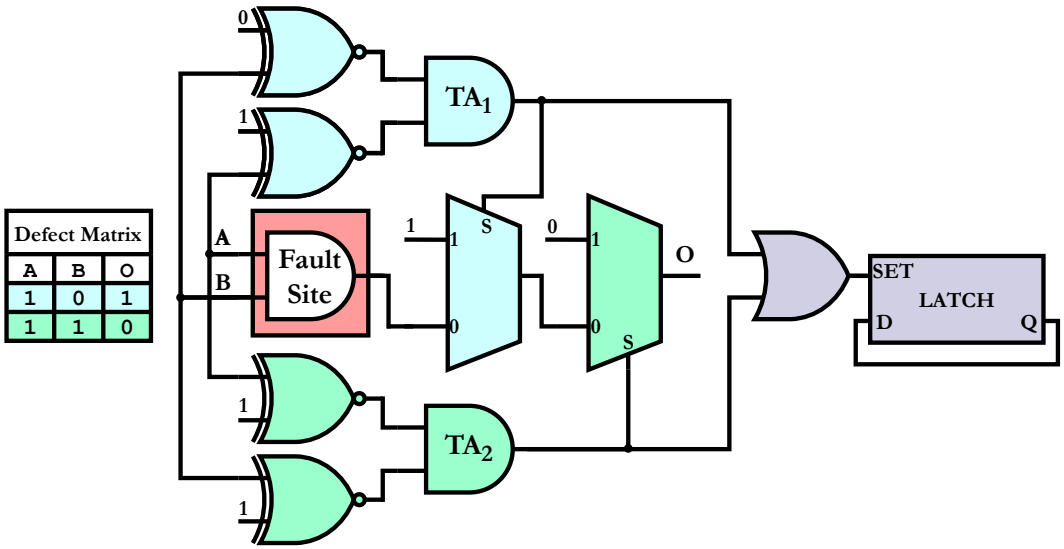

Fig. 3.5 Implementation of proposed cell-aware fault injection for example AND cell

Figure 3.5 depicts the injection of a cell-aware fault. On the left side, the defect matrix (similar to Table 3.4) is composed of two TAs. The inputs `A` and `B` of the faulty AND cell are tapped and a set of logic detectors (that are almost equivalent to the XNOR gates which are depicted instead for convenience) are encoded for each TA and for every input of the cell. The total number of XNOR gates required

for a faulty cell with n inputs with a corresponding defect matrix with m TAs is $n \times m$. Each color (green/blue) shows the corresponding entry of the defect matrix associated with the set of XNOR gates. The outputs of the XNOR gates are then placed into an AND gate which, implies that during BMC solving the respective fault condition must be triggered and thus, the output of the cell is replaced with the TA's faulty output value.

This is achieved by the chain of multiplexers that are placed downstream from the faulty cell's output. Each multiplexer's 0 input is driven by the cell's output. However, for S=1 each multiplexer instead propagates the fault effect of the respective TA of the defect matrix downstream. The S input of each multiplier is driven by the appropriate output of the TA_1 and TA_2 AND gates. The faulty machine propagates the fault effect in the downstream logic for S=1 on some TA, whereas the golden machine propagates the output O of the fault-free AND cell.

To encode the fault sensitization as a property in BMC, the circuit part colored in purple is added. The D-latch simply preserves its state and thus, we can now add the target property unit clause corresponding to the literal of the Q output of the latch to be positively assigned. This implies, given the D-latch's reset state is 0, that the SET input is 1 in at least one timeframe, and with that, the output of the preceding OR gate must be 1 which in turn implies that either TA_1 or TA_2 must be activated. To encode the fault propagation as a property in BMC, we simply have to ask that the difference that will be visible in the O output of the last multiplexer of Figure 3.5 will be shown as a difference to the XOR gates of the miter circuit (Figure 1.9) i.e., a XOR output driven by the circuits' POs to be set to 1.

To apply functional constraints during BMC, we once again employ the VCM concept (see Section 1.4.2), and the initial state of the processor is enforced by the activation of the `RESET` signal (see Section 1.4.3). The BMC process is repeated for each combinational cell of the DUT with its corresponding defect matrix and a testability verdict is drawn according to the response of the BMC solver. If the response is `Reachable` then this means that the cell-aware fault has been sensitized and propagated, thus it is testable. If the solver responds with `Unreachable` however, this means that the cell-aware fault is functionally untestable. The unreachability verdicts are provided by the solver after performing unbounded model checking via Craig interpolation [70] and k-induction [69]. Furthermore, to detect whether a fault is structurally untestable we perform SAT solving for exactly 1 timeframe.

The functional constraints are applied through the VCM once again however the initial circuit state is unconstrained. To prove that a fault is structurally untestable, we add a sensitization constraint to the SAT solver which corresponds to the output of the OR gate of Figure 3.5 (colored in purple) to be set to 1. If the solver responds with `Unsatisfiable` then this means that no TA can be applied to the fault site in a functional manner considering whichever possible state (due to the initial state being unconstrained and hence freely selected by the solver) and thus, the corresponding fault is structurally untestable.

3.3.3 Experimental results

According to the application running on the CUT, a different set of functional constraints may apply. We consider the general case where the set of minimal constraints is chosen to ensure the functional behavior of the processor. The minimal constraint set includes the following constraints and ensures that the ATPG only allows valid functional behavior.

- Valid processor control (reset, run, etc.)
- Valid pipeline control (stall, IF misses)
- Valid executed instructions (ISA)
- Valid control and status registers (CSRs)
- Disable interrupts
- Disable auxiliary processing unit (APU) interfaces
- Disable debug interface.

The executed instructions of the processor are constrained on the instruction bus itself by encoding the RISC-V ISA opcodes into the VCM by automatically transforming them from the official RISC-V opcode specifications into Verilog code [104]. The interrupts are disabled since the interrupt logic is typically tested resorting to different test solutions.

A total of 1,600 SystemVerilog code lines have been used for specifying the constraints via the VCM. 1,500 of them are automatically generated wire declarations

and connections for observing decoded RISC-V instructions and operands. The VCM is synthesized with Yosys and the resulting gate-level VCM has a size of 405 gates.

In order to validate the proposed approach we evaluated it resorting to the processor RI5CY [63] synthesized for the Nangate 45nm PDK [64]. The BMC depth has been set to 50 timeframes and the timeout to 5 minutes. The experiments have been run on an AMD Threadripper 3970X system (32 cores, 64 threads) with 256 GB of RAM and a dual AMD EPYC 7343 system (2x16 cores, 2x32 threads) with 2 TB of RAM in parallel.

Table 3.5 Untestable stuck-at faults identified by the proposed method.

Functional Unit	Combinational Cells	Sequential Cells	Stuck-At		
			Total Faults	Structurally Untestable	Functionally Untestable
if_stage	2,013	304	10,979	123 (1.12 %)	179 (1.63 %)
id_stage	2,944	590	16,240	226 (1.39 %)	196 (1.21 %)
id_stage/regs	3,561	992	27,079	0 (0.00 %)	0 (0.00 %)
ex_stage	168	6	593	26 (4.38 %)	2 (0.34 %)
ex_stage/alu	4,583	107	24,948	18 (0.07 %)	4 (0.02 %)
ex_stage/mult	3,814	3	24,381	2 (0.01 %)	19 (0.08 %)
ls_unit	712	40	4,511	0 (0.00 %)	15 (0.33 %)
cs_registers	2,088	974	15,730	188 (1.20 %)	3,679 (23.39 %)
pmp_unit	11,983	1	50,591	4 (0.01 %)	22,744 (44.96 %)
top	25	1	118	2 (1.69 %)	8 (6.78 %)
total	31,891	3,018	175,170	589 (0.34 %)	26,846 (15.33 %)
		CPU Time (hours)	128.96 h (2.42s per fault on EPYC system)		

For the evaluation two fault models in UDFM format are used. The first fault model is created to compare with the conventional stuck-at fault model. It is created by a custom tool and the resulting cell-aware fault model is equivalent to a simple stuck-at model that only contains stuck-at faults at all the input and output ports of the cells. Note that this fault list contains equivalent faults. This UDFM allows for a simplified comparison with conventional stuck-at simulation results. The UDFM has 534 faults with a total of 2,410 test alternatives. The second fault model is a cell-aware fault model generated by a commercial tool for the Nangate 45nm PDK which is converted into the UDFM format. It contains fault models for testable open, short,

and transistor open and transistor short defects derived from Nangate 45nm PDK SPICE cell models that have been extended with parasitic elements from the layout information. This UDFM has 1,244 faults with a total of 10,546 test alternatives.

Table 3.6 Untestable cell-aware faults identified by the proposed method.

Functional Unit	Combinational Cells	Sequential Cells	Cell-Aware		
			Total Faults	Structurally Untestable	Functionally Untestable
if_stage	2,013	304	22,387	161 (0.72 %)	218 (0.97 %)
id_stage	2,944	590	31,336	426 (1.36 %)	247 (0.79 %)
id_stage/regs	3,561	992	61,878	26 (0.04 %)	0 (0.00 %)
ex_stage	168	6	904	50 (5.53 %)	1 (0.11 %)
ex_stage/alu	4,583	107	48,782	201 (0.41 %)	7 (0.01 %)
ex_stage/mult	3,814	3	83,360	69 (0.08 %)	69 (0.08 %)
ls_unit	712	40	9,121	14 (0.15 %)	21 (0.23 %)
cs_registers	2,088	974	36,479	363 (1.00 %)	6,378 (17.48 %)
pmp_unit	11,983	1	98,792	144 (0.15 %)	31,790 (32.18 %)
top	25	1	150	2 (1.33 %)	11 (7.33 %)
total	31,891	3,018	393,189	1,456 (0.34 %)	38,742 (9.85 %)
		CPU Time (hours)	285.27h (2.50s per fault on EPYC system)		

Table 3.5 and Table 3.6 show the results of our method regarding the untestability analysis under the stuck-at fault model and the cell-aware model, respectively. The reported percentages for the two fault models are not correlated i.e., one is not a subset of the other. As mentioned earlier, the constraint set we have applied is the minimum functional one. That is, no further constraints derived from assumptions about the executed program(s) or the system configuration are made. For instance, combinatorial logic blocks in the fetch and decode stages that are driven by the reset and the clock gating enabling signals are marked as *safe* (i.e., faults that are unable to produce any failure) due to the fact that they are always expected to be forced to fixed values in order to ensure functional behavior. Furthermore, extra functionally untestable faults arise for logic related to the program counter, which is bound to a specific start and end address as is typical to happen in an embedded system in a mission-critical system with limited or shared resources with other peripheral devices that share the same memory which is divided into distinctive regions. In total, 1.12 % / 1.63 % of structurally / functionally untestable faults have been identified for the fetch stage for the stuck-at fault model and 0.72 % / 0.97 % for the cell-aware respectively.

Since the solver is completely agnostic of the architecture's ISA, we enforce all valid instructions with valid operands and operand ranges. Hence, any kind of interrupt stemming from an invalid instruction during decode is not triggered, and thus, the logic blocks related to handling such cases are functionally untestable faults. Furthermore, certain control and status registers related to the generation and the handling of interrupts also contain functionally untestable faults. This is because, in the in-field test scenario, the test is scheduled during idle periods of the system; hence, the interrupts are being disabled during this time to avoid any unpredictable scenario occurring which could potentially have catastrophic consequences to the system and finally to the user(s). For the decode stage, we have identified 1.39 % / 1.21 % of structurally / functionally untestable faults for the stuck-at and 1.36 % / 0.79 % for the cell-aware.

The highest amount of untestable faults are detected in the *physical memory protection* (PMP) unit and the CSRs. The PMP unit provides machine-mode control and status registers per hardware thread to allow memory access privileges to be specified for each physical memory region. The unit performs checks on both the instruction and data memory of the system and is accessible via dedicated CSRs. The usage of certain registers in our configuration was not considered in the BMC search space, which is something that leads to a large number of faults inside the PMP unit being identified as safe. Furthermore, regarding the CSR unit, the number of safe faults identified as a result of the fact that only user level [105] has been considered in the processor configuration. For the CSRs, we have identified 1.20 % / 23.39% of structurally / functionally untestable faults for the stuck-at and 1.00 % / 17.48 % respectively for the cell-aware. For the PMP unit, the percentages are 0.01 % / 44.96 % and 0.15 % / 32.18 % respectively.

Overall, a total of 9.85% of functionally untestable faults have been identified on the whole processor by our method for the static, combinational cell-aware fault model and 15.33% for the stuck-at model. To verify the results of our method, we have performed a fault simulation of a manually written STL that has been developed for the stuck-at fault model, reaching $\approx 60\%$ of fault coverage and compared it with our findings. Two flows were developed in Z01X by Synopsys. One for the stuck-at and one for the cell-aware model. The results showed that the proven untestable faults in Z01X which correspond to structurally untestable faults were a subset of the faults we have identified with our method. Lastly, all the proven functionally untestable faults remained undetected in Z01X.

Chapter 4

In-Field Test

4.1 Background

Functional testing is a crucial step in the development and quality assurance process of integrated circuits (ICs). It involves assessing the performance of the circuits to ensure they meet the specified functional requirements. For the purpose of this document, it is defined as a test performed acting on the functional inputs and observing the functional outputs only, without resorting to any kind of Design-for-Testability (DfT). It is used for end-of-manufacturing testing mainly because it may cover defects that may have escaped any other kind of structural test. However, in the domain of safety-critical applications, functional test is always performed as an in-field test because of its flexibility and ease of applicability and because it is guaranteed not to target safe faults. Also, functional test can be performed by the system company.

In the realm of in-field testing, unique challenges arise, stemming from various constraints that the testing process must adapt to. Such constraints often originate from the specific mission or application profile for the circuit, where the intended use case imposes certain limitations. Additionally, when the DUT is part of an SoC, such as a microprocessor core, further constraints may arise from the configuration of the larger system, potentially prohibiting certain functionalities that would otherwise be accessible in isolation.

In-field test is sometimes applied through software-based self-test (SBST) for the development of software test libraries (STLs) [25]. STLs are practically a collection

of test programs (TPs) and are typically executed in a repetitive manner between idle times of the system in order to ensure that the system functions correctly and safely. They are favored due to their non-intrusiveness as they preserve the system state and do not alter it, are flexible and capable of providing coverage for the widest possible number of faults.

The dominant models, are the stuck-at and transition delay fault models as they adequately model the vast majority of faults that may happen in an operational scenario. However as explained in Section 1.3.3, the generation of STLs is an arduous task for the test engineers as it typically requires a lot of manual effort, especially to target the hard-to-test faults of a modern design.

In this chapter we will introduce Bounded Model Checking (BMC)-based STL test generation routines to target permanent hardware faults for two cases. In the first, the considered DUT is a pipelined processor, whereas in the second, it is a sub-unit of a GPU.

4.1.1 Previous works on STL generation for processors

In [106], the researchers propose a novel SBST methodology for processor cores, addressing the limitations of traditional hardware-based logic Built-In Self-Test (BIST) techniques, especially for high-speed microprocessors. They highlight that while BIST is effective for embedded memory testing, its application to complex logic like microprocessors faces challenges due to high area and performance overheads, and low fault coverage for random-pattern-resistant circuits. Their methodology utilizes software embedded in the processor memory to generate and apply test patterns, enabling structural tests at the processor's operational speed without additional hardware or significant manual effort. This approach significantly enhances fault coverage and testing efficiency, leveraging the processor's own capabilities for self-testing.

In [107], the authors introduce an on-line SBST framework specifically designed for microprocessor cores. This methodology focuses on enabling periodic and non-concurrent on-line testing of microprocessor cores, such as the Motorola `PowerPC 603`, under stringent timing and coverage requirements. By leveraging deterministic SBST approaches, the framework allows for the execution of complex test programs with minimal hardware and software overhead, ensuring system availability and

quality of service in critical environments. The test routines are designed to be modular and fit within tight time slices, thereby not impacting the microprocessor's operational performance.

In [108], the authors introduce a groundbreaking technique for the automatic generation of instruction sequences that target hard-to-detect structural faults in processor cores. This method leverages a software-based approach for executing instruction sequences directly from the processor's cache, enhancing the efficiency and effectiveness of testing. By automating the propagation of module-level test responses to primary outputs, the technique simplifies the complex process of mapping module-level test vectors to instruction sequences, making it feasible to detect faults that are otherwise difficult to uncover with conventional testing methods. However, the authors acknowledge a limitation in their approach: the dependence on automatic test pattern generation (ATPG) tools for generating initial module-level test sequences, which may not always produce optimal results for all types of faults. This highlights an area for future improvement in ensuring the technique's applicability across a broader range of fault models and testing scenarios.

In [109], the authors present an advanced technique for automatically generating instruction sequences to robustly test delay defects in processors. This approach utilizes an ATPG engine for local delay test generation, a Satisfiability (SAT)-based verification engine for global test mapping, and an intelligent feedback mechanism to enhance efficiency. The methodology is capable of achieving nearly 96% coverage of delay faults by loading the generated instruction sequences into the processor's cache for functional testing. This technique represents a significant advancement in the field of processor testing, particularly for identifying and mitigating delay defects that could impact processor performance.

In [110], the authors present an SBST methodology tailored for system peripherals within Systems-on-Chips (SoCs), with a specific focus on direct memory access controllers. This approach is designed to generate compact and efficient test programs that leverage the functional description and high-level architectural view of the peripherals, thus facilitating both design validation and testing. The methodology encompasses two main phases: module configuration and operation, aiming to ensure comprehensive fault coverage through the application of tailored test programs. Experimental results demonstrate the method's effectiveness in achieving significant stuck-at fault coverage with limited test set size and duration.

In [24] the authors present a comprehensive overview of the SBST technique, discussing its role in the microprocessor test and validation process. They elaborate on how SBST serves as a complementary method alongside traditional functional- and structural-test approaches by utilizing software routines to test the processor's functionality. The paper offers a taxonomy of SBST methodologies based on test program development philosophies and summarizes research on optimizing key SBST aspects. Furthermore, it addresses the emerging market demands for higher computational performance at lower costs, which have made SBST an integral part of the manufacturing test flow for major processor vendors, highlighting the method's nonintrusiveness, ability for at-speed testing, avoidance of overtesting, and applicability in the field.

In [111], the authors propose an effective RT-level SBST methodology for embedded processor cores, focusing on enhancing fault coverage without the need for DfT modifications or changes to the processor architecture. This method, rooted in the instruction set architecture and RTL description of the processor, aims to address the challenge of testing SoCs that incorporate embedded processor cores. Their methodology demonstrates superior test quality by significantly increasing fault coverage and reducing test time compared to existing methods.

