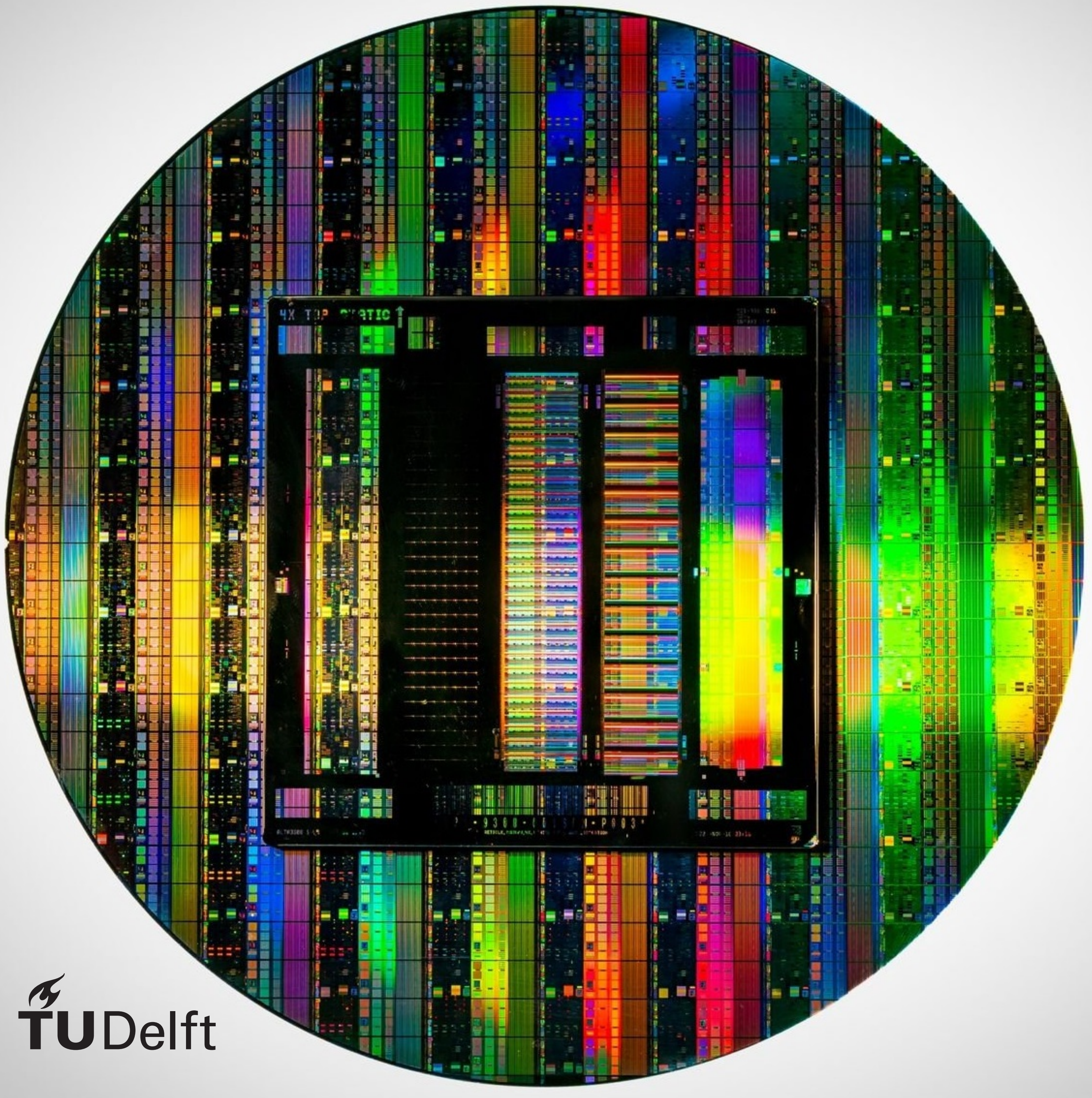
Complementing Software-Based Self-Test with DFT

P. Kremers



TU Delft

Complementing Software-Based Self-Test with DFT

by

P. Kremers

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday June 30, 2025 at 13:30.

Project duration:	November 11, 2024 – June 30, 2025	
Thesis committee:	Dr. ir. Moritz Fieback,	TU Delft, Daily Supervisor
	Dr. ir. Mottaqiallah Taouil,	TU Delft, Responsible Supervisor
	Prof. Dr. Koen Langendoen,	TU Delft

Cover: Wafer and DUV reticle mask by ASML

TU Delft

Abstract

Hyperscalers Meta and Google have observed a rare but severe phenomenon in cores throughout their processor fleets: Silent Data Errors (SDEs). Recent research efforts have indicated that marginal timing failures are the main cause of these SDEs. Currently, the underlying defects causing these failures, systematically escape production testing like full scan tests. Moreover, ageing-related failures that develop over the lifetime of a chip can also cause SDEs. To ensure reliability and to detect marginal defects, in-field testing is crucial. A promising approach for in-field testing is Software-Based Self-Test (SBST). This allows for online testing of a core by running a software program that activates faults and observes test responses. These programs are based on structural test patterns created by constraining ATPG (CATPG) to functionally possible inputs. The resulting test patterns are then converted into instructions, and faults are propagated into data memory, to be observed. Therefore, SBST programs enable testing a processor core while it is in functional mode. Recent SBST works have shifted from the stuck-at fault (SAF) model to the transition delay fault (TDF) model. Static fault models, such as the SAF model, fall short in modelling the marginal timing failures that cause SDEs. However, SBST programs targeting TDFs provide less coverage than the structural production test approach full scan. This thesis proposes a methodology to increase fault coverage (FC) of SBST programs, targeting TDFs, with a low area Design-for-Test (DFT) hardware addition to a core.

Increasing the FC of structural tests can be achieved by improving the testability of a circuit. Improving the testability of a circuit requires enhancing either its observability or its controllability. The focus of this thesis is on increasing the observability of a core through the addition of DFT hardware. To add the DFT hardware to a core in a way that complements SBST, an SBST generation framework, an SBST simulation framework, and three DFT designs were developed. The frameworks use the proprietary ATPG tool Tessent by Siemens. The SBST generation framework is based on partial scan CATPG test patterns. A design space exploration, to justify the DFT additions, is done by tweaking the partial scan configuration used by the constrained ATPG. The SBST program and the DFT are tested by the simulation framework, which converts the software program into a test pattern file. This test pattern file is then simulated in Tessent, providing FC results. The exploration of the design space follows a methodology that is based on the capabilities of the ATPG tool. The SBST program FC is increased by enhancing the observability of a selection of flip-flops using DFT.

Two DFT designs are implemented. One in the control and status registers (CSR) module, and another in the instruction decode (ID) stage module. Results are provided for both DFT designs. The DFT in the CSR module achieved a FC increase of 6.29 percentage points for SAF and 2.40 percentage points for TDF, at a cost of 0.84% area overhead. Furthermore, the DFT in the ID stage module achieved a FC increase of 1.01 percentage points for SAF and 1.92 percentage points for TDF, for a 0.65% area overhead. It is also observed that increased observability is more essential for detection of TDFs than the detection of SAFs. The SAF coverage is higher than some other works, but the SBST program size is significantly larger. When comparing the TDF coverage to other works it is clear that the FC results are significantly lower. However, when comparing the DFT area overhead to other works it is shown that the DFT additions introduce relatively little area overhead. In conclusion, this thesis has shown that DFT hardware that complements SBST can increase SBST program FC for a small area overhead. This indicates that DFT could aid SBST programs in matching the FC achieved by full scan. That would make SBST complemented by DFT an alternative to full scan, with a lower area overhead and flexible in-field testing capabilities.

To improve the results of this work, future work should be done to improve the baseline SBST TDF FC. This could be done by introducing a feedback loop from fault simulation to the constrained ATPG. This would allow proving that the same FC increase for area overhead trade-off can be made with a baseline SBST program that has higher FC. Furthermore, no efforts were done to minimize SBST program size. So, there is improvement that can be done in this area. The main limitations of this work are the justification step in the SBST pattern conversion and the functional constraint extraction process, resulting in lower fault coverage than that of state-of-the-art SBST works.

Acknowledments

Writing this thesis has been an exciting and rewarding challenge. It was by no means an easy task, which is why I have learned so much throughout the process. During this period, I have been fortunate to receive a great deal of support, and I would like to take a moment to thank those who have stood by me along the way.

First and foremost, I would like to thank my daily supervisor, Moritz Fieback. I truly appreciated our weekly meetings, during which he never rushed me and often took even more time than scheduled to help me. Moritz guided me in developing critical thinking skills, and whenever I felt stuck, he would suggest plenty different approaches I could try. I was always welcome to drop by his office, which made a big difference. I also wish to thank my responsible supervisor, Mottaqiallah Taouil, for his valuable feedback and thought-provoking questions.

I am also grateful to my friends on the tenth floor, who I worked alongside for 30 weeks. Our mutual venting about thesis frustrations, coffee breaks, and lunches kept me in good spirits. Additionally, I want to thank my friends outside the university for their constant encouragement and support in everything I pursue.

Finally, I would like to express my thanks to my family and my girlfriend. Their unwavering support and belief in me throughout both of my degrees has been instrumental in helping me reach this milestone.

Thank you all!

Pepijn Kremers
Delft, June 2025

Contents

List of Figures

List of Tables

Glossary

ALU	Arithmetic Logic Unit.
ASIC	Application Specific Integrated Circuit.
ATE	Automatic Test Equipment.
ATIG	Automatic Test Pattern Generation.
ATPG	Automation Test Pattern Generation.
BDD	Binary Decision Diagram.
BMC	Bounded Model Checking.
CATPG	Constrained Automation Test Pattern Generation.
CEE	Corrupt Execution Error.
CISC	Complex Instruction Set Computing.
CMOS	Complementary MOS.
CPU	Central Processing Unit.
CUT	Circuit-Under-Test.
DFT	Design-for-testability.
DSE	Design Space Exploration.
ECC	Error Correction Codes.
FC	Fault Coverage.
FCE	Functional Constraints Extraction.
FE	Fault Efficiency.
FSS	Fault-Sensitizing State.
GDF	Gate Delay Fault.
GPR	General Purpose Register.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
IC	Integrated circuit.
IPC	Instructions Per Cycle.
ISA	Instruction Set Architecture.
LoC	Launch-on-Capture.
LoS	Launch-on-Shift.
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor.
PDF	Path Delay Fault.
PPDF	Partial Path Delay Fault.
PUT	Processor-Under-Test.
RISC	Reduced Instruction Set Computing.
RL	Reinforcement Learning.

RTL	Register Transfer Level.
SAF	Stuck-at Fault.
SBST	Software-based Self-test.
SDC	Silent Data Corruption.
SDE	Silent Data Error.
SDF	Small Delay Fault.
SFI	Statistical Fault Injection .
SID	Selective Instruction Duplication.
SIMD	Single Instruction Multiple Data.
STIL	Standard Test Interface Language.
STL	Software Test Library.
TC	Test Coverage.
TDF	Transition Delay Fault.
VCM	Validity Checker Module.

1 Introduction

This chapter is an introduction to this thesis. The Introduction begins with Section 1.1, which explains the importance of and need for researching Software-based Self-test (SBST). This is followed by a discussion on the current state-of-the-art SBST including its limitations, in Section 1.2. Then, Section 1.3 covers the academic contributions of this thesis. Finally, in Section 1.4 the high level structure of the rest of the thesis is presented.

1.1. Motivation

In the information age chips can be found everywhere, i.e. in cars, smartphones, toys, household appliances, and even space. Due to the extremely large volume of integrated circuits (ICs) deployed and their wide range of applications, it is key that these ICs are manufactured correctly. For example, an IC in an electrical car that is manufactured incorrectly can result in a driver being unable to brake. A malfunctioning IC in a pacemaker can halt its service for a period of time and prove fatal. These examples are endless, especially in the context of industries like defence, health-care, automotive, and aerospace.