In [112], the authors propose an SBST methodology specifically designed for on-line testing of small cache memories, such as L1 caches and translation lookaside buffers, in microprocessors. They focus on overcoming the challenges posed by the hidden nature of these memories, utilizing special-purpose instructions and mechanisms like direct cache access instructions, performance monitoring, and trap handling to achieve comprehensive fault coverage. This approach combines crucial features for on-line testing, such as compact test validation, simplified coding style, low invasiveness, and a small memory footprint. The methodology's effectiveness is demonstrated on the `OpenSPARC T1` processor, showcasing significant improvements in test time and efficiency compared to previous SBST approaches that do not utilize direct cache access instructions.

In [113], the authors introduce an automated tool for SBST program generation for microprocessors, leveraging a novel approach that uses High-Level Decision Diagrams (HLDD) for modeling microprocessors and faults. This tool automates the creation of SBST programs from the instruction set architecture of the processor, employing previously developed code templates and HLDD models to generate efficient

test programs. By demonstrating the tool's functionality on the 8-bit PARWAN and 32-bit SPARCv8 microprocessor Leon 3, they showcase its potential to significantly enhance the fault coverage and testing efficiency for microprocessors, promising a valuable resource for test engineers.

In [114] the authors introduce a fault-independent test generation method for SBST of processor-based devices and SoCs. This approach is designed to address the shortcomings of existing SBST techniques that rely heavily on the stuck-at fault model, which is becoming increasingly inadequate for newer semiconductor technologies. By employing a novel SBST-oriented probabilistic metric, their method aims to achieve high coverage of non-modeled faults without the need for the extensive, complex, and CPU-intensive test-generation and fault-simulation processes typically associated with traditional SBST approaches. The proposed method is almost fully automated, significantly reduces the test-program generation time, and is demonstrated through extensive experiments on the OR1200 processor to effectively improve the coverage of non-modeled faults under strict test-application-time and test-program-size constraints.

In [115] the authors propose a high-level, implementation-independent approach for generating functional STL programs for RISC processors. This novel method is designed to quickly produce manufacturing tests with high stuck-at fault coverage, focusing on the separation of test generation for the control and data parts of processors' high-level functional units. For the control part, a new high-level control fault model is introduced, while for the data part, pseudo-exhaustive test strategies are applied, maintaining independence from implementation details. The methodology includes a novel high-level fault simulation method for evaluating fault coverage and identifies redundant gate-level faults in the control part. Experimental results showcase the effectiveness of this approach in generating tests for various units of a RISC processor.

In [116] the authors present a methodology to automatically improve the transition delay fault (TDF) coverage of STLs targeting delay faults starting from STLs for stuck-at faults. Their method identifies excited but not observed TDFs and enriches the STLs with suitable instructions to detect these faults. Through experimental validation on a RISC-V processor, they demonstrate the ability to significantly enhance TDF coverage with a reasonable increase in computational effort and test code size. The approach systematically tests not-observed TDFs with effects reaching user-

accessible registers, achieving nearly complete detection of targeted faults. However, they highlight the potential need for future strategies to test hidden register faults, aiming to closely match the upper bounds of recoverable fault coverage identified in prior studies.

Lastly, in [117] the authors introduce a binary decision diagram (BDD)-based self-test program generation technique for processor cores, focusing on minimizing the efforts required in test template development. By employing BDDs for the justification and optimization of test programs, this method aims to improve fault detection efficiency and coverage, specifically targeting transition delay faults. The methodology has been validated on a RISC-V processor, achieving a notable increase in fault coverage. However, the process of pattern-to-program conversion is identified as time-consuming, highlighting an area for future optimization.

4.1.2 Previous works on STL generation for GPUs

The SBST strategy has been extensively employed for the generation of STLs on microprocessors. More recently, researchers have begun applying this strategy to scenarios where the DUT is a GPU. This is connected to the adoption of GPUs by the safety-critical application sector where in-field testing is mandatory.

In [118], the authors propose a comprehensive SBST and fault localization methodology for CUDA Fermi GPUs. This innovative approach employs custom test strategies for different components within the GPU architecture, ensuring both permanent fault detection and precise fault localization. The methodology is validated through experiments on Fermi GPUs, showcasing its effectiveness in achieving high fault detection rates while maintaining short execution times, making it suitable for on-line testing applications.

In [119], the researchers propose a novel functional test methodology for the GPU scheduler, focusing on the NVIDIA Fermi architecture. This methodology aims to leverage SBST techniques to assess and ensure the reliability of the GPU scheduler, a critical component in managing computation cores and memory operations. The approach outlined seeks to extend traditional CPU-based testing methods to the unique parallel processing environment of GPGPUs, addressing the challenges of testing such complex and highly parallel systems.

In [120], the authors introduce a multi-kernel approach for testing permanent faults in pipeline registers of GPGPUs. This innovative method is designed to effectively identify and diagnose faults within the highly parallel and complex architecture of GPGPUs, specifically targeting the control path fields in pipeline registers. By leveraging multiple testing kernels, each focusing on different aspects of the GPU's functionality, the approach aims to maximize fault coverage and diagnostic resolution, addressing the unique challenges of GPU testing.

In [121], the authors develop an on-line testing technique targeting the scheduler memory of GPGPUs. They introduce a method to generate functional self-test programs that can identify permanent static faults in the memory of the warp scheduler, a critical component for GPU operation. This technique translates fault primitives into self-test functions and programs, enabling effective detection of faults through a sequence of operations designed to excite and propagate the fault to a detectable output. This method emphasizes the importance of maintaining GPU reliability, especially in applications where safety is paramount.

In [122], the authors propose a novel in-field testing strategy for the Divergence Stack Memory within GPGPUs, leveraging a SBST approach. This method focuses on the detection of permanent faults in the stack memory, which is crucial for managing divergent execution paths in parallel processing environments. By employing a modular and scalable testing framework, the strategy enables effective fault coverage and adaptability to different GPU architectures. The proposed approach is validated through extensive simulations, demonstrating its efficiency in detecting a wide range of faults, thereby enhancing the reliability of GPGPUs in safety-critical applications.

In [123], the authors present a functional testing methodology for Special Function Units (SFUs) in GPUs, utilizing a SBST approach. This method is innovative in its strategy for generating test programs that exploit the GPU's parallelism to enhance test speed and minimize memory requirements. The effectiveness of their approach is demonstrated on an open-source GPU model compatible with NVIDIA GPUs, achieving high fault coverage and demonstrating the technique's potential for efficiently testing SFUs in GPUs, particularly in safety-critical applications.

In [124], the authors introduce a new methodology for generating STLs for in-field GPU testing, utilizing High-Level Languages (HLLs) like CUDA. This approach aims to simplify the development process of STLs by reducing the complexity associated with assembly-level encoding. The methodology is demonstrated to be

effective for regular modules within the GPU, although the authors acknowledge that certain modules may require a combination of HLLs and Low-Level Languages due to compiler optimizations and architectural features that limit observability and controllability.

Lastly, in [125] the authors explore the use of STLs for effective in-field testing of GPU cores. Their work evaluates the fault coverage attainable through STLs for all logic modules within a GPU core, showcasing a method to support FMEA in safety-critical applications. They demonstrate that STLs can achieve up to 92.6% stuck-at fault coverage in GPU cores, highlighting the potential of STLs to ensure reliability and functional safety in GPUs without hardware modifications, especially for applications requiring ASIL B or higher when combined with other safety mechanisms.

4.2 STL generation for RISC-V processors

The introduction of the license-free RISC-V [126] ISA facilitated the creation of a vast amount of new processor cores featuring different micro-architectures, base instruction sets, and extensions posing new challenges for STL creation. Especially the shortened development cycles and the automated high-level synthesis of whole processor families that provide ISA extensions depending on the targeted use case require a new adaptive, automated approach.

BMC has been shown to allow semi-automatic generation of STLs for processors using manually constrained ATPG [127–129]. Extending that, [38] introduced an abstraction of the applied constraints by introducing the Validity Checker Module (VCM) concept, which allows for specifying constraints as a circuit written in an HDL and is used during SBST generation to apply constraints to the processor (see Section 1.4.2). By using a VCM the development of constraints for complex STL scenarios is simplified.

However, specifying SBST constraints for whole processor families requires an even higher abstraction level. In this section, we present a structured approach consisting of an enhanced VCM architecture together with an exemplary constraint set that allows for targeting multiple processor cores in an automated way in terms of functional test stimuli generation. Reusability of constraints is provided by having

a configurable constraint set that is specified in a processor-agnostic way. These constraints are mapped onto the processor at hand by a well-defined interface that relates signals and behavior between the processor and the interface. The architecture allows for generating STLs that show different behavior compared to the example constraint set presented in this section as a case study.

The example constraint set constructs an STL to be run by the firmware during idle times. It consists of only arithmetic instructions and targets hard-to-test faults in the ALU and register file. The STL generation makes minimal assumptions about the firmware and is designed to be completely independent of the firmware's instruction memory and state. It computes a checksum in an architecture register which is later verified by the firmware. If a mismatch is detected the firmware can take the appropriate measures according to its use case.

4.2.1 Proposed method

The STL generation is a computationally expensive and complex task. Therefore, we split the problem of SBST generation into simpler steps that are executed as shown in the listing below:

1. Processor and VCM gate-level description import
2. Fault list generation
3. Reset sequence generation
4. Testability check for each fault
5. Instruction sequence generation using BMC
6. Instruction sequence elimination
7. Instruction sequence concatenation to an STL
8. STL evaluation and statistics export.

During steps 3, 4, 5, and 8 the VCM is used to apply constraints to the processor or to evaluate the processor's behavior. For the STL generation, the processor and VCM are expected to be available as gate-level descriptions. The VCM and processor

sources are previously synthesized from RTL to a gate-level description using the synthesis tool of choice and a target technology library.

Initially, the gate-level sources are parsed (step 1). Then, a fault list is generated for the processor which serves as the DUT (step 2). A reset sequence is then generated in order to drive the processor to a well-defined state, i.e., all Flip-Flops (FFs) are assigned valid 0/1 logic values (step 3). Then, multiple testability checks are performed for each fault of the fault list. The checks are performed to find faults for which no test can be generated (for instance, faults that in order to be detected require a reset of the processor or a non-functional state in general, e.g., faults within the debug unit (step 4). After the testability check, the status of all faults, except untestable faults, is set to a `NOT TESTED` state, and the generation of instruction sequences for the STL starts. An instruction sequence is a short sequence of instructions that is created to activate and propagate the fault effect to an observable point (e.g., a Primary Output (PO) of the processor). A BMC problem that generates an instruction sequence is constructed and solved. Subsequently, when a solution is found a fault simulation is performed that tests all so far `NOT TESTED` faults. If they are detected, they are dropped (step 5). After all instruction sequences have been generated, a reverse fault simulation is performed. This removes duplicates and unnecessary or dominated instruction sequences that have been generated during the parallel instruction sequence generation (step 6). Lastly, the full STL is constructed by concatenating all instruction sequences (step 7) and the fault list is reset to its original state for a final fault simulation to be performed in order to evaluate the test program by computing the final fault coverage (step 8).

The original concept of the VCM as Boolean constraint specification has been significantly extended. Support for detecting `DON'T CARE` values in the miter circuit is added by providing additional `IS DON'T CARE` inputs for the VCM. This is required for processing `DON'T CARE` values in the VCM as it operates on a purely Boolean gate-level and for defining the BMC target state later on.

To support a fast adaptation of the STL generation to new processors a VCM architecture that abstracts from processor implementation details was devised. This architecture is shown in Figure 4.1 where the processor is depicted on the left side. On the right side is the VCM which consists of a mapping layer (orange) that is connected to the processor's essential components. The mapping layer connects the processor's signals to a well-defined interface that interacts with the generic

constraints that are shown on the right in red. The generic constraints encode the valid SBST and processor behavior, including the valid RISC-V instructions. For decoding and validating the executed RISC-V instructions an embedded decoding module is included. This decoder module's source code is automatically generated from the formal specification used by the MINRES DBT-RISE-RISC-V instruction set simulator [130] and is adjusted to the supported instruction set of the processor core at hand. The instruction set extensions A, M, F, and D are directly available through the formal specification. For custom extensions, the decoding module can be extended by specifying opcodes of supported RISC-V instructions in JSON format [131].

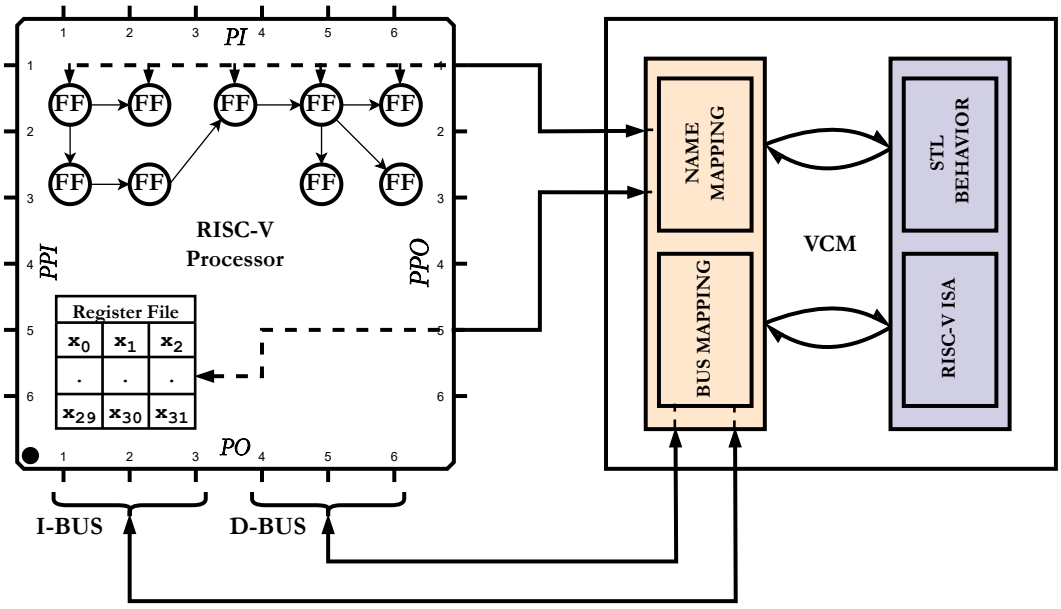

Fig. 4.1 Interaction of processor (left) and VCM (right) with mapping layer in between (middle)

The mapping layer is responsible for translating the processor's signal lines to the generic interface. By applying constraints to the generic side of the interface the constraints are propagated to the processor at hand. For the different steps of the STL generation, the processor signal mappings listed below are implemented:

- Processor control (reset, halt, run)
- Processor state (resetting, halted, running)
- Pipeline state (bubble, flush, halted)

- Program counter
- Architecture register file (`x1` to `x31`)
- Instruction bus transactions
- Data bus transactions.

The instruction and data bus interfaces are both mapped using a bus mapping module written in Verilog. It is exchanged according to the processor bus at hand to reduce adaption efforts. Currently, the bus protocols Advanced High-performance Bus (AHB), Open Bus Interface (OBI), `PicoRV32` [132], and `DarkRISCV` [133] protocols are implemented. Each bus interface is mapped to a set of nine generic signals that monitor transactions. These are a transaction active signal, a transaction commit signal that signals that the transaction was acknowledged, an address signal, a read enable and write enable signal, a read and write byte mask that signals the size of a transaction, and a signal for the read and written value.

For STL generation, the signals listed above are mapped in five variants corresponding to the miter-related inputs available in the VCM. This includes the fault-free and faulty signal, the `DON'T CARE` input for the fault-free and faulty signal, and a difference signal that observes fault propagations for the processor signal. This enables the VCM to implement many different constraints including checking for signals to be well-defined (not `DON'T CARE`), checking for expected signal values and behaviors, and constraining and enforcing fault propagation.

Building on top of the VCM architecture a configurable, generic constraint set is implemented. The constraints are translated through the mapping layer and are applied to the processor. Depending on the SBST generation step different subsets of constraints are enabled to enforce a desired behavior of the processor. We will now give a more detailed overview of the constraint subsets:

Step 3) The first subset of constraints is used to generate a reset sequence for the processor. The `RESET` processor control signal is forced to be active for at least one timeframe. Once the reset signal has been deactivated it is disallowed to be enabled again. Only `ADDI` instructions are allowed by enforcing the opcode on instruction bus read transactions. The data bus is constrained not to allow transactions during the reset sequence. As the BMC target, a defined program counter and a fully defined register file are enforced through constraining the respective `DON'T CARE` signals.

Step 4) The second subset is used for testability checking. This step finds untestable faults that are not relevant in a functional scenario. Table 4.1 shows the four stages that apply increasingly complex constraints to the processor. During each stage, a BMC problem is solved for each fault. If for a fault no solution exists, the fault is marked untestable and excluded from further processing. If a timeout occurs or the maximum unrolling depth is reached, the next step is executed, and no change to the fault status is made.

Table 4.1 Constraints for testability check

Constraint	1	2	3	4
Initial state	-	-	✓	✓
Processor running ∗	-	✓	✓	✓
RISC-V instructions ∗	-	✓	✓	✓
Fault activation and propagation	✓	✓	✓	✓
Directed fault propagation ∗	-	-	-	✓

[1] Combinational full-scan [2] Combinational partial SBST

[3] Sequential partial SBST [4] Sequential SBST

Constraints marked with ∗ are enforced via the VCM while the rest are enforced through the FM framework directly. The first constraint subset is equivalent to a classical full-scan ATPG that only enforces fault activation and propagation but no functional constraints. This step finds untestable faults caused by signal reconvergences. The next step additionally constraints the processor to be running via the processor state signals and allows only valid RISC-V opcodes on read transactions of the instruction bus. This step finds functionally untestable faults, i.e., faults that are only relevant for disallowed operational situations, e.g. when illegal instructions are executed or the processor resets. The third step additionally forces the SBST initial state (reset sequence) to be applied. This step finds faults requiring unreachable states via unbounded model checking features (e.g., Craig interpolation [70]) of the BMC solver. Such faults are also functionally untestable due to unreachable states of the processor like for example unused encodings of pipeline registers. The fourth and last step applies all the previous constraints combined with a directed fault propagation. The fault propagation is enforced by enforcing a difference signal

for either the data bus, the instruction bus, or the register file on the miter circuit (Figure 1.9).

Step 5) The most sophisticated constraint subset is used for the instruction sequence generation since each instruction sequence has to be constructed in a way that it tests its targeted fault but also allows for concatenation to build the final SBST program. The STL example considered is meant to be run in between firmware idle times to check for degradation and makes minimal assumptions regarding the firmware including the instruction memory, data memory, and the register file.

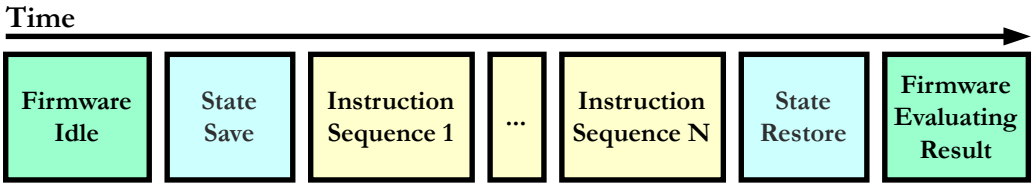

Fig. 4.2 STL execution from left to right: firmware starts STL, firmware context is saved, instruction sequences are run, firmware context is restored, firmware evaluates STL signature.

The final STL program in Figure 4.2 is built from multiple instruction sequences. It is executed by the firmware which first saves the register file to the data memory (e.g., in the stack), and then jumps to the STL program. The STL program computes a checksum in the architecture register `x1` and then jumps back to the firmware which does a state restore excluding the checksum register `x1`. The checksum is verified by the firmware and appropriate action is taken if the verification fails.

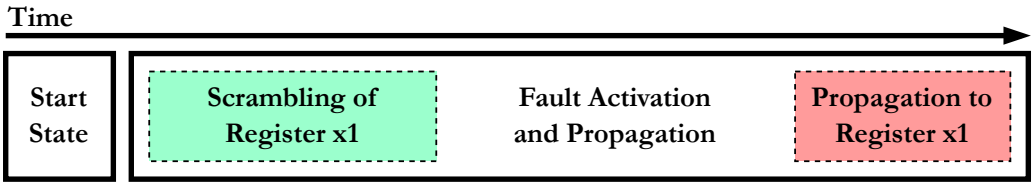

Fig. 4.3 Instruction sequence BMC problem with initial state, scrambling, and final propagation to the register `x1`.

Figure 4.3 shows the structure of an instruction sequence generation. The initial state is set to the reset sequence end state to start with a valid pipeline state. Then, a scramble sequence is generated that permutates register `x1` to reduce the likelihood of fault effect cancellations. Then, arbitrary instructions follow for fault activation

and propagation. Finally, the fault effect is propagated to the x1 register to update the checksum. An example of a scramble sequence is shown in Figure 4.4.

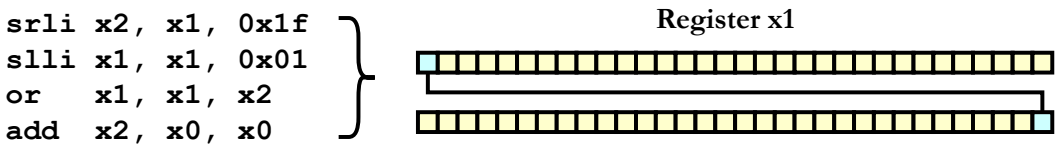

Fig. 4.4 Register x1 scrambling example.

The scramble sequence enforces pre-defined instructions for four time frames rotating register x1 by one bit and then clearing the temporarily used register x2. This results in the fault propagation spreading evenly to all bits of the register and reduces fault effect cancellations. The instruction sequence is enforced by directly constraining the first four instruction fetch transactions on the instruction bus.

The fault propagation to register x1 is enforced via the miter difference signals for register x1 in the BMC target. To be independent of previous checksum values only a limited set of interactions with register x1 are allowed. This requires constraining instruction fetch transactions that interact with register x1. First, only XOR and XORI instructions are allowed to interact with register x1 while the destination and exactly one source register is register x1, ensuring that the register is updated like a checksum.

Further constraints are applied to ensure a functional STL program and the independence of instruction sequences. Specifically, to ensure independence from one instruction sequence to another, we have to abstract from the current register state at the start of each instruction sequence. This is achieved through the VCM. Registers have to be initialized before they are used to further aid starting state independence. This is enforced by keeping a list of initialized architecture registers. Uninitialized registers are forbidden to be used as a source operand. Once a register is written to by an executed instruction, the register is marked as initialized and can be used as a source operand for following instructions. With that, multiple measures are made for instruction sequence independence from the program memory position, the firmware state, and the start state of the processor. Afterward, we proceed with the fault activation and propagation. Hence, we can easily concatenate the sequences without caring about previous processor states.

Furthermore, we enforce via the VCM for the processor to be running without interrupts. All transactions on the data bus are forbidden by constraining data transactions. A read or write on the data bus would either make the STL dependent on or change the firmware state which is disallowed. Instruction fetches are constrained to allow only valid RISC-V opcodes. The program counter is constrained to increase linearly creating a single, linear instruction stream. The STL program is later extracted by monitoring instruction fetches. Further, `JUMP` and `BRANCH` instructions, as well as instructions that are dependent on the value of the program counter, e.g. `AUIPC` are forbidden making the STL's memory location irrelevant.

Step 7-8) After all instruction sequences are generated they are concatenated to a single STL program and evaluated via a fault simulation. During fault simulation, the VCM monitors the processor and its environment and extracts instructions and fault propagations to the register `x1` and the program counter. The results are obtained from the VCM's result outputs and evaluated for fault coverage. A difference in register `x1` at the STL end marks the fault as detected.

4.2.2 Experimental results

The STL generation has been evaluated for two processor families with four configurations in total while considering permanent hardware faults (stuck-at). The non-hierarchical `DarkRISCV` (3-stage pipeline) [133] and a proprietary, hierarchical industrial core (5-stage pipeline) have been chosen to show the effectiveness but also the limitations of the presented constraint set and were synthesized with the Silvaco 45nm [64] technology library. The BMC depth has been set to 15 timeframes and the timeout to 5 minutes. All experiments have been conducted using an AMD Threadripper 3970X system (32 cores, 64 threads) with 256 GB of RAM.

Table 4.2 contains the resulting fault and STL statistics and the respective test generation times. It can be seen that fault coverages of roughly 80 % are achieved for the `DarkRISCV` processor with a program size of 16 kB to 26 kB. Even though the STL program has been constructed to target mainly units that can be tested with arithmetic instructions it can be seen from the test coverage that only roughly 20 % of the processor remains untested.

However, the evaluation of the proprietary core paints a different picture. Here, only the ALU and register file show a fault coverage over 93 % while the testable

Table 4.2 Experimental Results

Processor and ISA		Fault Coverage (percentage)	Testable Fault Coverage (percentage)	Generated Sequences (number)	Program Instructions (number)	Generation Time (hours)	Testable Faults (number)	Untestable Faults (number)	Aborted Faults (number)	Solver Timeouts (number)
DarkRISCV	RV32E	75.85%	79.10%	587	4,112	16.78h	17,301	933	4,576	2
	RV32I	82.42%	84.96%	934	6,526	46.80h	28,056	1,015	4,968	3
	RV32I_Zicsr	79.18%	82.17%	811	5,645	54.18h	28,222	1,299	6,124	7
Proprietary RV32I _Xunknown	IF Stage	6.54%	14.01%	13	76	15.59h	749	1,688	9,018	6,242
	ID Stage	40.29%	45.26%	58	479	4.08h	1,546	421	1,870	1,417
	└Register File	93.40%	99.67%	730	7,196	34.19h	16,536	1,112	56	3
	EX Stage	11.66%	13.65%	67	514	29.82h	2,060	2,575	13,027	10,260
	└ALU	99.70%	99.89%	428	3,087	5.36h	7,848	15	9	3
	MEM Stage	12.88%	18.94%	17	94	2.45h	303	753	1,297	975
	WB Stage	25.63%	26.67%	13	59	0.84h	132	20	363	261
	Miscellaneous	4.27%	5.21%	15	84	4.06h	135	569	2,455	2,029
	Sum	45.40%	51.06%	1,341	11,589	96.39h	29,309	7,153	28,095	21,200

fault coverage surpasses 99.5 %. Other units require an extended constraint set to test. Through the strict constraint set multiple components are not testable, e.g. the exception unit in the instruction fetch (IF) stage, the decoding, bypassing, and hazard detection logic in the instruction decode (ID) stage, the CSRs in the execute (EX) stage and the memory (MEM) stage as no data transactions are allowed.

The STL generation time ranges from 16 h to 100 h. However, 15 % to 50 % (not shown in the table) of the time was spent on fault simulation of the final SBST program in the FM framework and can be further optimized. Additionally, by moving the scramble sequence out of the BMC problem and prepending it manually the runtime of the BMC could be significantly reduced. This shows that the overall generation runtime has the potential to be optimized.

Comparing the program sizes with existing STLs [134] shows that the generated SBST programs require optimizations to reach the compactness of company-provided STLs (46 kB our approach vs 5.8 kB company-provided). However, considering that each instruction sequence currently only uses 4-byte instructions and contains a scrambling sequence that might not be required in most cases the current figures clearly show room for optimizations.

The STL programs have been additionally evaluated and validated with Z01X by Synopsys via the `R4VES` framework [44], which allows for an accurate memory model to be used. This allows simulating advanced fault propagations, e.g., propagation chains from the program counter to the instruction memory to the register file. A custom strobing module evaluates the behavior of the STL program and evaluates the content of register `x1` after the STL program has reached its end. This final validation step was performed in order to prove, with the assistance of a commercial-grade

fault simulator, the effectiveness of the proposed method. The results are listed in Table 4.3.

Table 4.3 Validation of STLs in Z01X

Processor and ISA		Detected	Potentially Detected	Undetected	End Not Reached	Data Bus Used	Exception Occurred	Unknown	Untestable
DarkRISCV	RV32E	69.24%	6.32%	12.74%	3.46%	0.05%	0.00%	3.39%	4.80%
	RV32I	75.70%	6.62%	8.47%	2.26%	0.03%	0.00%	2.16%	4.76%
	RV32I_Zicsr	71.23%	6.53%	11.39%	2.35%	0.03%	0.00%	3.63%	4.82%
Proprietary RV32I _Xunknown	IF Stage	3.43%	0.12%	78.16%	1.46%	0.01%	7.22%	3.40%	6.20%
	ID Stage	40.57%	0.05%	70.67%	0.91%	0.09%	7.44%	7.60%	4.30%
	└Register File	91.23%	0.00%	0.12%	0.00%	0.00%	0.08%	3.33%	5.24%
	EX Stage	11.69%	0.00%	75.24%	0.24%	0.00%	1.13%	6.15%	5.55%
	└ALU	93.75%	0.00%	0.03%	0.00%	0.00%	5.89%	0.20%	0.12%
	MEM Stage	14.42%	0.00%	76.57%	1.05%	0.00%	0.12%	2.20%	5.64%
	WB Stage	23.88%	0.10%	73.69%	0.68%	0.00%	0.05%	1.31%	0.29%
	Miscellaneous	4.25%	0.14%	83.03%	0.37%	0.03%	7.65%	0.70%	3.85%
	Sum	38.51%	0.04%	48.86%	0.51%	0.01%	3.34%	3.90%	4.83%

The faults in Table 4.3 are either classified as *Detected* if the content of the register is fully known at the program end and it differs from the golden value, or *Maybe Detected* if the register is dependent on an unknown state like unspecified instruction memory regions or uninitialized (CSR) registers, or *Undetected* if no difference from the golden signature emerges. Custom fault statuses were used to signal exceptions during the STL execution under fault influence. This includes cases where the end of the SBST is not reached (*End Not Reached*), the firmware state is modified or the fault detection is dependent on the firmware state (*Data Bus Used*), or the processor raises an exception or traps (*Exception Occurred*). Structurally untestable faults are shown in the *Untestable* column. All remaining behaviors that are not classified are put into the *Unknown* category.

The evaluation using Z01X shows that the detected faults for the `DarkRISCV` processors are roughly 5 % below the fault coverage of our approach. However, roughly 6 % of faults have shown to be dependent on uncontrolled factors like the firmware state and could potentially be detected. No exceptions occur since the processor has no exception handling. Two to four percent of faults do not fall into any described behavior (*Unknown*).

The proprietary, industrial core shows that a 3.3 % of the faults create an exception and can be considered as detected. The number of undetected faults for the ALU and register file shows that almost all detectable faults are detected by the built STL

program, validating the assumption of the STL being able to make hard-to-detect faults visible and propagate them to the checksum register. To detect faults in other modules (exception unit, CSRs, IF, ID, and MEM stages) however, the constraint set has to be extended.

4.3 Supporting the STL generation for GPUs

GPUs are particularly important for computationally intensive tasks involving machine learning (ML) applications and computer vision algorithms. In the automotive industry, GPUs have found widespread utilization in various applications such as autonomous guided vehicles [135, 136] and advanced driver assistance systems [137]. In the avionics and space industry, GPUs are utilized as underlying engines to support vision-based navigation and mid-air object detection [138]. Additionally, GPUs aid in computationally intensive tasks, including flight management and data processing, playing a vital role in the industry [139, 140]. Furthermore, there are plans to employ GPUs in railway systems as a powerhouse to enable trains' safe and dynamic management based on environmental and geometrical parameters [141]. Lastly, GPUs are utilized in various industrial applications, including industrial control robots and predictive equipment maintenance [142]. It is worth noting that GPUs are not the exclusive hardware options for data accumulation and processing tasks (e.g., silicon health and operational metrics [143]). In many cases, Tensor Processing Units (TPUs), ASICs, and FPGAs are used alongside GPUs for these purposes.

Given the outlined scenarios where GPUs are utilized in safety-critical contexts, it becomes imperative to implement testing methodologies that align with rigorous safety standards. In the realm of field testing, STLs emerge as a leading choice owing to their adaptability, flexibility, and non-disruptive nature. In practice, an STL is a collection of TPs. These TPs are mainly designed to target individual units and allow the singular identification of faults in a device. Several works [120, 118] have proposed strategies to develop TPs for GPUs, targeting the functional units, the register files, and the internal controllers.

In this section we will present a method based on BMC to aid the generation of TPs to form an STL targeting a specific GPU sub-unit while considering permanent faults. Assuming a pre-existing STL that achieves a certain fault coverage (FC) percentage, we target the remaining, untested hard-to-test faults and try to generate

functional test stimuli for them or, if possible, prove that they are functionally untestable.

4.3.1 GPU organization

Considering that GPUs are more complicated and dense circuits than processors, the problem of generating TPs to form an STL becomes harder. The test engineer must consider a bigger and more convoluted set of functional constraints given that the GPU architecture follows a mix of the Single-Instruction Multiple-Data (SIMD) and Multiple-Instructions Multiple-Data (MIMD) paradigms in order to efficiently process large amounts of data in a parallel fashion [144].

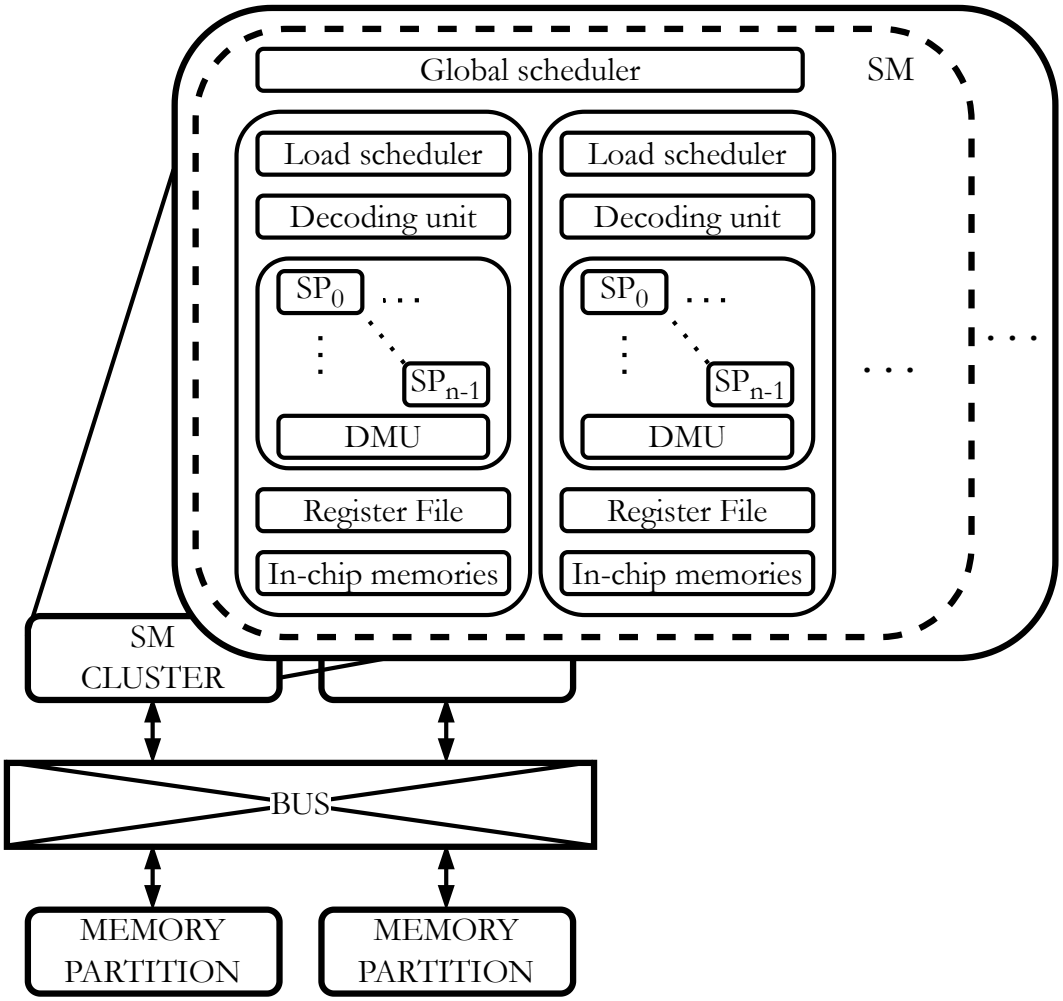

Fig. 4.5 General scheme of GPU's organization.

In general, each SM is organized as a set of pipeline stages, controlled by one or more schedulers and dispatcher units. In each pipeline stage, the SM executes one parallel instruction, internally divided into the procedures of fetching, decoding, execution, reading from memory, and writing to memory. Moreover, the SMs include several execution units (CUDA cores or *Streaming Processors* or SPs) and other accelerators (i.e., Special Function Units, or SFUs). In detail, one parallel program is divided as a set of blocks (Cooperative Thread Arrays, short CTAs) and distributed among the available SMs. The internal controllers submit one instruction from the parallel program for processing, each for a group of threads *(Warps)* [145]. The submitted instruction is initially decoded and then processed in parallel by the available SPs, as one SP per lane. In fact, the same instruction is processed in parallel by several threads using different operands per thread. Then, a new instruction (from the same or another thread group) is submitted and processed.

Modern GPU designs allow the decoding and execution of several instructions in parallel and divide the distribution of the available SPs per SM among the instructions to process, so more than one instruction per thread group can be executed simultaneously [146]. More in detail, the decoding unit plays an important role (as a control unit inside the SMs) in identifying incoming instructions and assigning hardware units and operand sources for the parallel processing among the different parallel threads.