This pressing issue is tackled by the field of study on testing for IC quality and reliability. The root cause are imperfections in the physical structure of chips introduced during the manufacturing process. These are addressed by testing, which checks if the physical structure of the produced IC matches the intended physical structure [1]. During the testing process, the main objective is to detect manufacturing defects. The ratio of chips that pass this testing process is called the yield. The yield can greatly affect IC cost, as well as the time it takes to test each IC. Due to the large amounts of money involved in chip manufacturing, see Figure 1.1, developing new and improving existing testing strategies is of the utmost importance.

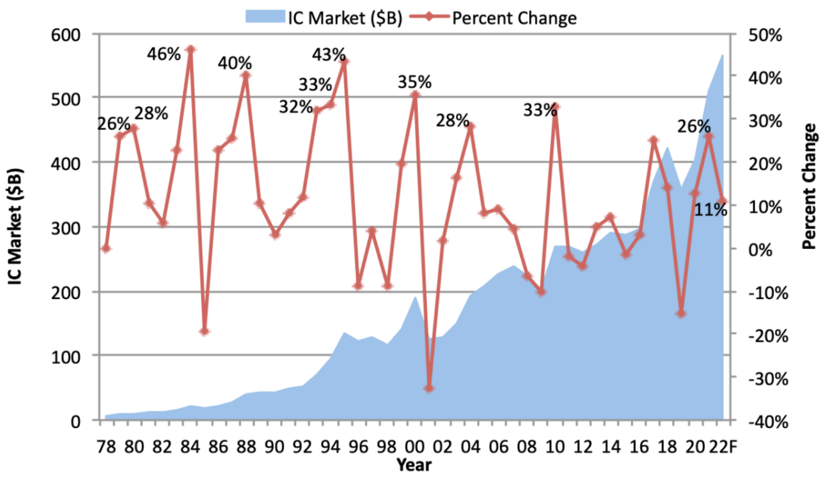


Figure 1.1: IC Sales Revenue from IEEE International Roadmap for Devices and Systems [2]

Despite decades of test development, currently deployed ICs, across device generations [3], have been observed to systematically suffer from errors that corrupt data and stay undetected [4]. This is extremely problematic, which is why there have been calls-for-help [4][5] by two industry hyperscalers: Google and Meta. These errors are bypassing current testing practices, and there are no effective in-field detection schemes. This has renewed enthusiasm in the IC testing community, and given a new perspective to a decades old issue: Silent Data Errors (SDEs). Due to SDEs corrupting data and staying undetected they are also known as Silent Data Corruptions (SDCs). Previously, the main hypothesis on what caused SDEs was that they manifest in memory due to transient soft errors caused by cosmic rays [6]. However, the previously mentioned hyperscalers, have a shared perspective on a newly identified potential cause of SDEs: manufacturing defects. SDEs are an extremely rare phenomenon, this is a reason why not much is known about them and why researching them is challenging. The scale of the operations of hyperscalers, more than a million of CPU cores per fleet [7], provides them with an unique opportunity to observe and study SDEs. They found a pattern of persistent SDEs that was traced back to the compute parts of cores. Meta states that debugging an SDE can take months of debug engineering time [5]. Their silent nature also allows SDEs to go unnoticed for a long time period. This corrupted data can do serious damage that is hard or impossible to reverse, especially when it is related to i.e. bank transitions, security feeds, legal logs, and decryption keys. That is why more thorough manufacturing tests need to be developed, as well as effective in-field detection schemes.

At their data centres Meta has observed that the manifestation of SDEs can depend on specific environmental conditions like temperature, frequency, and voltage [5]. Additionally, they found dependence on specific sequences of instructions, and even specific data values used in certain operations. What they described are intermittent faults, which are non-permanent faults that only appear under specific conditions. Intermittent faults can be caused by manufacturing defects as well as IC degradation due to ageing. Failures caused by degradation due to ageing are not present immediately after manufacturing, so they are not detected by post-manufacturing testing. The period of time an IC is specified to work correctly is called its lifetime, a metric used for this is reliability. Failures negatively impacting IC reliability can only be detected by applying in-field testing to deployed ICs. Meta calls them post-burnin failures and explains the increase of them by pointing to the decrease in feature size: *“Silicon feature sizes are now measured in nanometers, leaving smaller margins for error, and perhaps more risk of post-burnin (latent) failures.“* [5].

With the importance of testing, both immediately after manufacturing and in-field, established the right context is set to introduce Software-Based Self-Test (SBST). SBST is a method of testing a core in-field for defects, while it is in functional mode. It is called software-based because an SBST program is a sequence of instructions that can be run like a software program. An advantage of SBST as opposed to using scan design [1] for testing, is that the core can stay in functional mode and run the test program at-speed. Additionally, SBST programs can be split up in parts, allowing very small parts of a program to be run independently when the core has a small amount of idle time. This fits really well within the context of large data centres, which is where most of the SDEs are occurring. Additionally, most modern SBST methods are generated based on scan test patterns [8]. Scan tests are especially effective at detecting timing failures [9]. A prominent researcher in the test community, Adit Singh, has emphasized the need for *“new low-cost scan tests“* [10]. SBST can be seen as an answer to his call, as it has low overhead and is based on scan test patterns. This thesis includes a literature review of SDEs and its potential remedies. From this literature review SBST has emerged as a promising practical candidate solution.

The perspective that SBST is a promising test approach is shared by the test community, this can be seen alone by the vast amount of research on SBST in the last half decade [8][11][12][13][14][15][16][17][18][19][20]. However, current SBST solutions often still require a large amount of manual test engineering effort [14]. Even though this topic is still actively being researched, almost all works limit themselves to the restrictions set by the hardware of the circuit-under-test (CUT). Part of the challenge of developing an SBST program is ensuring the developed program is functional behaviour, so that there is no extra area overhead. Adding extra hardware that can be used during testing could aid in test development and execution. Furthermore, there have been very few to no recent works that attempt to enhance SBST by adding a form of DFT hardware. This work explores a methodology to enhance SBST by adding DFT hardware to a core. This is achieved by uniquely basing the SBST on partial scan patterns, instead of full scan patterns which is the standard.

1.2. State-of-the-Art

The topic SBST has many unsolved questions, thus there are many different works with methodologies focusing on different aspects. Section 1.2.1 introduces these methods and describes their main differences. This is followed by Section 1.2.2 which covers the works that have combined SBST with a DFT hardware addition. Finally, Section 1.2.3 concludes with limitations of the current state-of-the-art.

1.2.1. SBST Methods

Before introducing the various SBST methodologies, it is important to outline the main steps of SBST generation. The first step, Functional Constraints Extraction (FCE), entails extracting functional constraints from the CUT. Then, these constraints can be used to perform Constrained Automatic Test Pattern Generation (CATPG) on the CUT, which is the second step. The third step applies some form of postprocessing to the generated test patterns converting them into instructions and creating the final product: an SBST program. This third step is often called pattern-to-program conversion.

Next, a brief overview of state-of-the-art works with their focues on the first and third steps of SBST is provided. This work is based on structural testing, however functional SBST methods will be discussed in Chapter 4. The aforementioned chapter will also include more detail on the functional SBST methods mentioned below.

Functional Constraints Extraction

FCE is the process of extracting constraints from the processor, which can be applied during CATPG. To perform FCE manually, a great amount of manual effort and processor core knowledge is necessary. However, automating FCE is also really challenging to implement and not many works have done this. Furthermore, FCE is in practice always done partly by extracting constraints of some signals. In theory, it could be done completely covering every functionally possible state the core can be in. Perfectly complete FCE would ease the CATPG efforts, as for every test pattern a matching instruction sequence would exist.

In 2010, Zhang et al. [21] proposed an automatic FCE methodology. The methodology is based on expanded instructions, which are instructions with some extra information about the state of the rest of the circuit. Data mining through simulations is applied to map expanded instructions to signal values. From these mapping constraints follow which can be used in CATPG. The work targeted the fault model SAFs, more on this in Section 2.4, and their main contribution was an automated FCE method.

In 2013 Zhang et al. [22] developed a more advanced automatic FCE method. This method is based on executing-trace-based constraint extraction. This means that executing-traces from simulations are analyzed to find mappings between instructions and signals. Compared to the previous work from Zhang et al., this FCE method could additionally extract constraints for hidden control logic like branch predictors. So, this advance FCE method resulted in better results, even comparable to full-scan [1].

Pattern-to-Program Conversion

Pattern-to-program conversion is the core step in SBST program generation. It turns the test patterns generated by CATPG into a sequence of instructions that can run on a core like a program. The constraints used in this process are extracted by FCE. A high-level overview of some state-of-the-art works focusing in pattern-to-program conversion is provided below.

In 2016 Riefert et al. [23] propose the Validity Checker Module (VCM). This module is used to impose functional constraints on the CUT. They also model the problem of trying to reach a specific circuit state (specified in the test patterns), as a bounded model checking (BMC) problem. That way a BMC solver can be used to generate instruction sequences.

In 2021 Chen et al. [8] developed an SBST generation method based on a test program template. This template consists of instruction sequences that set different parts of the pipeline of the CUT into the desired state. This greatly simplified the conversion of full-scan patterns into a program of instructions.

Then, in 2023 Chen et al. [11] expand the test program template based SBST generation method. In contrast to modeling the justification problem as a BMC [23] problem, they model it as a binary decision diagram (BDD). Additionally, they develop a handcrafted test program for the register files.

In 2024 the test program template method is once again expanded, but by Kuo and Huang [12]. They propose a branch-aware SBST generation method. Essentially they rearrange the final SBST program to ensure faults during branch instructions are propagated to an observable output.

Lastly, it is interesting to note that in 2023 Anghel et al. [14] were the first and only work to generate an SBST program that targets path delay faults (PDFs). More on this fault model can be found in Section 2.4. They use a similar setup as [23], including a VCM and a BMC solver.

1.2.2. SBST Complemented by a DFT

The SBST works discussed thus far have all adhered to functional constraints imposed by the CUT and the instruction set architecture (ISA). Theoretically these constraints could be bypassed or adjusted by adding DFT hardware to the CUT. This would entail introducing area overhead, and negating one of SBSTs selling points: zero area overhead. However, the main appeal of SBST is that it can be performed at-speed and has high flexibility, making it ideal for in-field testing. Relatively little research has been done on the trade-off of area versus SBST performance or SBST development effort. A short introduction of the works that have combined SBST with some type of DFT hardware addition will follow.

Test Instructions

In 2001 Lai et al. [24] explored adding DFT hardware that enables SBST to use so called 'test instructions'. Their methodology for finding effective test instructions involves applying testability analysis to find registers with low controllability or observability. The *"testability analysis"* entails manually analyzing the micro-architecture of the core. Then, they add test instructions to increase the controllability and observability of those registers. They found that they were able to decrease SBST test time, while increasing fault coverage. At the cost of relatively little area increase. They verified their approach on the PARWAN core [25], and the DLX core. Their work targets *"path delay faults"*, but they do not define them and only mention them once.

Added Observation Points

In 2006 Nakazato et al. [26] proposed a DFT method to enhance SBST by adding observation points. However, they work under the assumption that each register in the core can be set to 0 or 1 by some function. The work by Nakazato et al. has some contradictions, it claims: *"A function to initialize the value of each register in the processor to '0' or '1' is added"*. But it also says: *"the proposed method adds only observation points to the original design"*. Additionally, there is no explanation on how these "functions" are implemented. Another thing missing from the work is any kind of nuance when choosing which observation points to add. Lastly, they do not mention for what fault model they are testing.