4.3.2 Proposed method

Although BMC is a known technique [38] for STL generation targeting processor circuits, in order to be applied to GPU hardware some parameters have to be considered in greater detail than in the case of CPUs. For example, the complex starting state of the GPU hardware circuit. In the aforementioned papers, a synchronization sequence is typically generated that drives the circuit in a well-defined initial state where there are no `DON'T CARE` values in the flip-flops of the DUT. Whereas, if an initial state is a-priori known and extracted (e.g., via logic simulation of the STL), in our method it can be directly applied on every memory cell saving valuable computation time considering the size of the GPU. Furthermore, certain GPU units depend on architecture-specific memory models (e.g., the DMU interacts with an auxiliary stack). Later in the text, we explain in detail how our method enables abstraction from memory models with the so-called free literals thus making the method suitable

for tackling such cases. The method is graphically presented in Figure 4.6 and will be presented step-by-step. We begin by identifying the target unit inside the GPU as our DUT. Also, we assume a pre-existing STL to be available as a baseline that achieves a certain percentage of coverage on the DUT under the stuck-at fault model. The method focuses on the faults left untested by this original STL. We also assume the DUT will be first synthesized into a gate-level representation using a user-given technology library to target relevant stuck-at faults accurately.

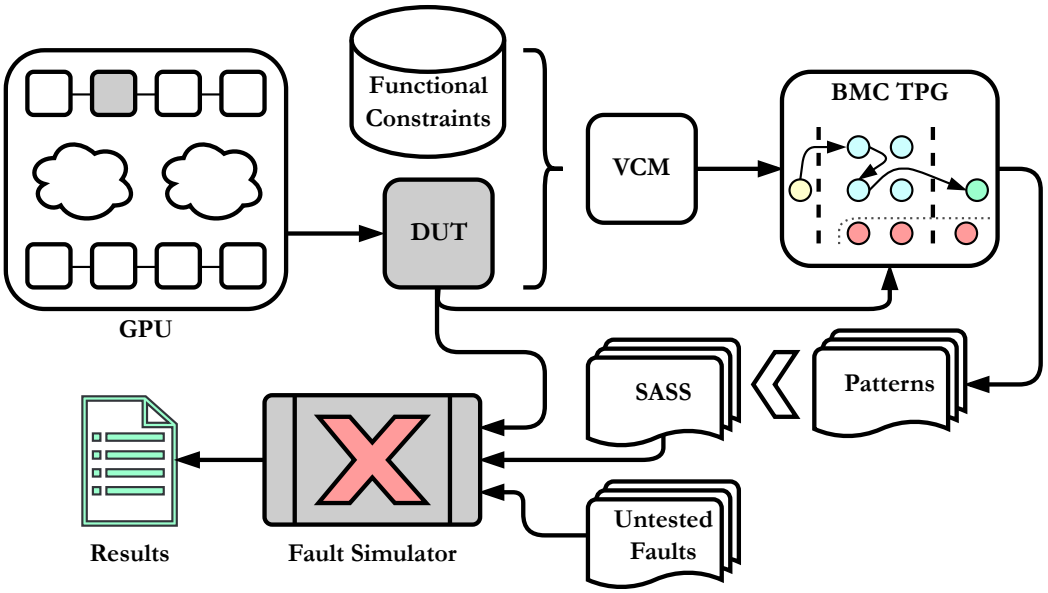

Fig. 4.6 Proposed method using BMC for pattern generation followed by SASS transformation and fault simulation.

A critical step in ensuring the success of the method lies in formulating and applying constraints to the DUT. To replicate the operating conditions of the DUT a functional constraint set has to be devised. This constraint set ensures that the behavior of the DUT can later be mapped to a functional STL. In general, this constraint set includes enforcing valid states and control inputs (e.g., limiting the behavior of component interfaces to adhere to the present bus protocols, limiting memory address ranges, etc.). This crucial task is carried out by the VCM (see Section 1.4.2) adapted to the special needs in the context of GPUs. These constraints are derived from various sources, including (i) the complex architectural characteristics and the valid architecture states of the DUT, (ii) the interactions between the DUT and the entire GPU; for example, certain DUT input signals may have unique values enforced by neighboring functional blocks, (iii) the dependencies of the DUT on

specific configurations; for instance, if a particular input is influenced by the program counter, it must not exceed a certain value. The VCM's logic computes if the state and behavior of the DUT match the constraints that have been encoded inside the VCM. The VCM's validation result is indicated via its outputs. A Boolean value of 1 indicates that the constraint is held, while a 0 indicates that the constraint is not held and the miter circuit shows an invalid behavior.

BMC-based ATPG

The next step in our method is the BMC-based ATPG. We start with a well-defined initial state for the DUT. That is, there are no `DON'T CARE` values in the circuit and all signal assignments abide by the functional constraints. The search or transition space is circumscribed implicitly by the constraints imposed through the VCM. Remember that, during BMC the VCM's validation outputs are constrained via an invariant always to have a value of 1. This enforces the VCM's constraints as they are propagated through the VCM's circuit logic to the VCM's inputs and with that to the miter circuit itself.

For each stuck-at fault of the DUT a property is generated and added as a target state of the BMC problem, responsible for activating and propagating the fault to an observable point. For example, for a generic stuck-at fault *X*, a textual definition of this property is "Can the fault site *X* be set to the opposite logic value and propagate the fault-effect to a primary output of the DUT?". When this iterative process finishes, we obtain solutions (models) for each BMC problem solved for every stuck-at fault.

In the case that the target state is reached for a fault, then this fault is marked as potentially testable, and decoding of the DUT's input signals' literals to 0/1 logic represents a potential test pattern for the fault. Note, that the verdict is potentially testable due to the fact that the propagation points are the primary outputs of the GPU unit and not the GPU itself. Hence, although it may be the case that a fault is identified as testable in the context of the GPU unit, it is not propagatable to a GPU output or *observation point*, i.e., a designated part of the circuit from which we are able to observe a signal's value and compare it with a known fault-free value to determine whether a fault is detected or not.

However, in the case that the target state cannot be reached, then the fault is marked as untestable under the functional constraints imposed during the BMC

process. It is safe to deduce that a fault classified as functionally untestable within the boundaries of the GPU's sub-unit will also be functionally untestable for the whole GPU since if the fault cannot be controlled within the sub-unit then it will also not be controllable when considering the whole circuit as well. Furthermore, considering that the observation points of the unit are the primary and pseudo-primary outputs of the sub-module if the fault is controlled but not observed, then it will also not be propagatable to the rest of the GPU. Lastly, if for a given fault during the BMC process, the solver exceeds a specific limit such as maximum unrolling depth or maximum solve time, then the fault is marked as aborted and no verdict about its testability is generated.

Conversion to SASS assembly & fault simulation

Once the candidate patterns are generated, a conversion and mapping process translates them into valid test assembly instructions, considering the supported ISA for the target device (e.g., SASS ISA in NVIDIA GPUs).

We initially identify the supported ISA of a targeted device and analyze two main features: *i)* instruction's opcode and *ii)* data operand formats, that are directly involved in the mapping of each test pattern as one or a set of instructions. Then, we determine the mapping complexity that depends on the location of the targeted unit in the GPU's structure and the unit's correlation with the assembly instructions. A low mapping complexity indicates that the targeted unit is part of the data path or directly interacts with the execution of assembly instructions. In contrast, high mapping complexity involves that other units interact with the instructions before the operation reaches a targeted unit.

In general, selecting an instruction opcode determines the type of operation performed (e.g., data movement or arithmetic operation). Thus, we analyze the test patterns to identify a feasible and valid opcode. Once a valid opcode is determined, we complete the instruction's missing parts with the information from the original test pattern. In this case, we resort to the identification of data operand formats, e.g., the definition of constant/immediate values or the identification of memory/register addresses that are used by the encoded assembly test instruction. At this point, most candidate test patterns can be mapped into valid instruction's opcode.

As an example let us consider the case of the ALU's adder circuit. The mapping in this scenario is relatively simple since we just have to identify the appropriate instruction operands. Another example, of medium mapping complexity can be considered the decoder unit. From the generated test pattern, we have to identify the instruction opcode and also identify the instruction operands to create the equivalent assembly instruction.

Regarding the mapping complexity, the low complexity case does not introduce additional considerations in the mapping process. Thus, the development of TPs is simplified. In contrast, a high mapping complexity requires the identification of those units involved in the execution of an assembly test instruction. In this case, we determine those possible complementary operations (instructions) to provide initialization conditions to correctly apply an instruction on a targeted unit (e.g., when a test instruction requires parallel data-movement operations from memory, we must initialize the involved registers with valid memory addresses or values for each thread).

Finally, we resort to focused fault simulations to validate the effectiveness of the mapped assembly test instructions on the targeted unit. In this case, we resort to commercial-grade logic simulators to confirm the effective activation and propagation of a targeted fault inside a unit by resorting to the equivalent effect from the mapped test instructions inside the architecture of the device. To reduce the execution time during validation, we focused on the individual execution of each candidate test pattern and its associated targeted fault. When one or a set of assembly test instructions correctly activate and propagate the effects into an observable point (e.g., unit's outputs), we classify the test instructions as effective and include them as part of TPs. When the validation of a test instruction fails, the fault propagation does not reach an observable point and we discard the instruction and classify it as potentially untestable for the system. It must be noted that custom frameworks are adapted according to a targeted unit inside the device.

Case study: The GPU decoding unit

Table 4.4 reports the main identified parallel operational constraints for the decoding unit, which are used to generate the VCM hardware module during the analysis. These constraints are determined considering the primary inputs and outputs of the

Table 4.4 Operational Constraints of the Decoding Unit in a GPU

Operational feature	**Operational constraint**
instruction set	All supported instructions from the SM 1.0 of the G80 architecture
warp processing	Dispatched and executed in increasing order (from 0 to 31)
warp lanes	Dispatched according to scheduler; Increasing sequence (from 0 to 3)
cooperative thread array (CTA) id	Dispatched according to scheduler; Round-robin approach (from 0 to 15)
thread register size	From 0 to 64
thread active state	Active threads in a warp during execution of an instruction; active (1) or inactive (0); at least one active field must be active to execute an instruction
warp instruction program counter	Within the limits of the GPU's system memory
pipeline stall	Status and control line in the SM; when active, the unit stops its execution until this line is released
pipeline done	Completion acknowledge status of a previous operation from all pipeline's stages in the SM; when active, the unit is ready for the next operation

unit and the interaction with the system (i.e., the parallel configuration parameters, which are commonly used for executing instructions). The first group of constraints comprises the supported instructions from the GPU's ISA. This constraint allows the generation of test patterns that are later translated into valid instructions. The second group of constraints depends on the feasible and valid configurations for instructions during the execution (i.e., the size of registers per thread and the number and distribution of warps, CTAs, and SP lanes).

Another group of constraints handles the control features in the unit, including the stall and done conditions in the operation of the unit, and the instruction program counter's limit, which is defined according to the system memory. Finally, the thread's status in a warp (*active/inactive*) is listed as an additional constraint for the unit since the status affects the execution of a possible instruction by the GPU.

4.3.3 Experimental results

In the experiments, we employed the FlexGripPlus GPU model [147, 148] targeting the gate-level description of the decoding unit inside one SM. Moreover, a custom workflow was developed to include the proposed formal method analysis and the evaluation and validation steps for the TPs. The formal analysis, the evaluation, and the verification fault-injection campaigns are performed on a server of 12 Intel Xeon CPUs running at 2.5 GHz and with 256 GB of RAM. The targeted module (the decoding unit) is synthesized using the Silvaco 45nm Open Cell Library [64], consisting of 987 combinational and 359 sequential cells that account for 11,610 permanent stuck-at faults. The proposed method has been implemented within our FM framework (see Section 1.4).

For the fault simulation experiments devoted to verifying possible new test patterns, we employ two commercial tools handling the GPU. First, a logic simulator (QuestaSIM) traces the execution of an existing or new TP, resorting to a mixed-level description of the GPU model (RT- + gate-level). In this case, the targeted unit is the only one simulated at the gate level. Moreover, in this procedure, a trace report is produced from the complete workload (TP) execution, covering the primary inputs and outputs of a targeted unit. Then, the generated trace report serves as input for a functional simulator (Z01X). This tool performs individual fault injection campaigns only on the targeted unit (injecting permanent stuck-at faults) to verify the effectiveness of the instructions in each TP in terms of activation and propagation of fault effects. A fault is identified as detected when at least one of the GPU primary outputs is modified by the effect of the propagation of a fault while executing one of the TPs.

Table 4.5 Main Features of the original STLs for the Decoding Unit

Test Program	**Duration (# of ccs)**	**Size (# of instructions)**	**FC (%)**
`IMM`	2,229,225	32,736	71.13
`MEM`	3,186,236	32,581	76.59
`CNTRL`	710,100	366	71.18
`IMM + MEM + CNTRL`	6,125,561	65,653	80.15

We employ one STL previously developed to functionally test the decoding unit of a GPU [149]. This STL is composed of three main TPs (denoted as `IMM`, `MEM`, and

CNTRL) using immediate operand, memory movement, and control-flow instructions, respectively, to excite permanent faults inside the decoding units and propagate their possible effects. The main features of the TPs are reported in Table 4.5. It is worth noting that all TPs (IMM, MEM, and CNTRL) were designed by employing a pseudo-random approach in combination with the operational constraints of the unit.

Table 4.6 Comparison with Commercial ATPG

	Full-Sequential ATPG	**BMC-based TPG**
# Generated test patterns	86	1,172
# New instructions	86	1,809
% Fully ISA-coherent instructions	100	50.34
% Increase in the STL FC	0	9.57

Then, we employed the set of previously described constraints to analyze the decoding unit and generate new test patterns, which are finally translated into instructions. Table 4.6 reports the results of this process. A commercial sequential ATPG tool was used for comparison purposes. In the analysis, the implemented framework identified 1,172 undetected permanent faults, which the original STLs do not cover. Moreover, the analysis provided a total of 1,809 new test patterns able to excite and propagate faults inside the decoding unit of the GPU. In detail, 535 faults are detected by an individual test pattern, while 637 faults are detected via a sequence of two patterns. The total run-time of the analysis framework was 198 seconds taking advantage of the heavy parallelization of the method to reduce its execution time.

After translating the test patterns into equivalent instructions from the GPU's ISA, we evaluated 1,172 test routines, including only the new instructions (one or two) and the parallel configuration constraints determined during the formal analysis of the unit (i.e., the number of active warps, thread ID, block number, etc.). The experimental results of the evaluation show that out of the newly identified test patterns, about 25% can be directly translated into valid instructions in the GPU's ISA and be executed with minimal restrictions of parallel configuration, so enhancing the test coverage directly without significant effort in the design of the TPs. Thus, these new instructions can be added to the existing TPs. In contrast, a moderate percentage of new test patterns (around 25.34%) require specific parallel configurations after being translated into instructions (i.e., specific memory locations or the addressing

of particular memory resources, such as the shared memory), which means that these instructions cannot be included in previously developed test routines and ad-hoc test programs must be developed.

On the other hand, a considerable percentage of the newly identified patterns (49.66%) can only be translated into valid instructions conditioned to the activation of unfeasible predicate flags (e.g., an always 'false' execution of a global memory load). These instruction types are valid for the ISA but are commonly avoided during the compilation procedures, so they cannot be generated by conventional GPU compilers. Hence, since it is not possible to generate an inline assembly code for GPUs and embed it in any application code, these faults can never force an application to produce a failure. According to safety standards (e.g., ISO 26262) these faults can thus be labeled as safe and removed from the computation of the achieved Fault Coverage. In the end, 16 stuck-at faults in the unit were proven to be uncontrollable and labeled as untestable during the BMC-based ATPG process.

Finally, we compute the overall fault coverage as a combination of the TPs in the original STLs, and the new TP including the newly generated instructions. The overall results provide an FC of 89.72% (an increase of 9.57%, when excluding the identified safe faults), which shows the effectiveness of the proposed technique. These results support the idea that formal methods can be used as a supporting and complementary technique to enhance the development of SBST routines for GPUs devoted to the safety-critical domain.

Although the experiments were performed targeting the decoding unit in a GPU, we claim that the same technique can be adapted for other controllers, functional units, and other modules in the GPU architecture.

Chapter 5

Conclusions

In this PhD thesis we propose solutions based on formal methods for addressing open challenges in burn-in testing, functional untestable fault identification, and in-field testing. While formal methods are widely recognized tools in some domains, their utilization in test and reliability, particularly in comparison to verification, remains relatively limited. This discrepancy can be attributed partly to the non-trivial adaptation required compared to algebraic and structural testing methods, as well as scalability concerns [34].

However, the expressive power offered by formal methods presents a compelling advantage that warrants further exploration across various testing domains. Formal methods have the potential to mitigate certain obstacles, such as formulating functional constraints during test generation. Additionally, the completeness provided by formal methods is a significant asset. A formal method-based approach consistently delivers comprehensive and optimal results based on the set of assumptions and constraints considered by the test engineer. Furthermore, in cases where the problem is unsolvable, formal methods yield counterexamples, offering insights that can aid in refining the overall test flow in which they are applied.

In Chapter 2 we provide methodologies based on formal methods to generate stress-inducting stimuli for the area of burn-in test. We consider a couple of stress metrics, i.e., the repeatable constant switching activity and the 2-multipoint switching activity. We showcase that by functional means it is possible to effectively and efficiently maximize the SWA in the DUT. Experimental results for the first SWA metric, applied to two RISC processors, namely `OR1200` and `RI5CY`, showcase an

improvement ranging from 2% up to 58% for the functional units considered in terms of induced stress. These improvements are computed with respect to other functional test methods such as stressful segments of stuck-at STLs and evolutionary-based approaches for the maximization of the same stress metric. Regarding the second SWA metric, which further incorporates layout information of the DUT, we not only see an improvement when it comes to comparing with other functional methods, but we also observe a deviation from the state-of-the-art approach (based on scan) ranging from 0.38% up to 11.16% (on the optimal stress efficiency). The difference in terms of stress efficiency between the functional and structural (scan) approaches is attributed to the presence of functionally uncontrollable lines. In the functional scenario, the set of nets that the scan method managed to sensitize remains uncontrollable. Hence, any potential fault stemming from those fault sites would be considered safe as it would be impossible to cause any system failure. Overall, by having a well-defined stress metric and a system specification available, we showcase that it is possible to generate functional constraints and optimal functional stress-inducing stimuli, with respect to the stress metric and the constraints by relying on FMs.

In Chapter 3, we emphasize the importance of the identification of functionally untestable faults. We focus on functionally uncontrollable lines in a circuit while considering permanent faults and propose two methods based on SAT-solving to face the issue of uncontrollable line identification. The effectiveness of the methods was showcased on the `OR1200` processor, where a percentage of 2.3% of functionally uncontrollable lines was detected compared to a 0.1% percentage detected by a commercial automatic test pattern generation engine. Furthermore, we consider the relatively new cell-aware test, and we propose a method based on BMC to identify functionally uncontrollable cell-aware faults while targeting combinational cells in the DUT. We underline that as the cell-aware test is a prime candidate for consideration in the new functional safety standards revisions, the issue of the identification of functionally untestable faults is likely to become soon a hot one. The method was applied on the `RI5CY` processor synthesized with the Nangate 45nm technology library for which cell test models were generated for each technology cell. A total of 9.85% of functionally untestable cell-aware faults were detected and 15.33% of functionally untestable stuck-at faults. To our knowledge, this is the first work proposing a solution able to systematically identify functionally untestable cell-aware faults.

Lastly, in Chapter 4, we employ FMs (more specifically, BMC) to generate in a ground-up manner functional test stimuli to compose STLs for the stuck-at fault model. We consider the case where the DUT is a scalar pipelined processor, but also the case where the DUT is a GPU, which is a denser and more sophisticated circuit. The STL generation method for processors was applied to `DarkRISCV` and a proprietary RISC-V core. The STL generation was focused on stuck-at faults on the register file and the ALU of the cores. For the `DarkRISCV` variants, a testable fault coverage of $> 79\%$ was achieved, whereas for the sub-units of the proprietary core percentages, a $> 99\%$ figure was reached (Table 4.2). For the case of the GPU, we isolated the decoding unit of the `FlexGripPlus` GPU and focused on hard-to-test stuck-at faults within this sub-module. By having as a reference an STL targeting the decoding unit and achieving 80.15% of coverage (Table 4.5) we managed to increase it to 89.72% (9.57% increase) by resorting to BMC-based test pattern generation with extensive functional constraints.

Overall, FMs represent a powerful tool, which can be adapted in a "divide and conquer" fashion to a wide variety of test and reliability problems and provide optimal and high-quality solutions. Currently, there are concerns about the scalability of FMs in the general test and reliability area. However, with the continuous growth and evolution of computational hardware, powerful FM engines (solvers) have been developed, able to process large Boolean formulas in combination with high-quality software engineering. Hence, considering the wide applicability that such tools have, while also weighting in the optimality and quality of the yielded solutions, future advancements in hardware and software engineering hold promising potential [150] to address scalability concerns and further enhance the applicability and quality of FM-based solutions in the field of test and reliability.

Future directions

The work presented in this PhD thesis opens the way for new investigations. Specifically, concerning the area of the Burn-In test (presented in Chapter 2) a wider variety of stress metrics can be devised, while also considering real, safety-critical hardware and observing the effects of the internal stress in a manner similar to [48]. By producing heat maps from the application of FM-based stress stimuli, one can compare with other solutions and showcase the quality of the proposed method. When it comes to stress metrics similar to the ones presented in Section 2.3.2, one can also

consider optimal net grouping approaches based on layout-derived information, as done in [151].

Regarding the area of functionally untestable fault identification (presented in Chapter 3), a wider variety of fault models can be considered. For instance, when it comes to cell-aware test, dynamic or 2-pattern faults are not covered by the method proposed in Section 3.3.2 and remain an open issue.

Lastly, when it comes to STL generation (presented in Chapter 4) approaches that better consider the mission profile should be explored, as well as programs that cover further sub-units of a processor core or GPU.

References

[1] Synopsys. What is Moore's Law? https://www.synopsys.com/glossary/what-is-moores-law.html#d. Accessed 19/12/2023.

[2] International Organization for Standardization. Road vehicles – Functional Safety. https://www.iso.org/obp/ui/en/#iso:std:iso:26262:-1:ed-2:v1:en, 2018. Accessed 31/12/2023.

[3] Radio Technical Commission for Aeronautics. Design Assurance Guidance for Airborne Electronic Hardware. https://www.rtca.org/training/do-254-training/, 2005. Accessed 31/12/2023.

[4] R. Vollertsen. Burn-In. In *IEEE International Integrated Reliability Workshop*, 1993.

[5] T. M. Mak. Infant Mortality - The Lesser Known Reliability Issue. In *IEEE International On-Line Testing Symposium (IOLTS)*, 2007.

[6] C. He. Advanced Burn-In - An Optimized Product Stress and Test Flow for Automotive Microcontrollers. In *IEEE International Test Conference (ITC)*, 2019.

[7] Y Han Ng, Y. Hock Low, and S. Demidenko. Improving Efficiency of IC Burn-In Testing. In *IEEE Instrumentation and Measurement Technology Conference (I2MTC)*, 2008.

[8] N. Aghaee, Z. Peng, and P. Eles. Temperature-Gradient-Based Burn-In and Test Scheduling for 3-D Stacked ICs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23:2992–3005, 2015.

[9] C. He and Y. Yu. Wafer Level Stress: Enabling Zero Defect Quality for Automotive Microcontrollers without Package Burn-In. In *IEEE International Test Conference (ITC)*, 2020.

[10] A. Sinha, G. Colon-Bonet, M. Fahy, P. Pant, H. Mao, and A. Shukla. Maximizing Stress Coverage by Novel DFT Techniques and Relaxed Timing Closure. In *IEEE International Test Conference (ITC)*, 2023.

[11] N I. Deligiannis, R. Cantoro, T. Faller, T. Paxian, B. Becker, and M. Sonza Reorda. Effective SAT-based Solutions for Generating Functional Sequences Maximizing the Sustained Switching Activity in a Pipelined Processor. In *IEEE Asian Test Symposium (ATS)*, 2021.

[12] N. I. Deligiannis, T. Faller, R. Cantoro, T. Paxian, B. Becker, and M. Sonza Reorda. Automating the Generation of Programs Maximizing the Repeatable Constant Switching Activity in Microprocessor Units via MaxSAT. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42:4270–4281, 2023.

[13] N. I. Deligiannis, R. Cantoro, and M. Sonza Reorda. Maximizing the Switching Activity of Different Modules Within a Processor Core via Evolutionary Techniques. In *Euromicro Conference on Digital System Design (DSD)*, 2021.

[14] N. I. Deligiannis, R. Cantoro, and M. Sonza Reorda. Automating the Generation of Programs Maximizing the Sustained Switching Activity in Microprocessor units via Evolutionary Techniques. *Microprocessors and Microsystems*, 98:104775, 2023.

[15] N. I. Deligiannis, T. Faller, Z. Chenghan, R. Cantoro, B. Becker, and M. Sonza Reorda. Automating the Generation of Functional Stress Inducing Stimuli for Burn-In Testing. In *IEEE European Test Symposium (ETS)*, pages 1–5, 2023.

[16] V.D. Agrawal and S.T. Chakradhar. Combinational atpg theorems for identifying untestable faults in sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14:1155–1160, 1995.

[17] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan. On-line functionally untestable fault identification in embedded processor cores. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013.

[18] X. Liu and M.S. Hsiao. On identifying functionally untestable transition faults. In *IEEE International High-Level Design Validation and Test Workshop*, 2004.

[19] International Electrotecnical Commission. *IEC 60812:2018 - Failure Modes and Effects Analysis (FMEA and FMECA)*. IEC, 2018.

[20] R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sanchez, and M. Sonza Reorda. About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications. In *IEEE Latin American Test Symposium (LATS)*, 2018.

[21] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J.-G. Mess. An Analysis of Test Solutions for COTS-based Systems in Space Applications. In *IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018.

[22] N. I. Deligiannis, R. Cantoro, M. Sauer, B. Becker, and M. Sonza Reorda. New Techniques for the Automatic Identification of Uncontrollable Lines in a CPU Core. In *IEEE VLSI Test Symposium (VTS)*, 2021.

[23] N. I. Deligiannis, T. Faller, I. Guglielminetti, R. Cantoro, B. Becker, and M. Sonza Reorda. Automatic Identification of Functionally Untestable Cell-Aware Faults in Microprocessors. In *IEEE Asian Test Symposium (ATS)*, 2023.

[24] N. Kranitis, D. Gizopoulos, A. Paschalis, and Y. Zorian. Instruction-based self-testing of processor cores. In *IEEE VLSI Test Symposium (VTS)*, 2002.

[25] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda. Microprocessor Software-Based Self-Testing. *IEEE Design & Test of Computers*, 27:4–19, 2010.

[26] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi. Systematic Software-Based Self-Test for Pipelined Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16:1441–1453, 2008.

[27] Y. Zhang, Y. Ding, Z. Peng, H. Li, M. Fujita, and J. Jiang. BMC-Based Temperature-Aware SBST for Worst-Case Delay Fault Testing Under High Temperature. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30:1677–1690, 2022.

[28] J. E. Rodriguez Condia, F. A. da Silva, A. Ç. Bağbaga, J. D. Guerrero-Balaguera, S. Hamdioui, C. Sauer, and M. Sonza Reorda. Using STLs for Effective In-Field Test of GPUs. *IEEE Design & Test*, 40:109–117, 2023.

[29] P. Parvathala, K. Maneparambil, and W. Lindsay. FRITS - a microprocessor functional BIST method. In *IEEE International Test Conference (ITC)*, 2002.

[30] F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero. Automatic test program generation: a case study. *IEEE Design & Test of Computers*, 21:102–109, 2004.

[31] T. Faller, N. I. Deligiannis, M. Schwörer, M. Sonza Reorda, and B. Becker. Constraint-Based Automatic SBST Generation for RISC-V Processor Families. In *IEEE European Test Symposium (ETS)*, 2023.

[32] N. I. Deligiannis, T. Faller, J. E. Rodriguez Condia, R. Cantoro, B. Becker, and M. Sonza Reorda. Using Formal Methods to Support the Development of STLs for GPUs. In *IEEE Asian Test Symposium (ATS)*, 2022.

[33] J. Anders, P. Andreu, B. Becker, S. Becker, R. Cantoro, N. I. Deligiannis, N. Elhamawy, T. Faller, C. Hernandez, N. Mentens, M. N. Rizi, I. Polian, A. Sajadi, M. Sauer, D. Schwachhofer, M. Sonza Reorda, T. Stefanov, I. Tuzov, S. Wagner, and N. Zidarič. A Survey of Recent Developments in Testability, Safety and Security of RISC-V Processors. In *IEEE European Test Symposium (ETS)*, 2023.

[34] National Aeronautics and Space Administration. What is Formal Methods? https://shemesh.larc.nasa.gov/fm/fm-what.html. Accessed 12/01/2024.

[35] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11:4–15, 1992.

[36] R. Drechsler, S. Eggergluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille. On Acceleration of SAT-based ATPG for Industrial Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27:1329–1333, 2008.

[37] K. Scheibler, D. Erb, and B. Becker. Accurate CEGAR-based ATPG in presence of unknown values for large industrial designs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[38] M. Sauer, B. Becker, and I. Polian. PHAETON: A SAT-Based Framework for Timing-Aware Path Sensitization. *IEEE Transactions on Computers*, 65:1869–1881, 2016.

[39] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, 1983.

[40] S.J.P. Marques and K.A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *IEEE International Conference on Computer Aided Design (ICCAD)*, 1996.

[41] J.P. Marques-Silva and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[42] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[43] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[44] CAD Group, Politecnico di Torino & University of Freiburg. R4VES: RISC-V Environment For Simulations. https://github.com/cad-polito-it/r4ves, 2023. Accessed 01/02/2024.

[45] A. Bierre. *Handbook of Satisfiability*. IOS Press, 2009.

[46] M. Fairuz Zakaria, Z. Abu Kassim, M. P. L. Ooi, and S. Demidenko. Reducing Burn-In Time Through High-Voltage Stress Test and Weibull Statistical Analysis. *IEEE Design & Test of Computers*, 23:88–98, 2006.

[47] W. Ruggeri, P. Bernardi, S. Littardi, M. Sonza Reorda, D. Appello, C. Bertani, G. Pollaccia, V. Tancorre, and R. Ugioli. Innovative methods for Burn-In related Stress Metrics Computation. In *International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2021.

[48] D. Appello, P. Bernardi, G. Giacopelli, A. Motta, A. Pagani, G. Pollaccia, C. Rabbi, M. Restifo, P. Ruberg, E. Sanchez, C.M. Villa, and F. Venini. A Comprehensive Methodology for Stress Procedures Evaluation and Comparison for Burn-In of Automotive SoC. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[49] S. Chowdhury and J. Sabir Barkatullah. Estimation of Maximum Currents in MOS IC Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9:642–654, 1990.

[50] H. Kriplani, F. Najm, and I. Hajj. Maximum Current Estimation in CMOS Circuits. In *ACM/IEEE Design Automation Conference (DAC)*, 1992.

[51] J. Monteiro, S. Devadas, A. Ghosh, K. Keutzer, and J. White. Estimation of Average Switching Activity in Combinational Logic Circuits Using Symbolic Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16:121–127, 1997.

[52] C. Y. Wang and K. Roy. Maximum Power Estimation for CMOS Circuits Using Deterministic and Statistical Approaches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6:134–140, 1998.

[53] S. Devadas, J. White, and K. Keutzer. Estimation of Power Dissipation in CMOS Combinational Circuits Using Boolean Function Manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11:373–383, 1992.

[54] K. C. Huang, C. L. Lee, and J.E. Chen. Maximization of Power Dissipation Under Random Excitation for Burn-In Testing. In *IEEE International Test Conference (ITC)*, 1998.

[55] A. Sagahyroon. Maximizing Heat Dissipation for Burn-In Testing. In *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2002.

[56] F. Najm and M. Zhang. Extreme Delay Sensitivity and the Worst-Case Switching Activity in VLSI Circuits. In *ACM/IEEE Design Automation Conference (DAC)*, 1995.

[57] S. Manich and J. Figueras. Maximizing the Weighted Switching Activity in Combinational CMOS Circuits Under the Variable Delay Model. In *IEEE European Conference on Design and Test (EDTC)*, 1997.

[58] W. Qing, Q. Qinru, and M. Pedram. Estimation of Peak Power Dissipation in VLSI Circuits Using the Limiting Distributions of Extreme Order Statistics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:942–956, 2001.

[59] F. A. Aloul and A. Sagahyroon. Estimation of the Weighted Maximum Switching Activity in Combinational CMOS Circuits. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2006.

[60] F. A. Aloul and A Sagahyroon. Using SAT Techniques in Dynamic Burn-In Vector Generation. In *IEEE Mediterranean Electrotechnical Conference (MELCON)*, 2010.

[61] R. Cantoro, M. Sonza Reorda, A. Rohani, and H. G. Kerkhoff. On the Maximization of the Sustained Switching Activity in a Processor. In *IEEE International On-Line Testing Symposium (IOLTS)*, 2015.

[62] OpenRISC. https://openrisc.io. Accessed 21/01/2024.

[63] PULP. https://pulp-platform.org/. Accessed 21/01/2024.

[64] Silvaco 45nm Open Cell Library. https://si2.org/open-cell-library. Accessed 21/01/2024.

[65] Sanchez E., Schillaci M., and Squillero G. *Evolutionary Optimization: the μGP toolkit*. Springer, Berlin, New York, 2011.

[66] A. Mahzoon, D. Große, and R. Drechsler. PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[67] D. Kaufmann, A. Biere, and M. Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *Formal Methods in Computer Aided Design (FMCAD)*, 2019.

[68] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 2018.

[69] T. Wahl. The k-Induction Principle. https://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf. Accessed 25/01/2024.

[70] K. McMillan. Applications of Craig Interpolation to Model Checking. In *International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*, 2005.

[71] H.-C. Liang, C. L. Lee, and J.E. Chen. A sequential redundant fault identification scheme and its application to test generation. In *IEEE Asian Test Symposium (ATS)*, 1994.

[72] H.-C. Liang, C. L. Lee, and J.E. Chen. Identifying Untestable Faults in Sequential Circuits. *IEEE Design & Test of Computers*, 12:14–23, 1995.

[73] M.A. Iyer and M. Abramovici. Sequentially Untestable Faults Identified Without Search ("Simple Implications Beat Exhaustive Search!"). In *IEEE International Test Conference (ITC)*, 1994.

[74] M.A. Iyer and M. Abramovici. Low-Cost Redundancy Identification for Combinational Circuits. In *International Conference on VLSI Design (ICVD)*, 1994.

[75] V.D. Agrawal and S.T. Chakradhar. Combinational atpg theorems for identifying untestable faults in sequential circuits. In *IEEE European Test Conference (ETC)*, 1993.

[76] D. E. Long, M.A. Iyer, and M. Abramovici. FILL and FUNI: Algorithms to Identify Illegal States and Sequentially Untestable Faults. *ACM Transactions on Design Automation of Electronic Systems*, 5:631–657, 2000.

[77] Q. Peng, M. Abramovici, and J. Savir. MUST: multiple-stem analysis for identifying sequentially untestable faults. In *IEEE International Test Conference (ITC)*, 2000.

[78] M. Syal and M.S. Hsiao. Untestable Fault Identification Using Recurrence Relations and Impossible Value Assignments. In *International Conference on VLSI Design (IVCD)*, 2004.

[79] M. Syal and M.S. Hsiao. New Techniques for Untestable Fault Identification in Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:1117–1131, 2006.

[80] J. Raik, H. Fujiwara, R. Ubar, and A. Krivenko. Untestable Fault Identification in Sequential Circuits Using Model-Checking. In *IEEE Asian Test Symposium (ATS)*, 2008.

[81] K. Heragu, J.H. Patel, and V.D. Agrawal. Fast Identification of Untestable Delay Faults Using Implications. In *International Conference on Computer Aided Design (ICCAD)*, 1997.

[82] Y. Shao, S.M. Reddy, S. Kajihara, and I. Pomeranz. An Efficient Method to Identify Untestable Path Delay Faults. In *IEEE Asian Test Symposium (ATS)*, 2001.

[83] X. Liu and M.S. Hsiao. On Identifying Functionally Untestable Transition Faults. In *IEEE International High-Level Design Validation and Test Workshop (HLDVT)*, 2004.

[84] M. Syal, S. Chakravarty, and M.S. Hsiao. Identifying Untestable Transition Faults in Latch Based Designs with Multiple Clocks. In *International Conference on Test (ITC)*, 2004.

[85] S. Padmanaban and S. Tragoudas. Efficient Identification of (Critical) Testable Path Delay Faults Using Decision Diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24:77–87, 2005.

[86] X. Lin and J. Rajski. The Impacts of Untestable Defects on Transition Fault Testing. In *IEEE VLSI Test Symposium (VTS)*, 2006.

[87] M. Syal, M.S. Hsiao, K.B. Doreswamy, and S. Chakravarty. Efficient implication-based untestable bridge fault identifier. In *IEEE VLSI Test Symposium (VTS)*, 2003.

[88] T. Nakura, Y. Tatemura, G. Fey, M. Ikeda, S. Komatsu, and K. Asada. SAT-based ATPG testing of inter- and intra-gate bridging faults. In *IEEE European Conference on Circuit Theory and Design (ECCTD)*, 2009.