Functional DFT

Recently, in 2024 Irith Pomeranz published a work on a functional DFT [27]. This work is more theoretical than the previous two discussed works. It was not tested in combination with an SBST program, but SBST is the context in which the idea for the functional DFT can be used. Therefore, there are no area results, only fault coverage results. In short, the idea is to add multiplexers in front of specific flip-flops. These multiplexers swap state variables (flip-flop content) so that this swap propagates a fault towards an observation point in case of a faulty circuit, and in case of a fault-free circuit the state does not change.

1.2.3. Limitations

The focus of state-of-the-art research on SBST is mainly on two subproblems: pattern-to-program conversion and FCE. Improving pattern-to-program conversion will directly lead to a higher fault coverage (FC), because more of the patterns from the CATPG are converted into functional instruction sequences. Enhancing FCE capabilities will cause stricter constraints for the CATPG, leading to more patterns that can be turned into instructions, and indirectly also leading to higher FC results. So the main goal of research on SBST is improving its FC metric. However, none of the recent state-of-the-art works considers adding extra hardware to increase FC. Some faults could linger in a core, but have no easy path to propagate to an observable output. For example, when a bit in a status or control register is corrupted, propagating it to an observable output is a very convoluted process. In that case DFT hardware could help make it observable. Nonetheless, the recent state-of-the-art works limit themselves to the hardware already available on the core, even though for other forms of testing some area overhead is considered the norm [28].

The recent functional DFT idea of Irith Pomeranz is a promising way to improve FC, but has yet to be tested in combination with SBST. The solutions discussed that combine a DFT hardware addition with

SBST, are outdated and have some flaws. Firstly, Lai et al. [24] do not give any source or explanation for what they mean with path delay faults, and their work is done on small and outdated processor cores. Their method for finding poorly observable and controllable registers is ad hoc and cannot be automated. Nakazato et al. [26] have a good idea, but also leave some questions. They claim to only add observation points, but they also add functions that can set each register or work under the assumption that these exist. Additionally, there is no nuance when picking the registers that are made observable. Lastly, they provide no FC results, but express their results in *"template level fault efficiency"* for no explicit fault model.

The following list states the shortcomings of current state-of-the-art research on SBST

- No SBST technique has 100% fault coverage, this metric can be further improved.
- There is no analysis on how varying constraints effect FC and the pattern-to-program conversion process.
- There is no SBST program that tests all PDFs, only a selection of PDFs.
- There is no combination of DFT and SBST that targets the current state-of-the-art fault model TDF.
- The current works that combine DFT and SBST lack a systematic method of identifying effective locations for DFT addition.
- No DFT and SBST combination has been tested on a modern core.
- There is no work that analyses the trade-off between area and FC in the context of SBST combined with a DFT.

1.3. Contribution

This thesis shows that a low-cost DFT addition can enhance SBST program FC. To reach this goal a baseline SBST program is generated. Also, before landing at the solution SBST, a thorough literature study on the issue of SDEs was conducted.

The main contributions of this thesis are:

- A literature study on SDEs.
- A literature study on SBST
- A framework using Tessent that can run fault simulation of a sequence of instructions on the CV32E40P core.
- A framework that can automatically generate functional constraints for different partial scan configurations of the CV32E40P core.
- A partial scan based SBST generator, using Tessent, for the CV32E40P core that can automatically generate SBST programs for different fault models. Achieving 76.57% FC for SAF and 64.29% FC for TDFs.
- An analysis of CATPG with varying configurations.
- A methodology for finding locations for effective DFT additions that will enhance SBST.
- An implementation of a DFT addition for usage during SBST program execution. Achieving a a FC increase of 6.29 percentage points for SAFs and 2.4 percentage points for TDFs, at the cost of 0.84% area increase.
- An analysis of the trade-offs that should be considered when adding DFT for usage during SBST.

1.4. Outline

This thesis is structured as follows. Starting with Chapter 2, this will provide the relevant background information needed to understand the rest of the thesis. Then, chapter 3 covers the literature review, starting from the issue of SDEs, and ending with a resolution: SBST. After this, Chapter 4 will go in depth on SBST, the implementation developed for this thesis, and the experimental setup. One of the main contributions of this thesis, a methodology for choosing DFT avenues, is described in Chapter 5. Moreover, three DFT designs are discussed, and their results are presented. Lastly, the thesis will be concluded in Chapter 6.

2 Background

This chapter will cover the background theory necessary to understand the rest of the thesis. The chapter starts off by introducing the digital design flow in Section 2.1. Then, Section 2.2 discusses modern processor cores and any relevant related information. Section 2.3 illustrates the chip life cycle, from the conception of an idea to the end of life of the chips. As well as how testing fits into it. Following this is Section 2.4 in which fault models are discussed. Section 2.5 explains how these fault models are used to evaluate tests. Then, more detail will be provided on how tests are generated based on these fault models in combinational circuits in Section 2.6. Section 2.7 specifically discusses test generation for sequential circuits.

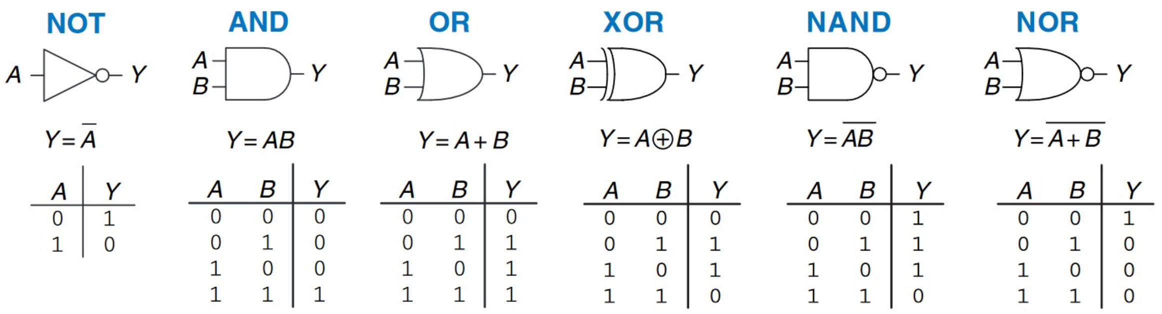
2.1. Digital Design

Digital design is a broad field covering the design of digital ICs. Digital ICs consist of digital circuits, while analogue ICs consist of analogue circuits, and mixed-signal ICs consist of both analogue and digital circuits. Digital circuits process digital signals, which are binary signals that can have 2 states (on/off). In contrast, analogue circuits work with analogue signals, which have a range of values they can take on. Digital systems consist of logic gates, made up of transistors, that carry out simple logic functions. These logic functions can be expressed in Boolean algebra. There are two main types of digital logic: combinational and sequential, these are discussed in Section 2.1.1. To design digital ICs, many steps need to be taken, see Section 2.1.3. First, the hardware needs to be designed at a high level of abstraction to be able to represent complex ideas, see Section 2.1.4. Then, this high level design needs to be translated into more detailed logic gates, see Section 2.1.5. After this, the design should be mapped onto the specific technology of the foundry that will manufacture it, see Section 2.1.5. More details on the final step in the digital design flow can be found in Section 2.1.6.

2.1.1. Logic Design

At the core of digital design one can find logic design. Logic design is the design of logic circuits with logic gates. Logic gates are defined as *"Logic gates are simple digital circuits that take one or more binary inputs and produce a binary output"* [29, p. 19]. These simple digital circuits can be represented by symbols, their inputs, and output. Logic gate functionality can be encoded in something called a truth table. The behaviour of logic gates can also be expressed in boolean algebra. An example of 6 simple logic gates can be seen in Figure 2.1. The aforementioned concepts are shown for NOT, AND, OR, XOR, NAND, and NOR gates. From top to bottom the figure shows for each gate: the symbol, the function (in boolean algebra), and the truth table.

Furthermore, logic design can be categorised into two categories: sequential logic design and combinational logic design. The two categories of logic design and their distinctions will be elaborated upon below.



A	Y
0	1
1	0

A	B	Y (AND)	Y (OR)	Y (XOR)	Y (NAND)	Y (NOR)
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

Figure 2.1: Basic Logic Gates including Truth Tables from [29]

Combinational Logic

A combinational circuit, consisting of combinational logic, is defined as *"A combinational circuit's outputs depend only on the current values of the inputs"* [29, p. 56]. An example of a combinational circuit is a logic gate. This means a combinational circuit has no memory, it can not recall the previous inputs it received. A combinational circuit merely carries out a specified logic function.

Even though combinational circuits do not use a clock, timing is an important aspect. After a change in input these circuits immediately change their outputs, or at least immediately after some propagation delay. In other words, combinational circuits are asynchronous. Propagation delay is the time it takes for a signal to travel through a path. The propagation delay of a circuit is the maximum time required for all changes in input signals to have propagated to the outputs of the circuit. A circuit also has a contamination delay, which is the minimum time it takes for a change in any input to change any output value. Figure 2.2 introduces two types of paths that every combinational circuit has, namely the critical path, and the short path. The critical path highlighted in blue goes through three logic gates, as can be seen its delay is the time the signal takes to propagate through these gates and the wires that connect them. It is the same for the short path highlighted in gray, but this path only contains one logic gate. These paths are very much related to the concepts propagation and contamination delay. The critical path of a circuit is the path from an input to an output with the largest propagation delay. From this it follows that the propagation delay of a circuit is the propagation delay of its critical path. The short path of a circuit is the path from an input to an output with the least delay. Therefore, the contamination delay of a circuit is equal to the propagation delay of its short path.

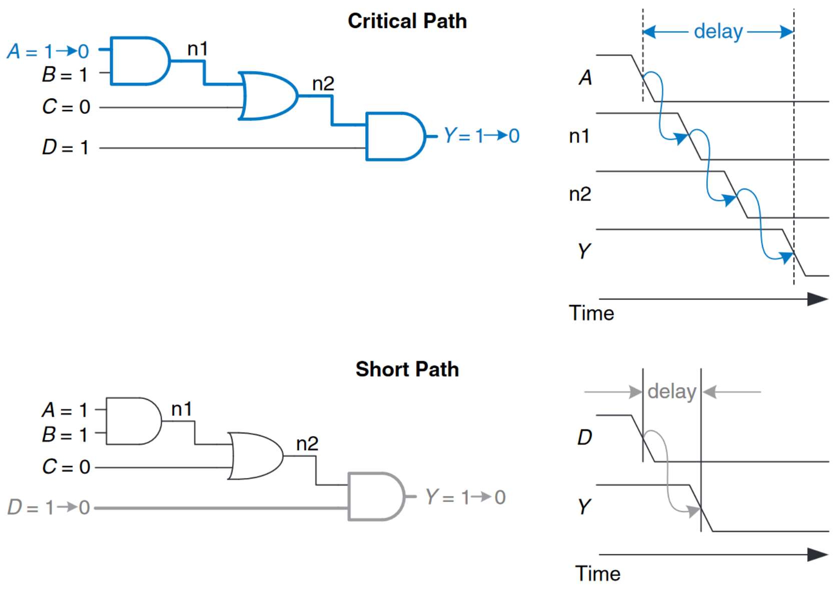
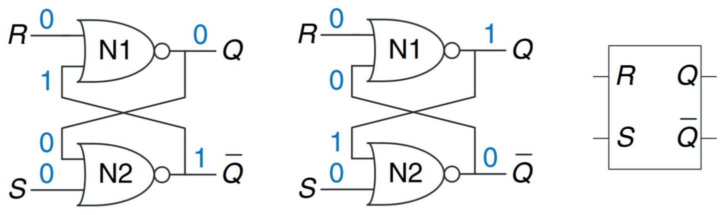


Figure 2.2: Critical and Short Path Examples from [29, p. 90]

Sequential Logic

Sequential logic is defined as follows *"The outputs of sequential logic depend on both current and prior input values"* [29, p. 109]. So, in contrast to combinational logic sequential logic does have memory. This implies that a sequential circuit can retain information about its past inputs. In practice, sequential circuits typically store a representation or transformation of prior inputs, commonly referred to as the state of the circuit. This state can be viewed as a set of bits called state variables [29], that contain information about the previous cycle of the circuit. The state or values of the state variables determine the behaviour of the circuit. The question that remains is: how does sequential logic have memory?

The essential component that enables building digital memory is a bistable element, an element with two stable states. Two simple and common bistable elements are latches and flip-flops. These are sequential circuits that store a state variable of one bit, and contain controllable inputs that can set this bit. There are many different types of latches, but to provide the necessary background for this thesis only the SR latch is briefly discussed. A simple example of a latch is an SR latch, and is shown in Figure 2.3. An SR latch consists of two cross-coupled NOR gates. Figure 2.3 shows the bistable states of an SR latch, its symbol, and its truth table. Case one indicates the inputs necessary to drive the latch to '0', case two drives the latch to '1', case three sets both outputs to '0' (creating an unstable state) and case four enables the latch to maintain state and store its current value. So the general idea of a latch, regardless of the type, is that it can store a bit and changes when its inputs change. This is asynchronous behaviour as a latch responds to changes on its inputs immediately.



Case	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 2.3: SR Latch: Bistable States, Symbol, and Truth Table [29]

In contrast to a latch, a flip-flop has synchronous behaviour and only changes its state value on the edge of a clock. The most common type of flip-flop is the D flip-flop. The design of a flip-flop is a bit more complicated than that of the previously discussed SR latch. In short, as shown in Figure 2.4 (a), a D flip-flop is made up of two D latches and an inverter that ensures the clock inputs of both D latches are complements of each other. A D latch consists of an SR latch, two AND gates, and an inverter. D latches have a clock input, and change continuously when the clock is high. The function of a D flip-flop is defined as *"a D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times"* [29, p. 114]. From this can be concluded that bistable elements that update their state in response to a clock edge are flip-flops, and all other bistable elements are latches.

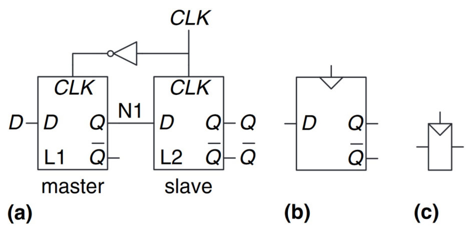


Figure 2.4: D Flip-Flop: (a) Schematic, (b) Symbol, and (c) Condensed Symbol [29, p. 114]

2.1.2. Transistors

Transistors are the fundamental building blocks of all digital ICs. All logic gates discussed in Section 2.1.1 can be built using transistors. A transistor is an electronic switch that can control the flow of current between two terminals based on the voltage applied to a third terminal. The most commonly used type in digital circuits is the MOSFET.

A MOSFET has three terminals: source, drain, and gate. The transistor is built on a silicon substrate, with regions doped to form either p-type or n-type material. Between the source and the drain lies a channel region, separated from the gate by a thin insulating layer of silicon dioxide. By applying a voltage to the gate, an electric field is generated that controls whether a conductive channel forms between the source and drain [30]. Figure 2.5 shows, in the context of a p-substrate, the two types of MOSFETs:

- **nMOS Transistor:** A positive gate voltage attracts electrons to the channel region beneath the gate oxide. If this positive voltage is large enough (above the threshold voltage), it creates an n-channel connecting the n-type source and drain regions, allowing electrons to flow from the source to the drain.
- **pMOS Transistor:** A negative gate voltage attracts holes to the channel region beneath the gate oxide. If this negative voltage is sufficiently large in magnitude (below the threshold voltage), it creates a p-channel connecting the p-type source and drain regions, allowing holes to flow from source to drain.

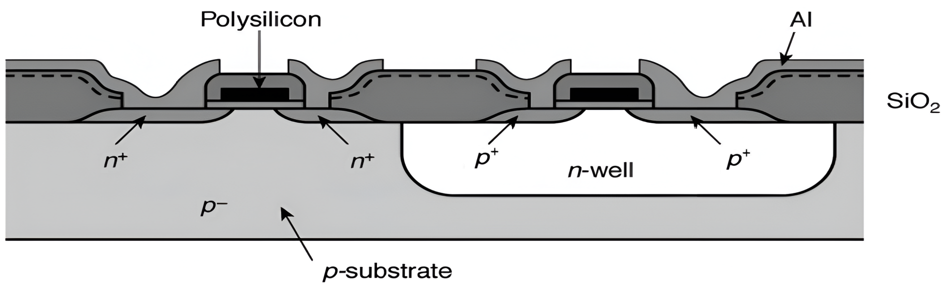


Figure 2.5: Cross-section of an N-well CMOS Process from [30, p. 42]

The ability to turn current flow on or off, which is logic switch behaviour, makes transistors essential for implementing logic gates, memory cells, and other digital structures. The small size and efficient switching of billions of transistors on a chip is what enables modern processors to operate at high speed and low power.

In CMOS (Complementary MOS) technology, both nMOS and pMOS transistors are used together to implement logic functions. This complementary structure minimises static power consumption and is the basis for all modern digital ICs.

2.1.3. ASIC Design Flow Steps

There are many steps in the design flow for application specific integrated circuits (ASIC), as shown in Figure 2.6. In this thesis only the front-end steps were applied. However, the back-end steps are still important context for the literature study and the thesis as a whole. Therefore, the following three sections discuss the large steps in the ASIC design flow. Namely: functional design, synthesis, and place and route.

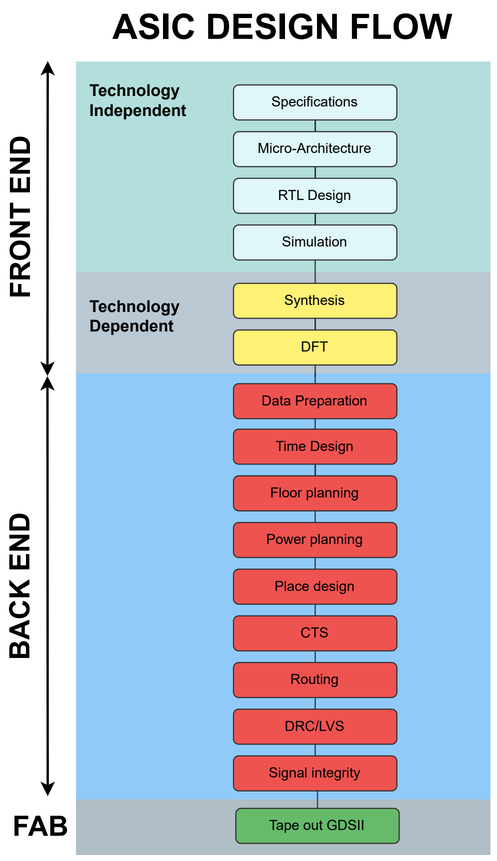


Figure 2.6: ASIC Design Flow inspired by [31]

2.1.4. Functional Design

Functional design is the step, in the ASIC design flow, with the most manual labour. This is where most of the creative designing takes place. Prior to this step it is crucial to form clear specifications for the design. These can then be used to guide the functional design. Functional design happens at register transfer level (RTL). This is an abstraction level that describes how data moves between registers in sequential circuits, and the logic that processes it in combinational circuits. This abstraction is necessary because it is not feasible to design complicated systems by describing logic gates, sequential elements, and wire connections. Designers can use the RTL abstraction to express the logical functionality of a design in a hardware description language (HDL), i.e. VHDL or Verilog.

When an RTL design is produced, the next step is simulation. The RTL design needs to be simulated to verify the correct functionality of the design. Ideally, the steps following functional design do not change the behaviour of the design. Additionally, the next steps require a lot of computation time. So, before proceeding one should be sure of the correctness of the design so far.

2.1.5. Synthesis

When a correct functional design is produced, the design can be synthesised. The synthesis step is where the design becomes technology dependent, this will be elaborated on below. Synthesis consists of three intermediate steps: Translation, Optimisation, and Mapping. They are briefly covered below:

1. **Translation**, is the process of converting a functional design to generic logic elements, i.e. logic gates and sequential elements. This step also flattens the design by removing all modular hierarchy. This effectively replaces all module instances with their internal logic.

2. **Optimization**, this process optimises the logic and structure of the design at a boolean or gate level to reduce area, power, and delay. It achieves this by doing things such as simplifying Boolean expressions, and removing redundant logic. All while not changing the circuit behaviour. This step also considers any design constraints that are provided, i.e. to ensure all paths meet the clock requirements.
3. **Mapping**, is when the design becomes technology dependent. The input of this step is an optimised netlist consisting of generic logic. This is converted into a netlist with real-world gates provided by a semiconductor foundry through a technology cell library. This is the first step that prepares the design for physical implementation on a chip. This steps also considers any design constraints that are provided, i.e. to ensure all paths meet the clock requirements. Furthermore, technology-dependent optimisation or post-mapping optimization happens after mapping has taken place.

Synthesis is concluded when all three steps are executed. The result of this process is an optimised netlist of the design expressed in logic gates provided by the foundry that will manufacture the chip.

2.1.6. Place and Route

After synthesis, there are many back-end operations that take place to convert the technology specific netlist into a GDSII file for production. A GDSII file is a binary file that contains the physical layout of an IC, this can be sent directly to the semiconductor foundry for manufacturing. The two main steps are placing and routing. Placing places the standard cells from the netlist onto the cell. Routing connects all components with wires according to the netlist. Moreover, there are optimisations taking place in this step such as the minimisation of wire delays. During place and route the setup and hold timing specified is also ensured. Setup time is the minimum time before the clock edge that the data input of a sequential element must be stable. Hold time is the minimum time after the clock edge that the data input must remain stable.

After placing, but before routing, a process called clock tree synthesis is carried out. This is the process of inserting and optimising the clock signal distributions network, so that every sequential element has a clock signal that meets certain requirements. These requirements include minimal clock skew, controlled latency, balanced loads, and low power consumption. Lastly, it is important to note that most DFT hardware will be inserted right before this step. The technology specific netlist is modified to accommodate DFT after synthesis, as this is significantly harder to do post place and route.

2.2. Processor Cores

The processor core is a concept that needs to be well understood to understand the entirety of this thesis. Because this thesis presents a case study on a specific processor core. The term processor is often used interchangeably with the term CPU. A CPU is a general-purpose processor, which means it is designed to execute a wide range of applications. An example of a non general-purpose processor is a graphics processing unit (GPU). GPUs were originally developed to run graphics processing tasks, i.e. rendering videos, images, and animations. This has made GPUs really efficient at parallel processing, and thus useful for AI computing.

In the context of this thesis whenever a processor core is mentioned, this refers to a CPU core. Modern CPUs have multiple CPU cores enabling multi-threading, and small fast cache memory for immediate data access. Modern CPUs will often also have access to some larger slower memory. However, this section will zoom in on the CPU core, not the system around it. The following subsections will cover the instruction set architecture that defines its capabilities, the types of instructions and operations it performs, its key structural components, the flow of data and control signals, and finally, the use of pipelining to improve performance.

2.2.1. Instructions

To use a core's hardware a language called instructions can be used [32]. Instructions can be represented by an arbitrary amount of bits, but often it will be 32 or 64 bits. When instructions are formatted as binary digits, it is called machine code. A sequence of instructions is called a program. Instructions are split up into segments called fields. Each field has unique information needed to execute the instruction. Some examples of information that can be found in fields are the opcode (unique for each instruction), the operands, and addresses.