[89] I. Pomeranz. Efficient Identification of Undetectable Two-Cycle Gate-Exhaustive Faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41:776–783, 2022.

[90] Henessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Elvesier, 6 edition, 2017.

[91] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2005.

[92] VCAS Laboratory, University of Ioannina. Tethorax: A RISC-V Base Integer ISA Implementation (RV32I). https://github.com/NikosDelijohn/Tethorax, 2019. Accessed 07/02/2024.

[93] F.J. Ferguson and T. Larrabee. Test Pattern Generation for Realistic Bridge Faults in CMOS ICs. In *IEEE International Test Conference (ITC)*, 1991.

[94] J. Rearick and J.H. Patel. Fast and accurate CMOS bridging fault simulation. In *IEEE International Test Conference - (ITC)*, 1993.

[95] P. Engelke, I. Polian, J. Schloeffel, and B. Becker. Resistive Bridging Fault Simulation of Industrial Circuits. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2008.

[96] K.Y. Cho, S. Mitra, and E.J. McCluskey. Gate Exhaustive Testing. In *IEEE International Conference on Test (ITC)*, 2005.

[97] J. Geuzebroek, E.J. Marinissen, A. Majhi, A. Glowatz, and F. Hapke. Embedded multi-detect ATPG and Its Effect on the Detection of Unmodeled Defects. In *IEEE International Test Conference (ITC)*, 2007.

[98] I. Pomeranz and S.M. Reddy. On N-detection Test Sets and Variable N-Detection Test Sets for Transition Faults. In *IEEE VLSI Test Symposium (VTS)*, 1999.

[99] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast. Cell-Aware Test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33:1396–1409, 2014.

[100] F. Hapke, R. Krenz-Baath, A. Glowatz, J. Schloeffel, H. Hashempour, S. Eichenberger, C. Hora, and D. Adolfsson. Defect-oriented cell-aware ATPG and fault simulation for industrial cell libraries and designs. In *IEEE International Test Conference (ITC)*, 2009.

[101] Synopsys. CMGen User Guide, 2022.

[102] Siemens EDA. Tessent Shell Reference Manual, 2019.

[103] S. Kundu, G. Bhargava, L. Endrinal, and L. Ranganathan. Using Custom Fault Models to Improve Understanding of Silicon Failures. In *IEEE International Test Conference (ITC)*, 2022.

[104] RISC-V International. RISC-V Opcodes. https://github.com/riscv/riscv-opcodes, 2022.

[105] RISC-V International. Volume 2, Privileged Specification version 20211203. https://riscv.org/technical/specifications/. Accessed 13/02/2024.

[106] C. Li and S. Dey. Software-Based Self-Testing Methodology for Processor Cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:369–380, 2001.

[107] A. Benso, A. Bosio, P. Prinetto, and A. Savino. An On-Line Software-Based Self-Test Framework for Microprocessor Cores. In *IEEE International Conference on Design and Test of Integrated Systems in Nanoscale Technology (DTIS)*, 2006.

[108] S. Gurumurthy, S. Vasudevan, and J.A. Abraham. Automatic Generation of Instruction Sequences Targeting Hard-to-Detect Structural Faults in a Processor. In *IEEE International Test Conference (ITC)*, 2006.

[109] S. Gurumurthy, R. Vemu, J.A. Abraham, and D.G. Saab. Automatic Generation of Instructions to Robustly Test Delay Defects in Processors. In *IEEE European Test Symposium (ETS)*, 2007.

[110] M. Grosso, W.J.H. Perez, D. Ravotto, E. Sanchez, M. Sonza Reorda, and J.V. Medina. A Software-Based Self-Test Methodology for System Peripherals. In *IEEE European Test Symposium (ETS)*, 2010.

[111] P. Sha'afi Kabiri and Z. Navabi. Effective RT-level Software-Based Self-Testing of Embedded Processor Cores. In *International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2012.

[112] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos. Software-Based Self-Test for Small Caches in Microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33:1991–2004, 2014.

[113] A. Jasnetski, R. Ubar, and A. Tsertov. Automated software-based self-test generation for microprocessors. In *Mixed Design of Integrated Circuits and Systems (MIXES)*, 2017.

[114] P. Georgiou, X. Kavousianos, R. Cantoro, and M. Sonza Reorda. Fault-independent test-generation for software-based self-testing. In *IEEE International Symposium on On-Line Testing And Robust System Design (IOLTS)*, 2018.

[115] A.S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik. High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors. *Journal of Electronic Testing*, 36:87–103, 2020.

[116] R. Cantoro, F. Garau, P. Girard, N. Kolahimahmoudi, S. Sartoni, M. Sonza Reorda, and A. Virazel. Effective techniques for automatically improving the transition delay fault coverage of Self-Test Libraries. In *IEEE European Test Symposium (ETS)*, 2022.

[117] H. Cheng, C.-J. Li, H.-L. Chen, and J.-L. Huang. Bdd-based self-test program generation for processor cores. In *IEEE International Test Conference in Asia (ITC-Asia)*, 2023.

[118] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta. A Software-Based Self Test of CUDA Fermi GPUs. In *IEEE European Test Symposium (ETS)*, 2013.

[119] B. Du, J.E.R. Condia, M. Sonza Reorda, and L. Sterpone. About the functional test of the GPGPU scheduler. In *International Symposium on On-Line Testing And Robust System Design (IOLTS)*, 2018.

[120] J.E.R. Condia and M. Sonza Reorda. Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach. In *International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019.

[121] S. Di Carlo, J.E.R. Condia, and M. Sonza Reorda. An On-Line Testing Technique for the Scheduler Memory of a GPGPU. *IEEE Access*, 8:16893–16912, 2020.

[122] J.E.R. Condia and M. Sonza Reorda. Testing the Divergence Stack Memory on GPGPUs: A Modular in-Field Test Strategy. In *International Conference on Very Large Scale Integration (VLSI-SOC)*, 2020.

[123] J.-D. Guerrero-Balaguera, J.E.R. Condia, and M. Sonza Reorda. On the Functional Test of Special Function Units in GPUs. In *IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2021.

[124] J.-D. Guerrero-Balaguera, J.E.R. Condia, and M. Sonza Reorda. A New Method to Generate Software Test Libraries for In-Field GPU Testing Resorting to High-Level Languages. In *VLSI Test Symposium (VTS)*, 2022.

[125] J.E.R. Condia, F.A. da Silva, A.Ç. Bağbaga, J.-D. Guerrero-Balaguera, S. Hamdioui, C. Sauer, and M. Sonza Reorda. Using STLs for Effective In-Field Test of GPUs. *IEEE Design & Test*, 40:109–117, 2023.

[126] RISC-V International. Volume 1, Unprivileged Specification version 20191213. https://riscv.org/technical/specifications/. Accessed 13/02/2024.

[127] Y. Zhang, A. Rezine, P. Eles, and Z. Peng. Automatic Test Program Generation for Out-of-Order Superscalar Processors. In *IEEE Asian Test Symposium (ATS)*, 2012.

[128] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, A. Sansonetti, and G. Squillero. Software-Based Self-Test Techniques for Dual-Issue Embedded Processors. *IEEE Transactions on Emerging Topics in Computing*, 8:464–477, 2020.

[129] A. Ruospo, R. Cantoro, E. Sanchez, P.D. Schiavone, A. Garofalo, and L. Benini. On-line Testing for Autonomous Systems driven by RISC-V Processor Design Verification. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2019.

[130] Minres. DBT-RISE-RISCV Instruction Set Simulator. https://github.com/Minres/DBT-RISE-RISCV. Accessed 13/02/2024.

[131] European Computer Manufacturers Association (ECMA). The JSON Data Interchange Standard. https://www.json.org/json-en.html. Accessed 13/02/2024.

[132] Yosys Headquarters. PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32. Accessed 13/02/2024.

[133] Marcelo Samsoniuk. DarkRISCV. https://github.com/darklife/darkriscv. Accessed 13/02/2024.

[134] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez. A Suitability Analysis of Software Based Testing Strategies for the On-line Testing of Artificial Neural Networks Applications in Embedded Devices. In *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021.

[135] S. Alcaide, L. Kosmidis, H. Tabani, C. Hernandez, J. Abella, and F.J. Cazorla. Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain. *IEEE Micro*, 38, 2018.

[136] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella. High-Integrity GPU Designs for Critical Real-Time Automotive Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[137] I.S. Olmedo, N. Capodieci, and R. Cavicchioli. A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS. In *Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2018.

[138] M. Benito, M.M. Trompouki, L. Kosmidis, J.D. Garcia, S. Carretero, and K. Wenger. Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[139] C. Adams, A. Spain, J. Parker, M. Hevert, J. Roach, and D. Cotten. Towards an Integrated GPU Accelerated SoC as a Flight Computer for Small Satellites. In *IEEE Aerospace Conference (AESS)*, 2019.

[140] L. Kosmidis, J. Lachaize, J. Abella, O. Notebaert, F.J. Cazorla, and D. Steenari. GPU4S: Embedded GPUs in Space. In *Euromicro Conference on Digital System Design (DSD)*, 2019.

[141] J. Athavale, A. Baldovin, and M. Paulitsch. Trends and Functional Safety Certification Strategies for Advanced Railway Automation Systems. In *IEEE International Reliability Physics Symposium (IRPS)*, 2020.

[142] NVIDIA Corporation. Industrial-Scale AI. https://www.nvidia.com/en-us/industries/industrial-sector/. Accessed 15/02/2024.

[143] Synopsys. Silicon Lifecycle Management. https://www.synopsys.com/solutions/silicon-lifecycle-management.html. Accessed 15/02/2024.

[144] J. Choquette. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro*, 43:9–17, 2023.

[145] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

[146] NVIDIA. NVIDIA Tesla V100 GPU Architecture White Paper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. Accessed 15/02/2024.

[147] J.E.R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone. FlexGripPlus: An Improved GPGPU Model to Support Reliability Analysis. *Microelectronics Reliability*, 109, 2020.

[148] J.E.R. Condia. FlexGrip: an Open Source GPU Model for Reliability Evaluation and Micro Architectural Simulation. https://github.com/Jerc007/Open-GPGPU-FlexGrip-, 2019. Accessed 16/02/2024.

[149] J.-D. Guerrero-Balaguera, J.E.R. Condia, and M. Sonza Reorda. A Compaction Method for STLs for GPU In-Field Test. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.

[150] R. Metta, R.K. Medicherla, and S. Chakraborty. BMC+Fuzz: Efficient and Effective Test Generation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.

[151] N.I. Deligiannis. LAP: Layout Aware Pairing. https://github.com/NikosDelijohn/LAP, 2023. Accessed 29/02/2024.

Appendix A

Example: Combinational ATPG via SAT-solving

Step 1: Unit Propagation $\longmapsto$ O = 1

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge (\text{sa1}) \wedge (\overline{\text{sa1}} \vee \overline{d_{\text{fm}}}) \wedge (\text{sa1} \vee d_{\text{fm}}) \wedge \\ & (\overline{d_{\text{gm}}} \vee \overline{d_{\text{fm}}} \vee \overline{\text{O}}) \wedge (d_{\text{gm}} \vee d_{\text{fm}} \vee \overline{\text{O}}) \wedge \cancel{(d_{\text{gm}} \vee \overline{d_{\text{fm}}} \vee \text{O})} \wedge \cancel{(\overline{d_{\text{gm}}} \vee d_{\text{fm}} \vee \text{O})} \wedge \cancel{(\text{O})}\end{aligned}$$

$$\Longleftrightarrow$$

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge (\text{sa1}) \wedge (\overline{\text{sa1}} \vee \overline{d_{\text{fm}}}) \wedge (\text{sa1} \vee d_{\text{fm}}) \wedge \\ & (\overline{d_{\text{gm}}} \vee \overline{d_{\text{fm}}}) \wedge (d_{\text{gm}} \vee d_{\text{fm}})\end{aligned} \tag{A.1}$$

Step 2: Unit Propagation $\longmapsto$ sa1 = 1

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge \cancel{(\text{sa1})} \wedge (\overline{\text{sa1}} \vee \overline{d_{\text{fm}}}) \wedge \cancel{(\text{sa1} \vee d_{\text{fm}})} \wedge \\ & (\overline{d_{\text{gm}}} \vee \overline{d_{\text{fm}}}) \wedge (d_{\text{gm}} \vee d_{\text{fm}})\end{aligned}$$

$$\Longleftrightarrow$$

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge (\overline{d_{\text{fm}}}) \wedge (\overline{d_{\text{gm}}} \vee \overline{d_{\text{fm}}}) \wedge (d_{\text{gm}} \vee d_{\text{fm}})\end{aligned} \tag{A.2}$$

Step 3: Unit Propagation $\longmapsto d_{\mathbf{fm}} = 0$

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge (\overline{d_{\text{fm}}}) \wedge (\overline{d_{\text{gm}}} \vee \overline{d_{\text{fm}}}) \wedge (d_{\text{gm}} \vee d_{\text{fm}}) \\ & \Longleftrightarrow \\ \text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge (d_{\text{gm}})\end{aligned} \tag{A.3}$$

Step 4: Unit Propagation $\longmapsto d_{\mathbf{gm}} = 1$

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}} \vee \overline{d_{\text{gm}}}) \wedge (c_{\text{gm}} \vee d_{\text{gm}}) \wedge \\ & (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \wedge (d_{\text{gm}}) \\ & \Longleftrightarrow \\ \text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}}) \wedge (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge \\ & (\overline{b} \vee c_{\text{fm}})\end{aligned} \tag{A.4}$$

Step 5: Unit Propagation $\longmapsto c_{\mathbf{gm}} = 0$

$$\begin{aligned}\text{CNF} \;=\; & (a \vee b \vee \overline{c_{\text{gm}}}) \wedge (\overline{a} \vee c_{\text{gm}}) \wedge (\overline{b} \vee c_{\text{gm}}) \wedge (\overline{c_{\text{gm}}}) \wedge (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge \\ & (\overline{b} \vee c_{\text{fm}}) \\ & \Longleftrightarrow \\ \text{CNF} \;=\; & (\overline{a}) \wedge (\overline{b}) \wedge (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}})\end{aligned} \tag{A.5}$$

Step 6: Unit Propagation $\longmapsto a = 0$

$$\begin{aligned}\text{CNF} \;=\; & (\overline{a}) \wedge (\overline{b}) \wedge (a \vee b \vee \overline{c_{\text{fm}}}) \wedge (\overline{a} \vee c_{\text{fm}}) \wedge (\overline{b} \vee c_{\text{fm}}) \\ & \Longleftrightarrow \\ \text{CNF} \;=\; & (\overline{b}) \wedge (b \vee \overline{c_{\text{fm}}}) \wedge (\overline{b} \vee c_{\text{fm}})\end{aligned} \tag{A.6}$$

Step 7: Unit Propagation $\longmapsto b = 0$

$$\text{CNF} \;=\; (\overline{b}) \wedge (b \vee \overline{c_{\text{fm}}}) \wedge (\overline{b} \vee c_{\text{fm}})$$

$$\Longleftrightarrow$$

$$\text{CNF} \;=\; (\overline{c_{\text{fm}}}) \tag{A.7}$$

Step 8: Unit Propagation $\longmapsto$ $c_{\mathbf{fm}} = 0$

$$\text{CNF} \;=\; \text{SAT} \tag{A.8}$$

Appendix B

Example: Sequential ATPG via BMC-solving

Base Case: $\mathbf{CNF}_0 = I^0 \wedge P^0$

The base case of the problem for $k = 0$ is comprised of the initial state and the desired property. This translates to checking whether the desired target property be reached directly from the initial state. Considering that the initial state of the circuit is 0, then the state I^0 is $(\overline{c_{\text{gm}}}) \wedge (\overline{c_{\text{fm}}})$. The desired property is for a difference to occur in the output of the XOR gate of the miter circuit (i.e., for the fault to propagate to a primary output). Hence, the target state P^0 would be (O). Hence, the generated CNF would be:

$$\text{CNF}_0 = (\overline{c_{\text{gm}}}) \wedge (\overline{c_{\text{fm}}}) \wedge (O) \tag{B.1}$$

This CNF is trivially Satisfiable, and its model is <0, 0, 1>. However, the transition relation is not considered for the base case, and thus, the circuit behavior is disregarded. To get around this issue, we latch the miter output and initialize the latch with the value of 0. This is evident in Figure 1.8 by the insertion of the D Flip-Flop at the output of the miter's XOR gate. The Flip-Flop will capture the output of the XOR gate, and its state translates to whether the fault effect has been observed, i.e., if the fault has been detected.