A sequence of these instructions is called a program [32], and can execute specific tasks. When a core runs a program it will keep track of which instruction it is at. This does not necessarily need to happen incrementally, it is also possible for a program to redirect the core to a specific instruction. There are different categories of instructions, and also individual instructions that do very specific operations. But the common instruction categories are:

- **Arithmetic:** Arithmetic instructions carry out arithmetic operations like subtraction and addition.
- **Data Transfer:** Instructions that fall under this category can be used to read or write data to memory.
- **Logical:** Logical instructions are used for execution of logic operations, i.e. and, or, and nor.
- **Conditional Branch:** This category consists of instructions that will take a branch, if a specific condition is satisfied.
- **Unconditional Branch:** This category consists of instructions that will take a branch unconditionally. This category consists of instructions that cause an unconditional branch, redirecting the core to a different instruction in the program rather than the one that sequentially follows.

2.2.2. Instruction Set Architecture

This subsection introduces the instruction set architecture (ISA), defined as *“An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.“* [32, p. 85]. Every CPU is an implementation of a specific ISA. Therefore, it understands the instructions described by that particular ISA. The ISA specifies how low-level software can control a CPU. This happens through a compiler that can compile software programs into machine code that consists of instructions compatible with a specific ISA.

Besides instructions, ISAs define other characteristics and functionalities of CPUs. One of the things that ISAs define are the registers in a CPU. This includes specifications like the purpose of specific registers, the number of registers, and the size of each register. Generally the size of register are 32 or 64 bits. But an ISA can also include special purpose registers like registers for SIMD instructions, which are usually 128, 256, or 512 bits. Another aspect that ISAs specify is the data types used for operations. Some examples of commonly used data types are 32-bit integers, 64-bit integers, floating point numbers, and fixed point numbers. ISAs also describe addressing modes, which define how instructions identify its operands. The operands of an instructions can be values stored in registers, memory, or even constants. Addressing modes provide flexibility in accessing data by supporting different methods like immediate, direct, indirect, or indexed.

There are two main categories of ISAs: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). CISC ISAs, such as x86, consist of a large number of complex instructions, often capable of performing multi-step operations. This increases the complexity in hardware to simplify software. In contrast, RISC ISAs, such as ARM and RISC-V, use a smaller, generalized set of simple instructions that often require more instructions to perform the same task, but with improved efficiency and performance due to streamlined hardware design.

The ISA used in the case study in this thesis is RISC-V. It is a modern RISC architecture designed to be simple, modular, and extensible. It provides a minimal base instruction set with optional standard extensions (e.g., for integer multiplication, compressed instructions, or floating-point arithmetic), making it highly adaptable for a wide range of applications from embedded systems to high-performance computing. Its open-source nature and academic origins have contributed to its wide usage in both research and industry [33].

2.2.3. Main Modules

The main modules of every processor core are an ALU, control unit, decoder, and registers. Figure 2.7 shows an example of a very simple core. The ALU is the main engine of most cores, it can execute a wide range of logical and arithmetic computations. What an ALU executes depends on which instructions are read from memory, this is decided by the control unit through the program counter (PC). The control unit ensures that the right instructions are fetched from memory by controlling the PC. These instructions are fed into the decoder module which decodes the incoming instructions. When instructions are decoded, it becomes clear to the core what it should do. The control unit sends control signals to the rest of the core, telling it what it should do to carry out the decoded instructions. Lastly, registers are used to temporarily store data which can be used by the ALU for computation.

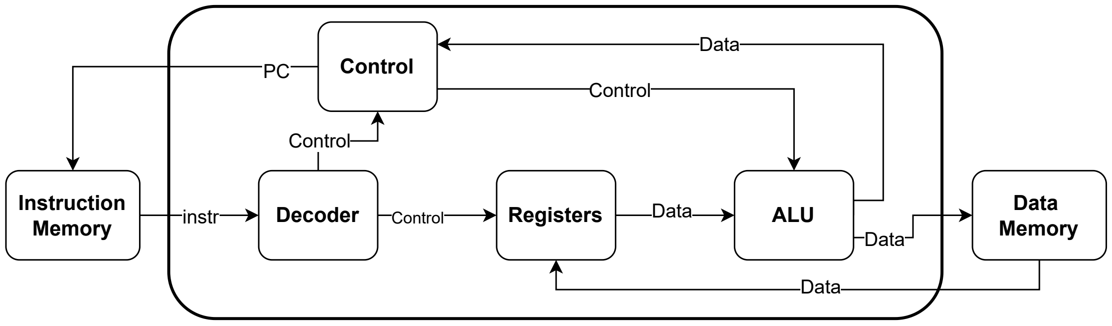


Figure 2.7: Simple Example Core

2.2.4. Data and Control Flow

In a processor core, executing an instruction requires the coordinated movement of data and control signals between the core's components. The data flow refers to how information, such as operand values, addresses, or results, moves through components like registers, the ALU, and memory interfaces. The control flow steers this movement by deciding the timing and coordination of operations via control signals.

Control signals determine things like whether data should be read from or written to registers, whether an ALU operation should be done, and where the results should be routed to. These control signals are often generated using some form of control unit based on the instruction opcode and status flags.

To manage data flow, the processor core uses data paths, the physical and logical paths through which data travels. A data path typically includes components such as multiplexers, registers, buses, and computation units like the ALU. For example, during an arithmetic instruction, the control unit enables specific registers to feed operands to the ALU, activates the ALU to perform an operation, and routes the result back to a destination register.

Timing is essential in ensuring correct operation. Most modern cores operate on a clocked cycle, meaning that data movement and processing are synchronised by a clock signal. On each clock pulse, depending on the control signals, data is fed into registers, and operations are triggered. This ensures that each step in the instruction cycle occurs in the correct sequence. These steps will be elaborated on below in Section 2.2.5.

The combination of data paths and control logic enables the CPU core to carry out complex instruction sequences in a structured and efficient manner.

2.2.5. Pipelining

To improve performance, processor cores universally use pipelining. Pipelining is defined as *"an implementation technique in which multiple instructions are overlapped in execution"* [32, p. 725]. This enables simultaneous processing of multiple instructions by dividing the execution path into distinct stages. Each stage is a step in the instruction cycle. A typical pipeline consists of the following stages listed below. Although it is important to note that these are the general stages, so some could be combined or split up even more.

- **Instruction Fetch (IF):** Retrieves the next instruction from memory.
- **Instruction Decode (ID):** Decodes an instruction and identifies the required operations and operands.
- **Execute (EX):** Performs arithmetic or logic operations via the ALU or other computation units like multiplier and dividers.
- **Memory Access (MEM):** Accesses memory if required.
- **Write-Back (WB):** Writes results to registers.

With pipelining, while one instruction is being decoded, another can be fetched, and a third can be executing etc. For a visualisation of this see Figure 2.8, this shows for three consecutive instructions in what stage each instruction is during what cycle. This overlap significantly increases throughput measured in instructions per cycle (IPC).

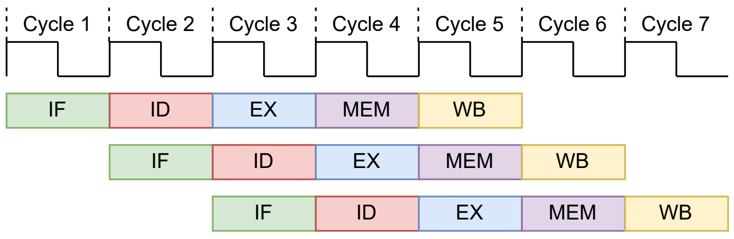


Figure 2.8: Example Pipeline Operation

However, pipelining introduces hazards that can disrupt smooth execution. Data hazards occur when instructions depend on the results of past instructions, but those past instructions are still in the pipeline. Control hazards arise from branches and jumps that may alter the instruction flow. Structural hazards happen when a required hardware resource is busy. To mitigate these hazards, several strategies are employed. Forwarding (bypassing) allows intermediate results to be used before they are written back. Figure 2.9 shows how forwarding is used to solve a data hazard. Stalling pauses pipeline stages until dependencies are resolved.

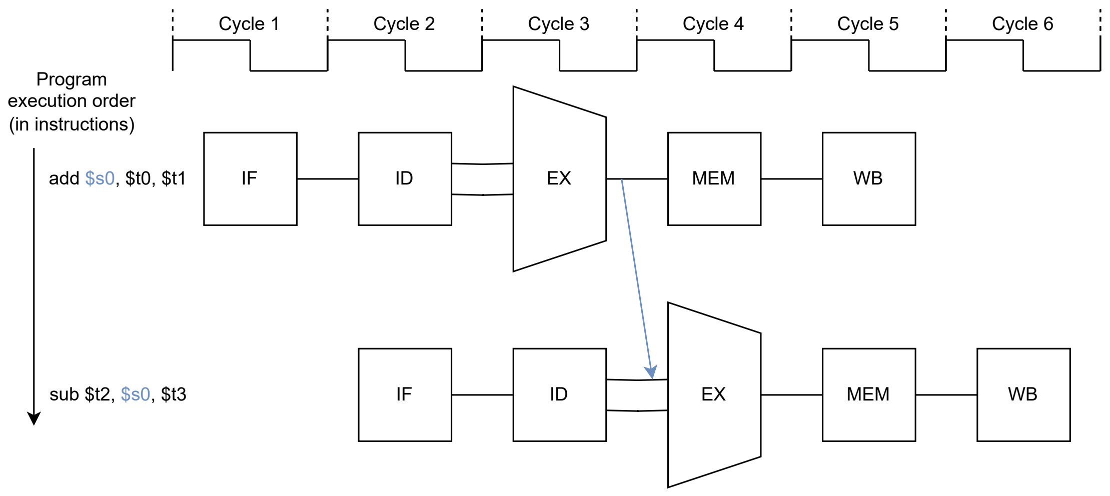


Figure 2.9: Example of Forwarding

Figure 2.10 shows an example where both forwarding and stalling are necessary to solve a data hazard. Lastly, there is also branch prediction which attempts to guess the outcome of branches to minimize delays caused by control hazards.

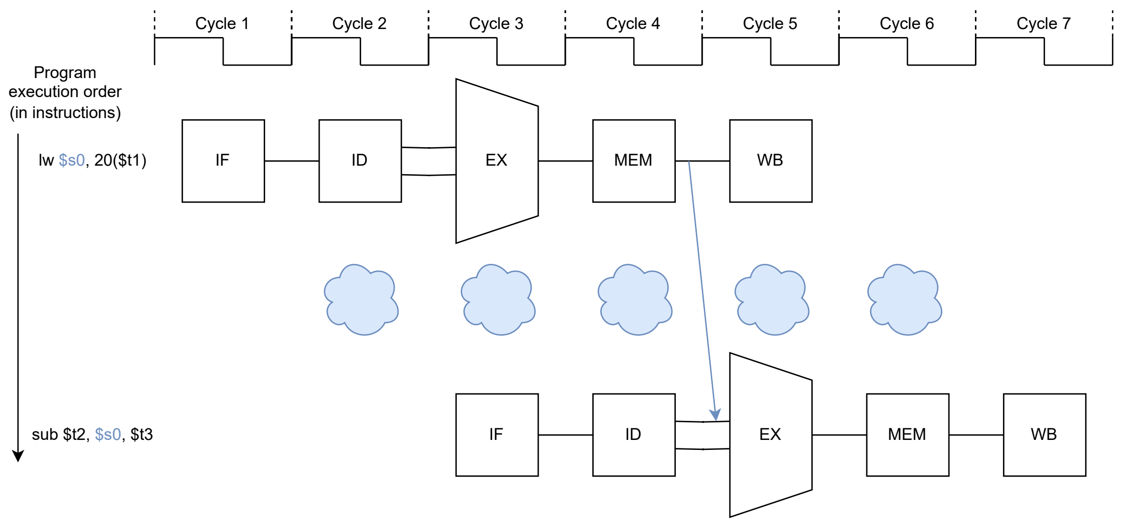


Figure 2.10: Example of Stalling

2.3. Chip Life Cycle

This thesis researches the field of testing. Therefore, it is crucial to explain where and how testing fits into the chip life cycle. The life cycle of a chip is shown in Figure 2.11. The operational life of a chip starts from the moment it is deployed until it cannot be used reliably any longer, this is called the wearout stage. However, the lifetime of a chip starts from the moment of manufacturing. Before manufacturing there is also the design process where the idea of the chip design is converted into a physical design ready for manufacturing. This section starts with defining chip quality and reliability. Then, the rest of this section will discuss how testing is incorporated into design, manufacturing, operational life, and wearout.