After this modification, the initial state I^0 is now $(\overline{c_{\text{gm}}}) \wedge (\overline{c_{\text{fm}}}) \wedge (\overline{P})$. The extra unit clause is required in order to explicitly encode that the target state is not reached without considering the transition relation. The Equation (B.1) is now corrected and

transformed to:

$$\text{CNF}_0 = (\overline{c_{\text{gm}}}) \wedge (\overline{c_{\text{fm}}}) \wedge \underbrace{(\overline{P}) \wedge (P)}_{\text{Contradiction}} \tag{B.2}$$

The CNF now is Unsatisfiable as we intended, and thus, we can proceed with the first unrolling of the circuit to check for the desired property. Before we move to the next step, let us see the CNF formula for the transition relation T of the example circuit. Below we present the general formula for T. The reader should note that the exponent on the literals indicates the timeframe to which they correspond.

$$\begin{aligned} T^{i\to i+1} = {} & \overbrace{(\overline{a} \vee \overline{c_{\text{gm}}} \vee \overline{b_{\text{gm}}}) \wedge (a \vee c_{\text{gm}} \vee \overline{b_{\text{gm}}}) \wedge (a \vee \overline{c_{\text{gm}}} \vee b_{\text{gm}}) \wedge (\overline{a} \vee c_{\text{gm}} \vee b_{\text{gm}})}^{\text{XOR}^{\text{gm}}} \\ & \wedge \overbrace{(\overline{a} \vee \overline{c_{\text{fm}}} \vee \overline{b_{\text{fm}}}) \wedge (a \vee c_{\text{fm}} \vee \overline{b_{\text{fm}}}) \wedge (a \vee \overline{c_{\text{fm}}} \vee b_{\text{fm}}) \wedge (\overline{a} \vee c_{\text{fm}} \vee b_{\text{fm}})}^{\text{XOR}^{\text{fm}}} \\ & \wedge \overbrace{(\overline{c_{\text{gm}}} \vee \overline{\text{sa0}} \vee \overline{O}) \wedge (c_{\text{gm}} \vee \text{sa0} \vee \overline{O}) \wedge (c_{\text{gm}} \vee \overline{\text{sa0}} \vee O) \wedge (\overline{c_{\text{gm}}} \vee \text{sa0} \vee O)}^{\text{XOR}} \\ & \wedge \overbrace{(\overline{\text{sa0}})}^{\text{stuck-at 0}} \wedge \overbrace{(b_{\text{gm}} \vee \overline{c_{\text{gm}}^{i+1}}) \wedge (\overline{b_{\text{gm}}} \vee c_{\text{gm}}^{i+1})}^{\text{D-FF}_{\text{gm}}^{i\to i+1}} \wedge \overbrace{(b_{\text{fm}} \vee \overline{c_{\text{fm}}^{i+1}}) \wedge (\overline{b_{\text{fm}}} \vee c_{\text{fm}}^{i+1})}^{\text{D-FF}_{\text{fm}}^{i\to i+1}} \\ & \wedge \overbrace{(O \vee \overline{P^{i+1}}) \wedge (\overline{O} \vee P^{i+1})}^{\text{D-FF}_{\text{Miter}}^{i\to i+1}} \end{aligned} \tag{B.3}$$

One should note that the D Flip-Flops are encoded as buffers (see Table 1.1). Each flip-flop of a sequential circuit is removed and its input is mapped to a literal corresponding to the current timeframe whereas its output is mapped to a literal corresponding to the following timeframe. This is also shown in Figure B.1 where a transition from one timeframe to another is shown for an arbitrary sequential circuit.

k = 1) $\text{CNF}_1 = I^0 \wedge T^{0\to 1} \wedge P^1$

During the first unrolling of the circuit, the transition relation from the initial state to timeframe 1 will be considered. The corresponding CNF will be the following:

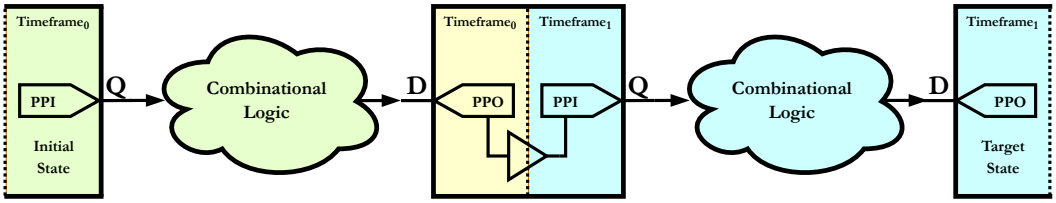

Fig. B.1 Circuit unrolling for timeframe 0 to 1.

$$
\begin{aligned}
\mathrm{CNF}_1 &= I^0 \wedge T^{0\to1} \wedge P^1 \\
&= \overbrace{(\overline{c_{\mathrm{gm}}}) \wedge (\overline{c_{\mathrm{fm}}}) \wedge (\overline{P})}^{I^0} \\
&\wedge (\overline{a} \vee \overline{c_{\mathrm{gm}}} \vee \overline{b_{\mathrm{gm}}}) \wedge (a \vee c_{\mathrm{gm}} \vee \overline{b_{\mathrm{gm}}}) \wedge (a \vee \overline{c_{\mathrm{gm}}} \vee b_{\mathrm{gm}}) \wedge (\overline{a} \vee c_{\mathrm{gm}} \vee b_{\mathrm{gm}}) \\
&\wedge (\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee c_{\mathrm{fm}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \wedge (\overline{a} \vee c_{\mathrm{fm}} \vee b_{\mathrm{fm}}) \\
&\wedge (\overline{c_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}} \vee \overline{O}) \wedge (c_{\mathrm{gm}} \vee \mathrm{sa0} \vee \overline{O}) \wedge (c_{\mathrm{gm}} \vee \overline{\mathrm{sa0}} \vee O) \wedge (\overline{c_{\mathrm{gm}}} \vee \mathrm{sa0} \vee O) \\
&\wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
&\wedge \underbrace{(P^1)}_{P^1}
\end{aligned}
\quad \Big\} T^{0\to1}
\tag{B.4}
$$

Step 1.1: Unit Propagation $\longmapsto$ $c_{\mathrm{gm}} = 0$

$$
\begin{aligned}
\mathrm{CNF}_1 &= \cancel{(\overline{c_{\mathrm{gm}}})} \wedge (\overline{c_{\mathrm{fm}}}) \wedge (\overline{P}) \\
&\wedge \cancel{(\overline{a} \vee \overline{c_{\mathrm{gm}}} \vee \overline{b_{\mathrm{gm}}})} \wedge (a \vee \cancel{c_{\mathrm{gm}}} \vee \overline{b_{\mathrm{gm}}}) \wedge \cancel{(a \vee \overline{c_{\mathrm{gm}}} \vee b_{\mathrm{gm}})} \wedge (\overline{a} \vee \cancel{c_{\mathrm{gm}}} \vee b_{\mathrm{gm}}) \\
&\wedge (\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee c_{\mathrm{fm}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \wedge (\overline{a} \vee c_{\mathrm{fm}} \vee b_{\mathrm{fm}}) \\
&\wedge \cancel{(\overline{c_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}} \vee \overline{O})} \wedge (\cancel{c_{\mathrm{gm}}} \vee \mathrm{sa0} \vee \overline{O}) \wedge (\cancel{c_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}} \vee O) \wedge \cancel{(\overline{c_{\mathrm{gm}}} \vee \mathrm{sa0} \vee O)} \\
&\wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
&\wedge (P^1) \\
&\Longleftrightarrow
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{CNF}_1 = {} & (\overline{c_{\mathrm{fm}}}) \wedge (\overline{P}) \\
& \wedge (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee c_{\mathrm{fm}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \wedge (\overline{a} \vee c_{\mathrm{fm}} \vee b_{\mathrm{fm}}) \\
& \wedge (\mathrm{sa0} \vee \overline{O}) \wedge (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
& \wedge (P^1)
\end{aligned}
\tag{B.5}
$$

Step 1.2: Unit Propagation $\longmapsto$ $c_{\mathrm{fm}} = 0$

$$
\begin{aligned}
\mathrm{CNF}_1 = {} & \cancel{(\overline{c_{\mathrm{fm}}})} \wedge (\overline{P}) \\
& \wedge (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge \cancel{(\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}})} \wedge (a \vee \cancel{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge \cancel{(a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}})} \wedge (\overline{a} \vee \cancel{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \\
& \wedge (\mathrm{sa0} \vee \overline{O}) \wedge (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
& \wedge (P^1)
\end{aligned}
$$

$$
\Longleftrightarrow
$$

$$
\begin{aligned}
\mathrm{CNF}_1 = {} & (\overline{P}) \\
& \wedge (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (\mathrm{sa0} \vee \overline{O}) \wedge (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
& \wedge (P^1)
\end{aligned}
\tag{B.6}
$$

Step 1.3: Unit Propagation $\longmapsto$ P = 0

$$
\begin{aligned}
\mathrm{CNF}_1 = \; & \cancel{(\overline{P})} \\
& \wedge\ (a \vee \overline{b_{\mathrm{gm}}}) \ \wedge\ (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge\ (a \vee \overline{b_{\mathrm{fm}}}) \ \wedge\ (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge\ (\mathrm{sa0} \vee \overline{O}) \ \wedge\ (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \ \wedge\ (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \ \wedge\ (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \ \wedge\ (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \ \wedge\ (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \ \wedge\ (\overline{O} \vee P^1) \\
& \wedge (P^1)
\end{aligned}
$$

$$\Longleftrightarrow$$

$$
\begin{aligned}
\mathrm{CNF}_1 = \; & \\
& (a \vee \overline{b_{\mathrm{gm}}}) \ \wedge\ (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge\ (a \vee \overline{b_{\mathrm{fm}}}) \ \wedge\ (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge\ (\mathrm{sa0} \vee \overline{O}) \ \wedge\ (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \ \wedge\ (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \ \wedge\ (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \ \wedge\ (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \ \wedge\ (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \ \wedge\ (\overline{O} \vee P^1) \\
& \wedge (P^1)
\end{aligned}
\tag{B.7}
$$

Step 1.4: Unit Propagation $\longmapsto$ sa0 = 0

$$
\begin{aligned}
\mathrm{CNF}_1 = \; & \\
& (a \vee \overline{b_{\mathrm{gm}}}) \ \wedge\ (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge\ (a \vee \overline{b_{\mathrm{fm}}}) \ \wedge\ (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge\ \cancel{(\mathrm{sa0} \vee \overline{O})} \ \wedge\ \cancel{(\overline{\mathrm{sa0}} \vee O)} \\
& \wedge \cancel{(\overline{\mathrm{sa0}})} \ \wedge\ (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \ \wedge\ (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \ \wedge\ (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \ \wedge\ (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \ \wedge\ (\overline{O} \vee P^1) \\
& \wedge (P^1)
\end{aligned}
$$

$$\Longleftrightarrow$$

$$
\begin{aligned}
\mathrm{CNF}_1 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (\overline{O}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
&\wedge (P^1)
\end{aligned}
\tag{B.8}
$$

Step 1.5: Unit Propagation $\longmapsto$ $O = 0$

$$
\begin{aligned}
\mathrm{CNF}_1 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge \cancel{(\overline{O})} \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (\cancel{O} \vee \overline{P^1}) \wedge \cancel{(\overline{O} \vee P^1)} \\
&\wedge (P^1)
\end{aligned}
$$

$$\Longleftrightarrow$$

$$
\begin{aligned}
\mathrm{CNF}_1 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\left.\begin{aligned} &\wedge (\overline{P^1}) \\ &\wedge (P^1) \end{aligned}\right\} \text{Contradiction}
\end{aligned}
$$

$$\Longleftrightarrow$$

$$\mathrm{CNF}_1 = \mathrm{UNSAT} \tag{B.9}$$

k = 2) $\mathbf{CNF}_2 = I^0 \wedge T^{0\to 1} \wedge T^{1\to 2} \wedge P^2$

During the second unrolling of the circuit, the transition relation from the initial state to timeframe 1 and from timeframe 1 to timeframe 2 will be considered, and the resulting CNF will be the following:

$$
\begin{aligned}
\mathrm{CNF}_2 &= I^0 \wedge T^{0\to 1} \wedge T^{1\to 2} \wedge P^2 \\
&= \overbrace{(\overline{c_{\mathrm{gm}}}) \wedge (\overline{c_{\mathrm{fm}}}) \wedge (\overline{P})}^{I^0} \\
&\left.\begin{array}{l}
\wedge (\overline{a} \vee \overline{c_{\mathrm{gm}}} \vee \overline{b_{\mathrm{gm}}}) \wedge (a \vee c_{\mathrm{gm}} \vee \overline{b_{\mathrm{gm}}}) \wedge (a \vee \overline{c_{\mathrm{gm}}} \vee b_{\mathrm{gm}}) \wedge (\overline{a} \vee c_{\mathrm{gm}} \vee b_{\mathrm{gm}}) \\
\wedge (\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee c_{\mathrm{fm}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \wedge (\overline{a} \vee c_{\mathrm{fm}} \vee b_{\mathrm{fm}}) \\
\wedge (\overline{c_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}} \vee \overline{O}) \wedge (c_{\mathrm{gm}} \vee \mathrm{sa0} \vee \overline{O}) \wedge (c_{\mathrm{gm}} \vee \overline{\mathrm{sa0}} \vee O) \wedge (\overline{c_{\mathrm{gm}}} \vee \mathrm{sa0} \vee O) \\
\wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1)
\end{array}\right\} T^{0\to 1} \\
&\left.\begin{array}{l}
\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2)
\end{array}\right\} T^{1\to 2} \\
&\wedge \underbrace{(P^2)}_{P^2}
\end{aligned}
\tag{B.10}
$$

Step 2.1: Unit Propagation $\longmapsto$ $c_{\mathbf{gm}} = \mathbf{0}$

$$
\begin{aligned}
\mathrm{CNF}_2 = {} & \cancel{(\overline{c_{\mathrm{gm}}})} \wedge (\overline{c_{\mathrm{fm}}}) \wedge (\overline{P}) \\
& \wedge \cancel{(\overline{a} \vee \overline{c_{\mathrm{gm}}} \vee \overline{b_{\mathrm{gm}}})} \wedge (a \vee \cancel{c_{\mathrm{gm}}} \vee \overline{b_{\mathrm{gm}}}) \wedge \cancel{(a \vee \overline{c_{\mathrm{gm}}} \vee b_{\mathrm{gm}})} \wedge (\overline{a} \vee \cancel{c_{\mathrm{gm}}} \vee b_{\mathrm{gm}}) \\
& \wedge (\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee c_{\mathrm{fm}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \wedge (\overline{a} \vee c_{\mathrm{fm}} \vee b_{\mathrm{fm}}) \\
& \wedge \cancel{(\overline{c_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}} \vee \overline{O})} \wedge (\cancel{c_{\mathrm{gm}}} \vee \mathrm{sa0} \vee \overline{O}) \wedge (\cancel{c_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}} \vee O) \wedge \cancel{(\overline{c_{\mathrm{gm}}} \vee \mathrm{sa0} \vee O)} \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
\\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
& \wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
& \wedge (P^2) \\
\\
& \Longleftrightarrow \\
\\
\mathrm{CNF}_2 = {} & (\overline{c_{\mathrm{fm}}}) \wedge (\overline{P}) \\
& \wedge (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee c_{\mathrm{fm}} \vee \overline{b_{\mathrm{fm}}}) \wedge (a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \wedge (\overline{a} \vee c_{\mathrm{fm}} \vee b_{\mathrm{fm}}) \\
& \wedge (\mathrm{sa0} \vee \overline{O}) \wedge (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
\\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
& \wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
& \wedge (P^2)
\end{aligned}
\tag{B.11}
$$

Step 2.2: Unit Propagation $\longmapsto$ $c_{\mathrm{fm}} = 0$

$$
\begin{aligned}
\mathrm{CNF}_2 = {} & \cancel{(\overline{c_{\mathrm{fm}}})} \wedge (\overline{P}) \\
& \wedge (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge \cancel{(\overline{a} \vee \overline{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}})} \wedge (a \vee \cancel{c_{\mathrm{fm}}} \vee \overline{b_{\mathrm{fm}}}) \wedge \cancel{(a \vee \overline{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}})} \wedge (\overline{a} \vee \cancel{c_{\mathrm{fm}}} \vee b_{\mathrm{fm}}) \\
& \wedge (\mathrm{sa0} \vee \overline{O}) \wedge (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
\\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
& \wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
& \wedge (P^2) \\
& \qquad \Longleftrightarrow \\
\mathrm{CNF}_2 = {} & (\overline{P}) \\
& \wedge (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (\mathrm{sa0} \vee \overline{O}) \wedge (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
\\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
& \wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
& \wedge (P^2)
\end{aligned}
\tag{B.12}
$$

Step 2.3: Unit Propagation $\longmapsto$ $P = 0$

$$
\begin{aligned}
\mathrm{CNF}_2 = {} & \cancel{(\overline{P})} \\
& \wedge\ (a \vee \overline{b_{\mathrm{gm}}})\ \wedge\ (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge\ (a \vee \overline{b_{\mathrm{fm}}})\ \wedge\ (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge\ (\mathrm{sa0} \vee \overline{O})\ \wedge\ (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}})\ \wedge\ (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}})\ \wedge\ (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}})\ \wedge\ (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}})\ \wedge\ (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1})\ \wedge\ (\overline{O} \vee P^1) \\
\\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
& \wedge\ (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge\ (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1})\ \wedge\ (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1})\ \wedge\ (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1)\ \wedge\ (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
& \wedge (\overline{\mathrm{sa0}^1})\ \wedge\ (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}})\ \wedge\ (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}})\ \wedge\ (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}})\ \wedge\ (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \wedge (O^1 \vee \overline{P^2})\ \wedge\ (\overline{O^1} \vee P^2) \\
& \wedge (P^2) \\
\\
& \Longleftrightarrow \\
\\
\mathrm{CNF}_2 = {} & \\
& (a \vee \overline{b_{\mathrm{gm}}})\ \wedge\ (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge\ (a \vee \overline{b_{\mathrm{fm}}})\ \wedge\ (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge\ (\mathrm{sa0} \vee \overline{O})\ \wedge\ (\overline{\mathrm{sa0}} \vee O) \\
& \wedge (\overline{\mathrm{sa0}})\ \wedge\ (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}})\ \wedge\ (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}})\ \wedge\ (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}})\ \wedge\ (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \wedge (O \vee \overline{P^1})\ \wedge\ (\overline{O} \vee P^1) \\
\\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
& \wedge\ (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge\ (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1})\ \wedge\ (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1})\ \wedge\ (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1)\ \wedge\ (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
& \wedge (\overline{\mathrm{sa0}^1})\ \wedge\ (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}})\ \wedge\ (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}})\ \wedge\ (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}})\ \wedge\ (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \wedge (O^1 \vee \overline{P^2})\ \wedge\ (\overline{O^1} \vee P^2) \\
& \wedge (P^2)
\end{aligned}
\tag{B.13}
$$

Step 2.4: Unit Propagation $\longmapsto$ sa0 = 0

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (\cancel{\mathrm{sa0} \vee \overline{O}}) \wedge (\cancel{\overline{\mathrm{sa0}} \vee O}) \\
&\wedge (\cancel{\overline{\mathrm{sa0}}}) \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
&\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2) \\
&\qquad\qquad \Longleftrightarrow \\
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (\overline{O}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (O \vee \overline{P^1}) \wedge (\overline{O} \vee P^1) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
&\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2)
\end{aligned}
\tag{B.14}
$$

Step 2.5: Unit Propagation $\longmapsto$ $O = 0$

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge \cancel{(\overline{O})} \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (\cancel{O} \vee \overline{P^1}) \wedge \cancel{(\overline{O} \vee P^1)} \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
&\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2) \\
\\
&\Longleftrightarrow \\
\\
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge (\overline{P^1}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
&\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2)
\end{aligned}
\tag{B.15}
$$

Step 2.6: Unit Propagation $\longmapsto$ P^1 = 0

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
&\wedge \cancel{(\overline{P^1})} \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
&\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2) \\
\\
&\Longleftrightarrow \\
\\
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&(\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \mathrm{sa0}^1 \vee \overline{O^1}) \wedge (c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1) \wedge (\overline{c^1_{\mathrm{gm}}} \vee \mathrm{sa0}^1 \vee O^1) \\
&\wedge (\overline{\mathrm{sa0}^1}) \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2)
\end{aligned}
\tag{B.16}
$$

Step 2.7: Unit Propagation $\longmapsto$ $\mathrm{sa0}^1 = 0$

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge \cancel{(\overline{c^1_{\mathrm{gm}}} \vee \overline{\mathrm{sa0}^1} \vee \overline{O^1})} \wedge (c^1_{\mathrm{gm}} \vee \cancel{\mathrm{sa0}^1} \vee \overline{O^1}) \wedge \cancel{(c^1_{\mathrm{gm}} \vee \overline{\mathrm{sa0}^1} \vee O^1)} \wedge (\overline{c^1_{\mathrm{gm}}} \vee \cancel{\mathrm{sa0}^1} \vee O^1) \\
&\wedge \cancel{(\overline{\mathrm{sa0}^1})} \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2) \\
\\
&\Longleftrightarrow \\
\\
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (c^1_{\mathrm{gm}} \vee \overline{O^1}) \wedge (\overline{c^1_{\mathrm{gm}}} \vee O^1) \\
&\wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge (\overline{O^1} \vee P^2) \\
&\wedge (P^2)
\end{aligned}
\tag{B.17}
$$