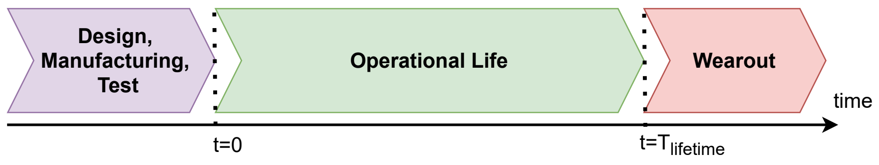


Figure 2.11: Chip Life Cycle

2.3.1. Quality and Reliability

Prior to discussing the chip life cycle and how testing is involved, it is key to define the two concepts quality and reliability. There is no formal definition of quality in [1], but the next best is *“The highest quality refers to the product meeting its requirements at lowest possible cost“* [1, p. 47]. This means that test quality determines if an IC can perform its function at the start of its operational life. The book includes cost in the indirect definition, because there is a trade-off between thoroughness of testing and test cost. This trade-off is included in the quality metric, so a good definition of quality would be *“Chip quality is the extent to which an IC meets its requirements at the beginning of its life, while minimising test cost“*. Gielen et al. define reliability as: *“Reliability is defined as the ability of a circuit to conform to its specifications over a specified period of time under specified conditions.“* [34]. So reliability ensures a chip functions according to its requirement during its entire (specified) lifetime.

2.3.2. Design

The design stage and process has already been partly discussed in Section 2.1.3. Therefore, this section will be brief. In this stage the desired functionality is converted into a physical design. During this process verification testing is done, this tests the functionality of the design not its structure. After the intermediate step synthesis a netlist with generic components is produced, this is often when test engineering takes place. This netlist can be used to generate test patterns, that can be applied after manufacturing to test the physical chip. Besides test pattern generation, DFT insertion for scan design also takes place after synthesis.

2.3.3. Manufacturing

From a finished design, a physical chip can be produced by a foundry, this process is called manufacturing. Manufacturing itself consists of two main processes: wafer processing, and assembly and packaging.

Wafer processing involves producing the ICs on a silicon wafer, through steps which build up or remove material in specific regions of the wafer, to create the transistors and interconnects that form the chip. A step called photolithography [35] is used to pattern specific regions on the wafer using light and a material called photoresist. This defines where material will be added or removed through an optical photomask [35]. Doping introduces impurities into the silicon to modify electrical properties and form the source, drain, and channel regions of transistors. Oxidation and deposition are used to create insulating or conductive layers, while etching removes unwanted material. This sequence of steps is repeated many times to create the multilayered structure of a modern IC.

After wafer processing, the wafer contains hundreds or thousands of identical chips. These chips on the wafer are tested with manufacturing tests. In the assembly and packaging stage, the wafer is diced into individual dies, and each die is mounted in a protective package. This package provides mechanical support and protection, and also contains the electrical connections that can connect the chip to a larger electronic system. Before the chips are shipped to customers, they once again undergo manufacturing tests to ensure correct functionality. This is a crucial step in quality control, as even a small defect in fabrication can lead to chip failure. Manufacturing tests include:

- **Probe Testing:** Is done while the dies are still on the wafer. Tests include basic electrical measurements and functional patterns, which as mentioned earlier are generated during the design phase.
- **Package Testing:** Performed after packaging to verify that the packaging process did not introduce faults and that the chip still functions correctly.
- **Burn-in Testing:** Exposing a chip to high temperature to accelerate the aging process, and detect aging-related failures. A burn-in duration of 50-150 hours at 125 degrees is effective in detecting 80-90% of production-induced defects [36].

Yield, the percentage of functional chips on a wafer, is a key metric in manufacturing. Manufacturing defects are often spatially distributed and caused by process variations. Higher yield translates to lower cost per working chip. However, testing also costs money and time: *“Device testing represents the single largest manufacturing expense in the semiconductor industry, costing over $40 billion a year.“* [36, p. 3]. These figures are probably low estimates, as this book was published more than 20 years ago in 2003. Figure 2.12 shows a trend that points to testing becoming more expensive than fabrication. This is due to the continuous decrease in feature size, which increases the number of transistors per chip.

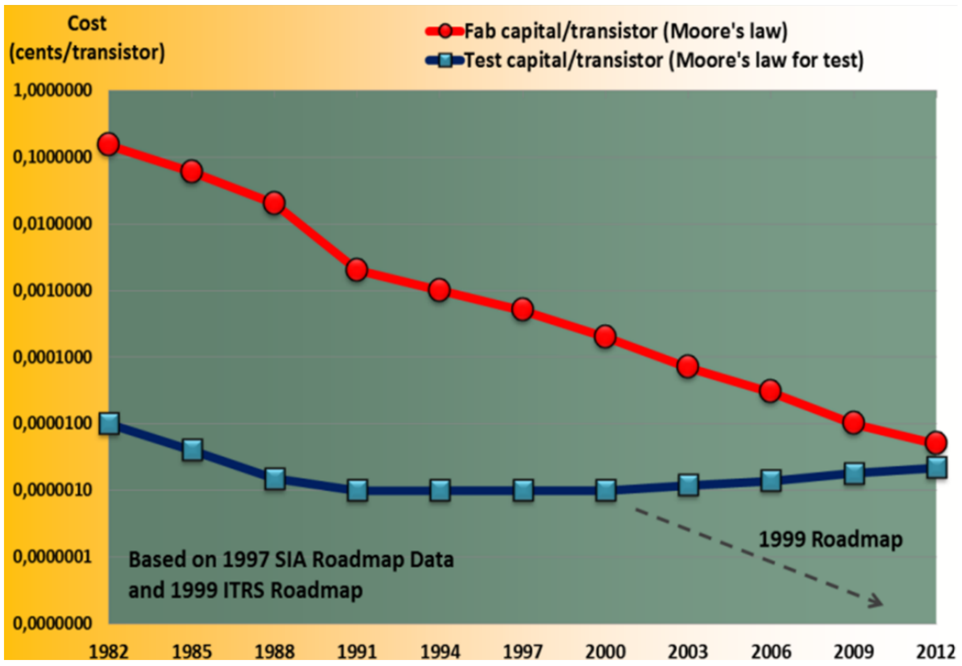


Figure 2.12: Fabrication and Test Cost per Transistor [37]

Overall, manufacturing translates the logical design into a physical object while testing acts as a filter that separates functioning devices from defective ones. The close interplay between manufacturing quality and test strategy directly impacts the cost, performance, and reliability of modern integrated circuits.

2.3.4. Post Manufacturing

Referring back to Figure 2.11, the post manufacturing period includes the chip's operational life and wearout. The first testing that happens post manufacturing is known as incoming inspection or acceptance testing [1]. This test is performed by the user or for the use by an independent testing house, to ensure product quality. Any test performed on a chip after this is called an in-field test. In-field tests can take on many forms, i.e by using scan design (see Section 2.7.1) or JTAG. In-field tests ensure the reliability of a chip.

Wearout

To explain wearout, the concept of failure mechanisms is important. Failure mechanisms are described as *"the physical and electrical causes for faults"* [36, p. 1]. The formal definition of faults will be discussed below in Section 2.4, but for now faults are an abstraction of defects. So, failure mechanisms can be seen as the physical and electrical causes of age defects. Figure 2.13 shows a classification of failure mechanisms. As can be seen failure mechanisms can be categorised in one of three categories. Electrical stress failures occur due to poor design or careless handling. Intrinsic failure mechanisms are inherent to the semiconductor die, i.e. dislocations, processing defects and crystal defects. These failure mechanisms are usually caused during wafer fabrication. Extrinsic failure mechanisms can be introduced during the packaging and interconnection processes.

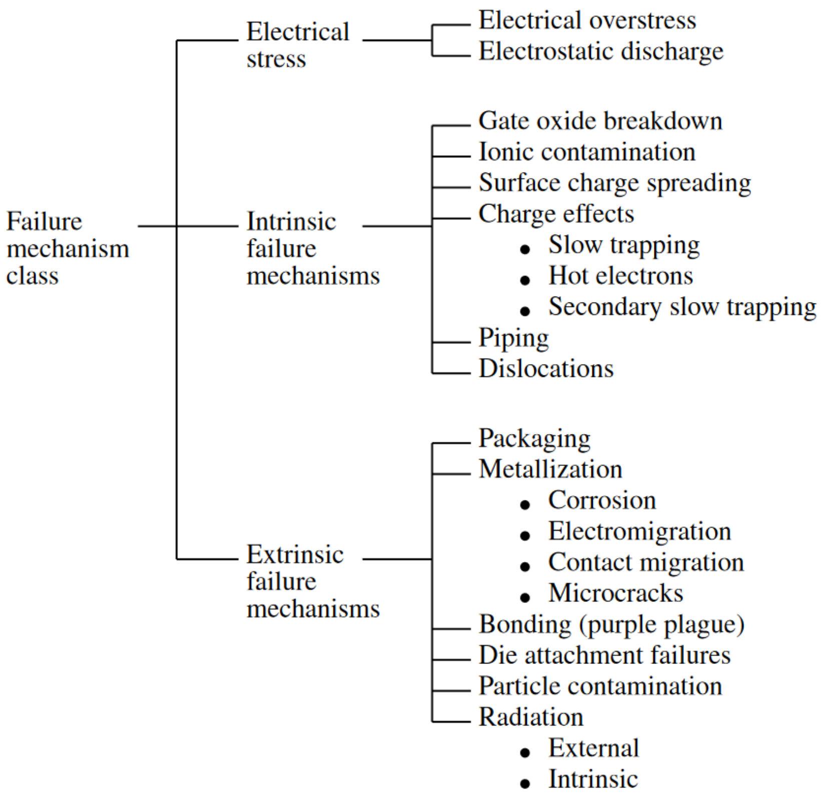


Figure 2.13: Classification of Failure Mechanisms from [36, p. 1]

2.4. Fault Modelling

The formal definitions of an error has already been discussed. Faults and defects however have only briefly been mentioned. A fault is defined as *"A representation of a "defect" at the abstracted function level is called a fault"* [1, p. 58]. An abstraction level often used is logic level. Specific faults adhere to their corresponding fault models. Which defects are represented by a fault depends on its fault model. The concept of fault models is necessary because the fault model aids in the development of structural tests (mentioned earlier in Section 2.4.1). Most test generation and test evaluation algorithms are built for specific fault models, this will be further discussed in Section 2.6.2 and Section 2.5. The following two sections will defects, fault modelling and fault types.

2.4.1. Defects

The definition of a defect is *"A defect in an electronic system is the unintended difference between the implemented hardware and its intended design"* [1, p. 58]. The goal of structural testing is to detect defects in an IC. In contrast, the goal of functional testing is to detect erroneous behaviour of an IC. Errors are defined as *A wrong output signal produced by a defective system is called an error. An error is an "effect" whose cause is some "defect."* [1, p. 58]. That being said, defects are a fundamental concept in testing. Hence, if an IC does not pass functional testing it will not pass structural testing. However, if an IC does not pass structural testing it can still pass functional testing. Defects can occur during manufacturing or during the use of devices. Some common defects in ICs are [1]:

- **Material Defects:** Caused during manufacturing due to issues with material, i.e. bulk defects (cracks, crystal imperfections), and surface impurities.
- **Process Defects:** Caused due to issues in the manufacturing process, i.e. missing contact windows, parasitic transistors, and oxide breakdown.
- **Package Defects:** Defects introduced during packaging, i.e. contact degradation, and seal leaks.