Step 2.8: Unit Propagation $\longmapsto$ P^2 = 1

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (c^1_{\mathrm{gm}} \vee \overline{O^1}) \wedge (\overline{c^1_{\mathrm{gm}}} \vee O^1) \\
&\wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1 \vee \overline{P^2}) \wedge \cancel{(\overline{O^1} \vee P^2)} \\
&\wedge \cancel{(P^2)}
\end{aligned}
$$

$$\Longleftrightarrow$$

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (c^1_{\mathrm{gm}} \vee \overline{O^1}) \wedge (\overline{c^1_{\mathrm{gm}}} \vee O^1) \\
&\wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge (O^1)
\end{aligned}
\tag{B.18}
$$

Step 2.9: Unit Propagation $\longmapsto$ O^1 = 1

$$
\begin{aligned}
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (c^1_{\mathrm{gm}} \vee \cancel{\overline{O^1}}) \wedge \cancel{(\overline{c^1_{\mathrm{gm}}} \vee O^1)} \\
&\wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\wedge \cancel{(O^1)} \\
\\
&\Longleftrightarrow \\
\\
\mathrm{CNF}_2 = \\
&(a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
&\wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
&\wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge (\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}}) \\
&\wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge (c^1_{\mathrm{gm}}) \\
&\wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})
\end{aligned}
\tag{B.19}
$$

Step 2.10: Unit Propagation $\longmapsto c^1_{\mathbf{gm}} = 1$

$$
\begin{aligned}
\mathrm{CNF}_2 = \quad & (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (b_{\mathrm{gm}} \vee \overline{c^1_{\mathrm{gm}}}) \wedge \cancel{(\overline{b_{\mathrm{gm}}} \vee c^1_{\mathrm{gm}})} \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{gm}}} \vee \overline{b^1_{\mathrm{gm}}}) \wedge \cancel{(a^1 \vee c^1_{\mathrm{gm}} \vee \overline{b^1_{\mathrm{gm}}})} \wedge (a^1 \vee \overline{c^1_{\mathrm{gm}}} \vee b^1_{\mathrm{gm}}) \wedge \cancel{(\overline{a^1} \vee c^1_{\mathrm{gm}} \vee b^1_{\mathrm{gm}})} \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge \cancel{(c^1_{\mathrm{gm}})} \\
& \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \qquad\qquad \Longleftrightarrow \\
\mathrm{CNF}_2 = \quad & (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (b_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \\
& \wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})
\end{aligned}
\tag{B.20}
$$

Step 2.11: Unit Propagation $\longmapsto b_{\mathbf{gm}} = 1$

$$
\begin{aligned}
\mathrm{CNF}_2 = \quad & (a \vee \overline{b_{\mathrm{gm}}}) \wedge (\overline{a} \vee b_{\mathrm{gm}}) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (b_{\mathrm{gm}}) \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \\
& \wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
& \qquad\qquad\qquad\qquad \Longleftrightarrow \\
\mathrm{CNF}_2 = \quad & (a) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \\
& \wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})
\end{aligned}
\tag{B.21}
$$

Step 2.12: Unit Propagation $\longmapsto a = 1$

$$
\begin{aligned}
\mathrm{CNF}_2 = \quad & (a) \\
& \wedge (a \vee \overline{b_{\mathrm{fm}}}) \wedge (\overline{a} \vee b_{\mathrm{fm}}) \\
& \wedge (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}}) \wedge (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
& \\
& \wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge (a^1 \vee b^1_{\mathrm{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}}) \wedge (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}}) \wedge (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
& \wedge (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}}) \wedge (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})
\end{aligned}
$$

$$\Longleftrightarrow$$

$$\begin{aligned}
\mathrm{CNF}_2 = \\
&(b_{\mathrm{fm}}) \\
&\wedge\ (b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}})\ \wedge\ (\overline{b_{\mathrm{fm}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge\ (a^1 \vee b^1_{\mathrm{gm}}) \\
&\wedge\ (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge\ (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}})\ \wedge\ (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}})\ \wedge\ (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}})\ \wedge\ (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})
\end{aligned} \tag{B.22}$$

Step 2.13: Unit Propagation $\longmapsto$ $b_{\mathrm{fm}} = 1$

$$\begin{aligned}
\mathrm{CNF}_2 = \\
&\cancel{(b_{\mathrm{fm}})} \\
&\wedge\ \cancel{(b_{\mathrm{fm}} \vee \overline{c^1_{\mathrm{fm}}})}\ \wedge\ (\cancel{\overline{b_{\mathrm{fm}}}} \vee c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge\ (a^1 \vee b^1_{\mathrm{gm}}) \\
&\wedge\ (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge\ (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}})\ \wedge\ (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}})\ \wedge\ (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}})\ \wedge\ (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \\
&\Longleftrightarrow \\
\mathrm{CNF}_2 = \\
&(c^1_{\mathrm{fm}}) \\
\\
&\wedge (\overline{a^1} \vee \overline{b^1_{\mathrm{gm}}}) \wedge\ (a^1 \vee b^1_{\mathrm{gm}}) \\
&\wedge\ (\overline{a^1} \vee \overline{c^1_{\mathrm{fm}}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee c^1_{\mathrm{fm}} \vee \overline{b^1_{\mathrm{fm}}})\ \wedge\ (a^1 \vee \overline{c^1_{\mathrm{fm}}} \vee b^1_{\mathrm{fm}})\ \wedge\ (\overline{a^1} \vee c^1_{\mathrm{fm}} \vee b^1_{\mathrm{fm}}) \\
&\wedge\ (b^1_{\mathrm{gm}} \vee \overline{c^2_{\mathrm{gm}}})\ \wedge\ (\overline{b^1_{\mathrm{gm}}} \vee c^2_{\mathrm{gm}})\ \wedge\ (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}})\ \wedge\ (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})
\end{aligned} \tag{B.23}$$

Step 2.14: Unit Propagation $\longmapsto c^1_{\text{fm}} = 1$

$$
\begin{aligned}
\text{CNF}_2 = \;& \cancel{(c^1_{\text{fm}})} \\
& \wedge (\overline{a^1} \vee \overline{b^1_{\text{gm}}}) \wedge (a^1 \vee b^1_{\text{gm}}) \\
& \wedge (\overline{a^1} \vee \cancel{\overline{c^1_{\text{fm}}}} \vee \overline{b^1_{\text{fm}}}) \wedge \cancel{(a^1 \vee c^1_{\text{fm}} \vee \overline{b^1_{\text{fm}}})} \wedge (a^1 \vee \cancel{\overline{c^1_{\text{fm}}}} \vee b^1_{\text{fm}}) \wedge \cancel{(\overline{a^1} \vee c^1_{\text{fm}} \vee b^1_{\text{fm}})} \\
& \wedge (b^1_{\text{gm}} \vee \overline{c^2_{\text{gm}}}) \wedge (\overline{b^1_{\text{gm}}} \vee c^2_{\text{gm}}) \wedge (b^1_{\text{fm}} \vee \overline{c^2_{\text{fm}}}) \wedge (\overline{b^1_{\text{fm}}} \vee c^2_{\text{fm}}) \\
& \qquad\qquad \Longleftrightarrow \\
\text{CNF}_2 = \;& (\overline{a^1} \vee \overline{b^1_{\text{gm}}}) \wedge (a^1 \vee b^1_{\text{gm}}) \\
& \wedge (\overline{a^1} \vee \overline{b^1_{\text{fm}}}) \wedge (a^1 \vee b^1_{\text{fm}}) \\
& \wedge (b^1_{\text{gm}} \vee \overline{c^2_{\text{gm}}}) \wedge (\overline{b^1_{\text{gm}}} \vee c^2_{\text{gm}}) \wedge (b^1_{\text{fm}} \vee \overline{c^2_{\text{fm}}}) \wedge (\overline{b^1_{\text{fm}}} \vee c^2_{\text{fm}})
\end{aligned}
\tag{B.24}
$$

Step 2.15: Decision $\longmapsto a^1 = 1$

$$
\begin{aligned}
\text{CNF}_2 = \;& (\cancel{\overline{a^1}} \vee \overline{b^1_{\text{gm}}}) \wedge \cancel{(a^1 \vee b^1_{\text{gm}})} \\
& \wedge (\cancel{\overline{a^1}} \vee \overline{b^1_{\text{fm}}}) \wedge \cancel{(a^1 \vee b^1_{\text{fm}})} \\
& \wedge (b^1_{\text{gm}} \vee \overline{c^2_{\text{gm}}}) \wedge (\overline{b^1_{\text{gm}}} \vee c^2_{\text{gm}}) \wedge (b^1_{\text{fm}} \vee \overline{c^2_{\text{fm}}}) \wedge (\overline{b^1_{\text{fm}}} \vee c^2_{\text{fm}}) \\
& \qquad\qquad \Longleftrightarrow \\
\text{CNF}_2 = \;& (\overline{b^1_{\text{gm}}}) \wedge (\overline{b^1_{\text{fm}}}) \\
& \wedge (b^1_{\text{gm}} \vee \overline{c^2_{\text{gm}}}) \wedge (\overline{b^1_{\text{gm}}} \vee c^2_{\text{gm}}) \wedge (b^1_{\text{fm}} \vee \overline{c^2_{\text{fm}}}) \wedge (\overline{b^1_{\text{fm}}} \vee c^2_{\text{fm}})
\end{aligned}
\tag{B.25}
$$

Step 2.16: Unit Propagation $\longmapsto b^1_{\text{gm}} = 0$

$$
\begin{aligned}
\text{CNF}_2 = \;& \cancel{(\overline{b^1_{\text{gm}}})} \wedge (\overline{b^1_{\text{fm}}}) \\
& \wedge (\cancel{b^1_{\text{gm}}} \vee \overline{c^2_{\text{gm}}}) \wedge \cancel{(\overline{b^1_{\text{gm}}} \vee c^2_{\text{gm}})} \wedge (b^1_{\text{fm}} \vee \overline{c^2_{\text{fm}}}) \wedge (\overline{b^1_{\text{fm}}} \vee c^2_{\text{fm}}) \\
& \qquad\qquad \Longleftrightarrow
\end{aligned}
$$

$$\mathrm{CNF}_2 = (\overline{b^1_{\mathrm{fm}}}) \wedge (\overline{c^2_{\mathrm{gm}}}) \wedge (b^1_{\mathrm{fm}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge (\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}}) \tag{B.26}$$

Step 2.17: Unit Propagation $\longmapsto b^1_{\mathrm{fm}} = 0$

$$\mathrm{CNF}_2 = \cancel{(\overline{b^1_{\mathrm{fm}}})} \wedge (\overline{c^2_{\mathrm{gm}}}) \wedge \cancel{(b^1_{\mathrm{fm}}} \vee \overline{c^2_{\mathrm{fm}}}) \wedge \cancel{(\overline{b^1_{\mathrm{fm}}} \vee c^2_{\mathrm{fm}})}$$

$$\Longleftrightarrow$$

$$\mathrm{CNF}_2 = (\overline{c^2_{\mathrm{gm}}}) \wedge (\overline{c^2_{\mathrm{fm}}}) \tag{B.27}$$

Step 2.18: Unit Propagation $\longmapsto \overline{c^2_{\mathrm{gm}}} = 0$

$$\mathrm{CNF}_2 = \cancel{(\overline{c^2_{\mathrm{gm}}})} \wedge (\overline{c^2_{\mathrm{fm}}})$$

$$\Longleftrightarrow$$

$$\mathrm{CNF}_2 = (\overline{c^2_{\mathrm{fm}}}) \tag{B.28}$$

Step 2.19: Unit Propagation $\longmapsto \overline{c^2_{\mathrm{fm}}} = 0$

$$\mathrm{CNF}_2 = \mathrm{SAT} \tag{B.29}$$

Appendix C

List of Publications by the Author

C.1 Journal Publications

2021

1. N. I. Deligiannis, R. Cantoro, M. Sonza Reorda, M. Traiola and E. Valea, "Towards the Integration of Reliability and Security Mechanisms to Enhance the Fault Resilience of Neural Networks," in IEEE Access, vol. 9, pp. 155998-156012, 2021, doi: 10.1109/ACCESS.2021.3129149.

2023

1. N. I. Deligiannis, T. Faller, R. Cantoro, T. Paxian, B. Becker and M. Sonza Reorda, "Automating the Generation of Programs Maximizing the Repeatable Constant Switching Activity in Microprocessor Units via MaxSAT," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 42, no. 11, pp. 4270-4281, Nov. 2023, doi: 10.1109/TCAD.2023.3252467.

2. N. I. Deligiannis, R. Cantoro and M. Sonza Reorda, "Automating the Generation of Programs Maximizing the Sustained Switching Activity in Microprocessor units via Evolutionary Techniques," in Elvesier Microprocessors and Microsystems, vol.98, 2023, doi: https://doi.org/10.1016/j.micpro.2023.104775.

2024

1. N. I. Deligiannis, T. Faller, J. E. R. Condia, R. Cantoro, B. Becker and M. Sonza Reorda, "Enhancing the Effectiveness of STLs for GPUs via Bounded Model Checking," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, **Submitted - Under Review**.

C.2 Conference Proceedings Publications

2020

1. R. Cantoro, N. I. Deligiannis, M. Sonza Reorda, M. Traiola and E. Valea, "Evaluating the Code Encryption Effects on Memory Fault Resilience," 2020 IEEE Latin-American Test Symposium (LATS), Maceio, Brazil, 2020, doi: 10.1109/LATS49555.2020.9093670.

2. R. Cantoro, N. I. Deligiannis, M. Sonza Reorda, M. Traiola and E. Valea, "Evaluating Data Encryption Effects on the Resilience of an Artificial Neural Network," 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Frascati, Italy, 2020, doi: 10.1109/DFT50435.2020.9250869.

2021

1. N. I. Deligiannis, R. Cantoro, M. Sauer, B. Becker and M. Sonza Reorda, "New Techniques for the Automatic Identification of Uncontrollable Lines in a CPU Core," 2021 IEEE VLSI Test Symposium (VTS), San Diego, CA, USA, 2021, doi: 10.1109/VTS50974.2021.9441040.

2. N. I. Deligiannis, R. Cantoro and M. Sonza Reorda, "Maximizing the Switching Activity of Different Modules Within a Processor Core via Evolutionary Techniques," 2021 Euromicro Conference on Digital System Design (DSD), Palermo, Italy, 2021, doi: 10.1109/DSD53832.2021.00086.

3. N. I. Deligiannis, R. Cantoro, T. Faller, T. Paxian, B. Becker and M. Sonza Reorda, "Effective SAT-based Solutions for Generating Functional Sequences

Maximizing the Sustained Switching Activity in a Pipelined Processor," 2021 IEEE Asian Test Symposium (ATS), Matsuyama, Ehime, Japan, 2021, doi: 10.1109/ATS52891.2021.00025.

2022

1. N. I. Deligiannis "New Solutions for Generating Functional Sequences Maximizing the Sustained Switching Activity of Complex SoCs." 2022 IEEE European Test Symposium (ETS), PhD Forum, Barcelona, Spain, 2022, uri: http://hdl.handle.net/2117/369981.

2. N. I. Deligiannis, R. Cantoro, M. Sonza Reorda, M. Traiola and E. Valea, "Improving the Fault Resilience of Neural Network Applications Through Security Mechanisms", 2022 IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Baltimore, MD, USA, 2022, doi: 10.1109/DSN-S54099.2022.00017.

3. N. I. Deligiannis, T. Faller, J. E. Rodriguez Condia, R. Cantoro, B. Becker and M. Sonza Reorda, "Using Formal Methods to Support the Development of STLs for GPUs", 2022 IEEE Asian Test Symposium (ATS), Taichung City, Taiwan, 2022, doi: 10.1109/ATS56056.2022.00027.

2023

1. J. E. Rodriguez Condia, N.I Deligiannis, J. Sini, R. Cantoro, M. Sonza Reorda. "Functional Testing with STLs: A Step Towards Reliable RISC-V-based HPC Commodity Clusters", High Performance Computing. ISC High Performance 2023. Lecture Notes in Computer Science, vol 13999. Springer, doi: https://doi.org/10.1007/978-3-031-40843-4_33

2. N. I. Deligiannis, T. Faller, Z. Chenghan, R. Cantoro, B. Becker and M. Sonza Reorda, "Automating the Generation of Functional Stress Inducing Stimuli for Burn-In Testing", 2023 IEEE European Test Symposium (ETS), Venice, Italy, 2023, doi: 10.1109/ETS56758.2023.10174232.

3. T. Faller, N. I. Deligiannis, M. Schwörer, M. Sonza Reorda and B. Becker, "Constraint-Based Automatic SBST Generation for RISC-V Processor Fami-

lies," 2023 IEEE European Test Symposium (ETS), Venice, Italy, 2023, doi: 10.1109/ETS56758.2023.10174156.

4. J. Anders, P. Andreu, B. Becker, S. Becker, R. Cantoro, N. I. Deligiannis, N. Elhamawy, T. Faller, C. Hernandez, N. Mentens, M. N. Rizi, I. Polian, A. Sajadi, M. Sauer, D. Schwachhofer, M. Sonza Reorda, T. Stefanov, I. Tuzov, S. Wagner, N. Zidarič, "A Survey of Recent Developments in Testability, Safety and Security of RISC-V Processors", 2023 IEEE European Test Symposium (ETS), Venice, Italy, 2023, doi: 10.1109/ETS56758.2023.10174099.

5. N. I. Deligiannis, T. Faller, I. Guglielminetti, R. Cantoro, B. Becker and M. Sonza Reorda, "Automatic Identification of Functionally Untestable Cell-Aware Faults in Microprocessors", 2023 IEEE 32nd Asian Test Symposium (ATS), Beijing, China, 2023, doi: 10.1109/ATS59501.2023.10317988.

2024

1. N. I. Deligiannis, R. Cantoro, M. Sonza Reorda and S. E. D. Habib, "Evaluating the Reliability of Integer Multipliers With Respect to Permanent Faults", 2024 27th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Kielce, Poland, 2024, hdl: https://hdl.handle.net/11583/2986512.

2. M. Bartolomucci, N. I. Deligiannis, R. Cantoro and M. Sonza Reorda, "Fault Grading Techniques for Evaluating Software-Based Self-Test with Respect to Small Delay Defects" IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), Rennes, France, 2024.