The materials and process defect can be related back to transistor physics discussed in Section 2.1.2. Surface impurities can take the form of unwanted particles or unintended dopants, and disrupt the doping regions (n+ and p+) seen in Figure 2.5, leading to leakage currents or threshold voltage shifts. These defects can interfere with the insulating silicon layer or junction integrity, degrading transistor performance or causing device failure. Parasitic transistors have unwanted current paths between adjacent n+ and p+ regions across the p-substrate or n-well, which can be introduced during manufacturing, leading to signal interference.

2.4.2. Fault Types

Faults manifest due to various failure mechanisms, which can differ in their occurrence patterns and persistence. Figure 2.14 show the different categories permanent, non-permanent, intermittent, and transient. Depending on the underlying failure mechanism, faults can either be permanent or non-permanent. Permanent faults exist for an indefinite period of time, while non-permanent faults are not constantly present but only part of the time. Non-permanent faults can be further split up into transient and intermittent faults. Transient (or environmental) faults occur once randomly and are caused by environmental conditions. Related failure mechanisms are radiation, noise, and power supply fluctuations. Realistically these faults cannot be tested for, because they occur randomly and are not reproducible. Intermittent (non-environmental) faults are caused by non-environmental conditions, so they manifest periodically and therefore are somewhat reproducible. Some examples of these non-environmental conditions are loose connections, timing issues due to aging components, and hazards in critical timing [1]. The key takeaway from intermittent faults is that they are reproducible, so they can be modelled by permanent fault models. Even though testing might have to be repeated several times until the intermittent fault is detected.

Current test practices are built around permanent faults. However, Chapter 1 pointed out that a lot of SDEs are most likely caused by non-permanent failure mechanisms. As transient faults cannot be systematically detected through testing, that leaves intermittent faults. So when considering the problem of SDEs testing for intermittent faults should be considered a solution with great potential.

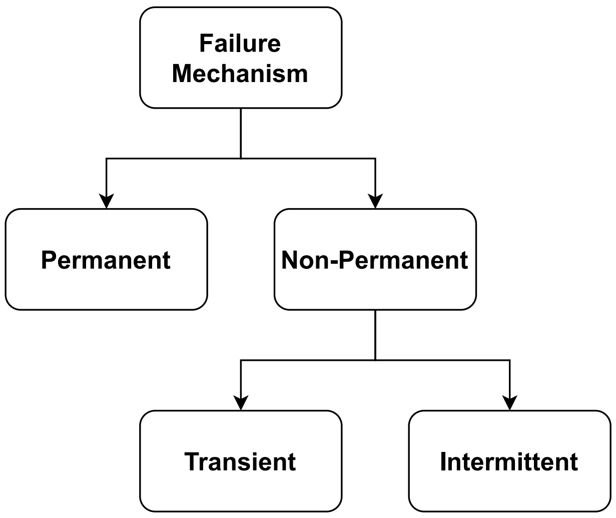


Figure 2.14: Failure Mechanism Classification

In engineering, modelling is often used to formulate a problem in a way that makes it easier to solve. In the case of fault modelling, test engineers deal with a problem that manifests in physical reality. Fault models create a mathematical abstraction of this physical reality. This enables the usage of analytical tools to solve the problem. These fault models exist at varying abstraction levels. The most used abstraction level for this is RTL level, but there are also fault models at transistor level and other lower levels.

Static Faults

The most used fault model is the single Stuck-at-fault (SAF) model. This model operates at the gate abstraction level. The single SAF model is defined in [1, p. 71] as:

"

Three properties characterize a single stuck-at fault:

1. *Only one line is faulty.*
2. *The faulty line is permanently set to either 0 or 1.*
3. *The fault can be at an input or output of a gate.*

"

Figure 2.15 shows an example of a single SAF. The line between the OR gate and most right AND gate is stuck at the value '1'. To test if this SAF is present in this circuit a test vector is applied. The correct behaviour of the circuit would output a '0', when the test vector would be applied at the circuit's inputs. However, if the aforementioned SAF is present the output will become '1' and the SAF can be detected. The multiple SAF model is similar to the single SAF model but it assumes two or more lines are faulty.

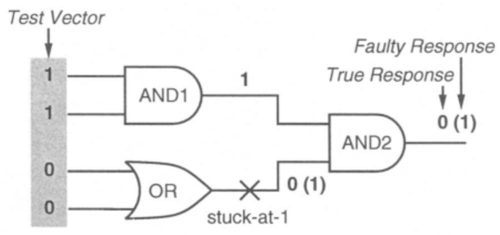


Figure 2.15: An Example of a Single SAF from [1, p. 71]

Another fault model at gate level is the bridging fault, shown in Figure 2.16. Bridging faults are a short circuit between two or more lines causing logic interference. A fault model at transistor level is the transistor fault, which can be stuck-open or stuck-short. A stuck-open transistor fails to conduct when it should, causing floating outputs. A stuck-short transistor conducts when it should not, causing shorts between the power source and the ground. Analogue faults are modelled at the analogue level, an example is parametric faults which are deviations in component values like resistance or threshold voltage. At a higher abstraction level like architecture behaviour faults can occur. An example of a behaviour fault is when conditional branch instructions are always taken.

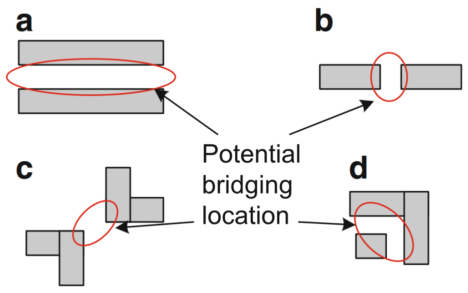


Figure 2.16: An Example of potential bridging faults from [9, p. 4]

Delay Faults

Delay fault models will play a key part in any solution to SDEs. This will be further elaborated on in Chapter 3 when low voltage testing is discussed. But at this point in the thesis it is already known that many SDEs are likely caused by intermittent failure mechanisms [5][4], which includes timing problems. This is already one reason delay fault models should play a part in any solution to SDEs.

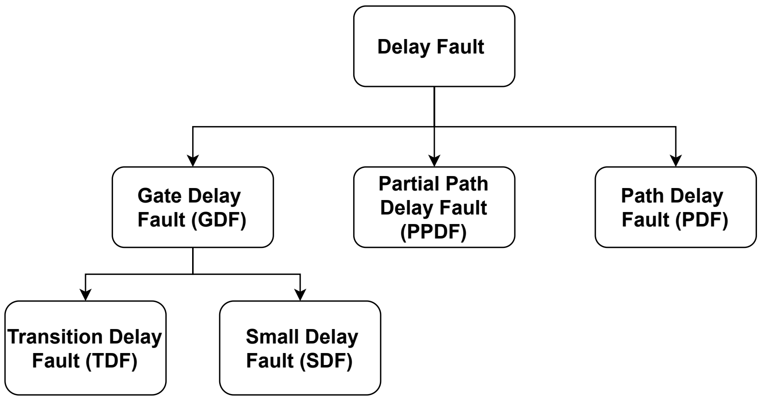


Figure 2.17: Delay Fault Model Taxonomy

Figure 2.17 shows the different delay fault models and their relations. The simplest delay fault model is the GDF model. The GDF model is defined as: *"A circuit is said to have a GDF in some gate if an input or output of the gate has a lumped delay fault manifested as a slow 0→1 or 1→0 transition"* [36, p. 38]. The most used GDF model is the TDF model [38], because in the context of delay fault models it is relatively easy to develop test for. An example of a TDF is shown in Figure 2.18. As can be seen in the timing diagram of the faulty circuit, the TDF is causing a delay big enough so the transition of D2 is delayed until after the clock period. The consequence of this is that the output D flip-flop will store and output the incorrect value. A SDF at the same line as the illustrated TDF, would also cause a delay in the transition of the value of D2. However, this would not be directly problematic because the transition would still take place between the end of the clock period.

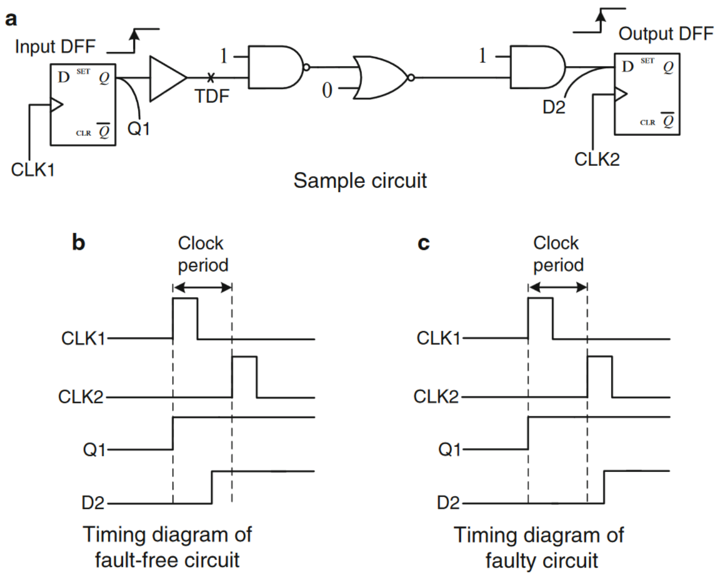


Figure 2.18: An Example of a TDF from [9, p. 5]

The PDF model is defined as *"It assumes that there is a cumulative delay defect along a combinational path, which causes the path to exceed some specified duration. This combinational path can begin at a primary input or a clocked flip-flop and can end at a primary output or a clocked flip-flop"* [9, pp. 5, 6]. This specified duration can be the clock period. From this definition a few insights related to GDFs can be made. PDF effectively models SDFs. When an accumulation of SDFs becomes problematic, creating a path delay that is larger than the clock period, it manifests as a PDF. Furthermore, a TDF can be detected by any PDF for which the path passed through the gate or line where the TDF is located. However, a PDF is not always detected by a TDF that lies on its path. For example, when the TDF is sensitized and propagated through another significantly shorter path.

Clearly the PDF model [39] is more thorough and will potentially be able to detect more defects than the TDF model. On the other hand, in a large design the number of paths can be huge and will result in extremely long PDF test development and execution times. As a compromise between the two there is the PPDF model. It is known as the segment delay fault model, but it has the same abbreviation as transition delay fault so it shall hereby be renamed to partial path delay fault PPDF. Because this fault model targets transitions on path segments of length L or smaller [36]. So, these segments contain L or fewer lines. L can be chosen based on the test requirements. Picking $L = 1$ would result in a TDF, and setting L to the maximum length of any path through the circuit would result in a PDF.

2.5. Test Evaluation

Test evaluation plays a key part in testing. It reports fault coverage that test patterns provide circuits, but it also plays a key role in test generation. ATPG uses test evaluation in a feedback loop to get feedback on the test patterns were generated. The main engine behind test evaluation is fault simulation. In essence it simulates the circuit under test and injects faults into it. Then, it compares good machine simulation [38] with fault injection simulation to determine if the injected fault is detected. Figure 2.19 shows how fault simulation is included in ATPG. As can be seen the fault list is updated based on what faults are detected by current test vectors according to the fault simulation. That way ATPG focuses on faults that are not yet detected by already generated test patterns.

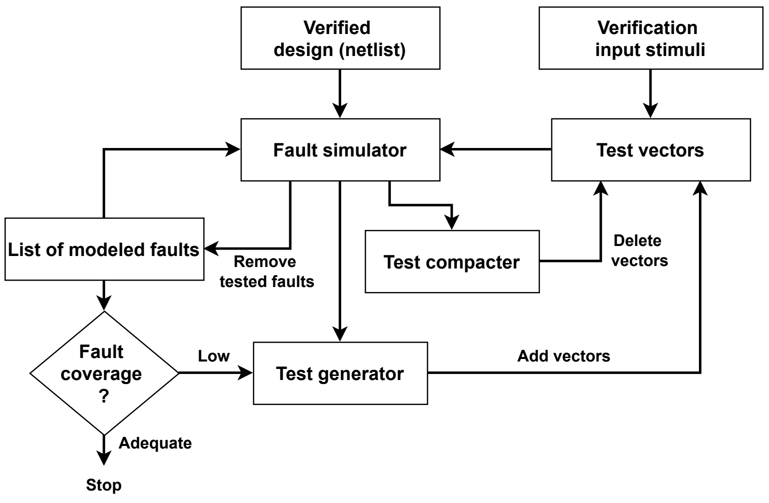


Figure 2.19: Fault Simulation in the Context of Test Generation based on [1, p. 88]

2.5.1. Fault Simulation

As mentioned above fault simulation checks what faults are detected by which test patterns. This is done by comparing simulation of a fault injected circuit with good machine simulation. Because this process is used very frequently during test development there are different methods with varying levels of efficiency. The two fault simulation methods most used in practice are:

- **Serial Simulation:** Simulates a single fault at a time, by running all test patterns with a single fault present. This is simple but can be time-consuming for a circuit with many faults.
- **Parallel Simulation:** Simulates multiple faults simultaneously. The idea is to *“use the bit-parallelism of logical operations in a digital computer“* [1, p. 107]. This enables simultaneous simulation of n circuits (for an n-bit word machine) with identical behaviour but different values.

2.5.2. Metrics

It should now be clear that fault simulation computes the results of test patterns based on some fault model. The relevant metrics for test will be covered in this section. First, the metrics that indicate how good a test is at detecting a certain fault model are fault coverage FC, and fault efficiency FE. As can be seen in Equation 2.1, FC is the ratio of how many of the total faults for a specific fault model are detected. Equation 2.2 shows that FC is similar to FE, but FE also takes into account undetectable faults. Therefore, FE is a better indicator of how the ATPG performs. Both FE and FC can be expressed as a fraction or a percentage.

$$Fault\ coverage = \frac{number\ of\ detected\ faults}{total\ number\ of\ faults} \tag{2.1}$$

$$Fault\ efficiency = \frac{number\ of\ detected\ faults}{total\ number\ of\ faults\ -\ number\ of\ undetectable\ faults} \tag{2.2}$$

These are very important performance criteria, but not the only criteria. Another key metric is test set size. The test set size is the total number test vectors in a test set. This can be seen as a cost, because the larger the test set size is the longer a test set will take to apply. Also, the test set size determines how much memory is needed to store the test set on automatic test equipment or even on chip. Another cost metric is CPU time. CPU time, expressed in any time unit, is the time it takes for the ATPG tool to generate the test set. Often CPU time is not a trivial metric because test development for a design only needs to be done once, and then the generated test set can be used on all chips with the design. However, when CPU time blows up and it takes weeks to run ATPG the metric can be a good indicator of the limits of the ATPG tool.

2.6. Combinational Test Generation

Combinational test generation is structural test generation for combinational circuits. Structural test generation is the process of creating structural test patterns to detect defects in an IC. It "is based on fault models, and focuses on detecting manufacturing defects in the circuit by considering how a defect manifests at the logic level" [9, p. 12]. In contrast, in functional testing the goal is to verify the expected behaviour of a design. Structural test generation for ICs is something that realistically can not be done manually, therefore the standard used is automatic test pattern generation ATPG. By abstracting physical defects into mathematical fault models, ATPG algorithms can systematically explore the circuit's behaviour to activate and propagate faults to the primary outputs. Combinational ATPG treats the circuit as a purely Boolean network, ignoring state elements. It relies on the deterministic nature of logic gates and the finite input and output space. The goal is to generate a set of test vectors that achieves high coverage of a specified fault list. This section will cover different aspects involved in combinational test generation.

2.6.1. Search Space Abstractions

ATPG tools use data structures to describe the search space for test patterns. Data structures used for this are binary decision diagrams and binary search trees. When a binary search tree is used to model a logic circuit there is a branch leaf for each input combination. In the worst case ATPG algorithms completely search this tree to prove untestability of a fault. This is undesirable because the exhaustive search space is 2^n input vectors (for n inputs). Therefore, these are pruned through logical abstractions and implications:

- **Justification:** Assign values to primary inputs to satisfy a desired internal signal value.
- **Implication (Forward/Backward):** Once an assignment is made, Boolean consequences are propagated forward (to update gate outputs) and backward (to constrain preceding gates), reducing choices at other inputs.
- **Backtracking:** Inconsistent assignments trigger backtracking to previous decision points, enabling search over a much smaller tree than 2^n.
- **Fault Dominance and Equivalence:** Faults that are detected by every test pattern for another fault can be removed from the fault list and assumed to be detected when the other fault is detected.

A further optimization is the use of ATPG algebra, which is *"a higher-order Boolean set notation with the purpose of representing both the "good" and the "failing" circuit (or machine) values simultaneously"* [1, p. 159]. In Roth's five-valued algebra (see Table 2.1), the symbol D means '1' in the good machine,

’0’ in the faulty machine, and $\overline{D}$ means ’0’ in the good, ’1’ in the faulty. The values ’0’ and ’1’ represent exactly that in both good and faulty machines, while X means “unknown in both,” so only one pass of ATPG is necessary to propagate differences between faulty and good machines. Muth extended this idea to a nine-valued algebra by splitting X into cases where one side is known and the other unknown ($G0$, $G1$, $F0$, $F1$), which can improve combinational-circuit testability when only one side’s value is observable.

Table 2.1: Roth’s Five-valued and Muth’s Nine-valued Algebras

Symbol	Meaning	GM	FM	Algebra
D	1/0	1	0	Roth’s Algebra [5 values]
$\overline{D}$	0/1	0	1	
0	0/0	0	0	
1	1/1	1	1	
X	X/X	X	X	
G0	0/X	0	X	Muth’s additions
G1	1/X	1	X	
F0	X/0	X	0	[Extended unknowns]
F1	X/1	X	1	

2.6.2. ATPG Algorithms

Different ATPG algorithms differ in how they traverse the search space. There are three main categories of ATPG algorithms, namely: exhaustive ATPG, random ATPG, and ATPG based on path sensitisation methods. Exhaustive ATPG tries all possible test vectors, therefore this is not a scalable solution. Random ATPG is based on generating random patterns. Because this approach does not analyse the circuit’s logic structure, it is limited in the fault coverage it can achieve. Path sensitisation methods apply path sensitisation techniques and are currently the standard ATPG method. It consists of the following three steps:

1. **Fault sensitisation:** also known as fault activation or excitation, forces a signal to create a difference between the good and the faulty circuit.
2. **Fault propagation:** also known as path sensitisation, ensures that the fault effect is propagated to at least one primary output of the circuit.
3. **Line justification:** Applies backwards justification to find primary input values that match the fault sensitisation and propagation signal assignments made in the previous two steps.

Now a brief explanation of the first path sensitisation algorithms, the D-algorithm, will follow. The D-algorithm uses Roth’s D notation ($1/0$ on a line) and iteratively applies sensitisation, propagation, and justification until a test is found or the fault is proven to be untestable. Figure 2.20 shows an example of the D-algorithm finding a pattern for a stuck-at 0. First a symbol from Roth’s algebra is assigned to the fault site, in this case a D (so ’1’ for the good machine and ’0’ for the faulty machine). Then, a path is selected to an output, and the off-path inputs are set accordingly (in green). Lastly, justification is applied to find matching primary input values resulting in a test vector. A test vector V consists of a value for each primary input present in the CUT.

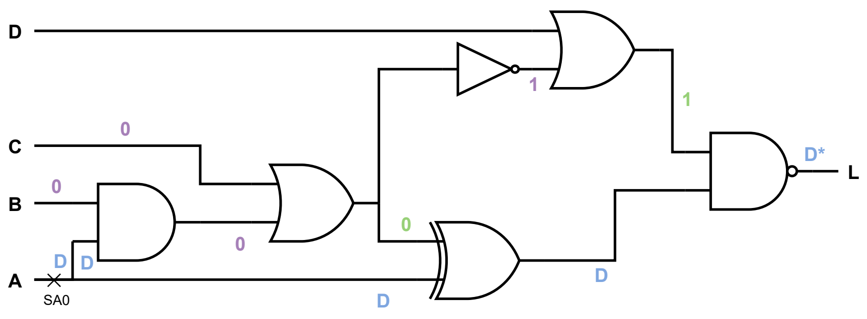


Figure 2.20: An Example of The D-algorithm

2.7. Sequential Test

Sequential tests are tests for circuits with sequential elements combined with combinational logic. In Section 2.4.2 delay faults were covered, delay tests test for delay faults. Delay test thus check if transitions (slow-to-rise and slow-to-fall) complete within the set time constraint. Delay faults can be observed at sequential elements, so they are often applied in combination with scan design. However, scan design is also useful for static faults in sequential circuits. In short, scan design enables setting and observing sequential elements in a circuit. This section first introduces scan design before discussing TDF, PDF, and SDF testing.

2.7.1. Scan Design

The essence of scan design is to make flip-flops controllable and observable, to improve FC. In a non-scan design the only observable and controllable nodes are the primary input and outputs. However, this imposes limitations on the amount of faults that can be detected. That is why scan DFT hardware is added. This is achieved by converting the flip-flops into scan flip-flops (SFFs). This involves adding a MUX, controlled by the scan enable signal (SE), in front of the flip-flops. When SE is ’0’ the MUX will ensure normal circuit behaviour, but when SE is ’1’ (this is called test mode) the MUX will feed the scan in signal to the flip-flop. As shown in Figure 2.21, these scan flip-flops are connected into one or more scan chains, that act as shift registers when the circuit is in test mode. This enables setting and observing the flip-flop values through shifting, via the additional primary input scan in (SI) and primary output scan out (SO). There are two types of scan design full scan and partial scan. Full scan is when all flip-flops turned into scan flip-flops, so sequential ATPG is reduced to combinational ATPG. Partial scan entails that a selection of flip-flops are turned into scan flip-flops, introducing trade-offs between area overhead and test difficulty.

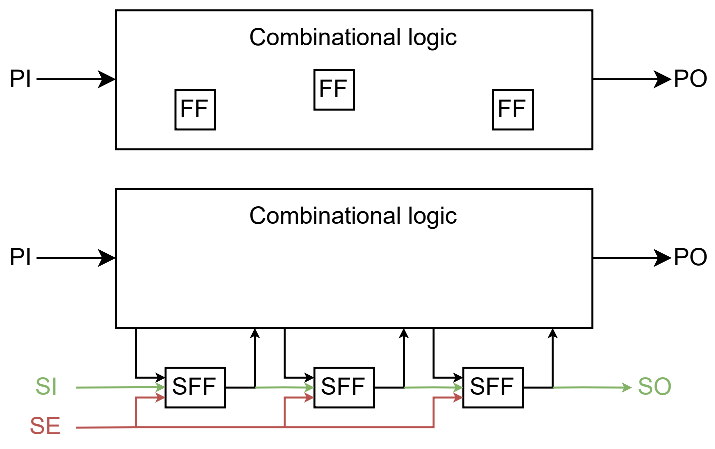


Figure 2.21: A Sequential Circuit Inserted with Scan Design

A waveform is shown in Figure 2.22, that displays the functionality of the scan design from Figure 2.21. It is can be seen that when the SE signal is high the bits a, b, and c are shifted into the scan flip-flops. Once the circuit is in the right state, SE goes to ‘0‘ again and the test pattern is applied through the primary inputs (PI). At the end the primary outputs (PO) and state variables d, e, and f are captured to check for any unexpected behaviour. Finally, the SE is high again so that bits d, e, and f can be shifted out and be made observable at output SO. At the same time the circuit can be set into the desirable state for the next test pattern, see bits c, b, and a being shifted in.

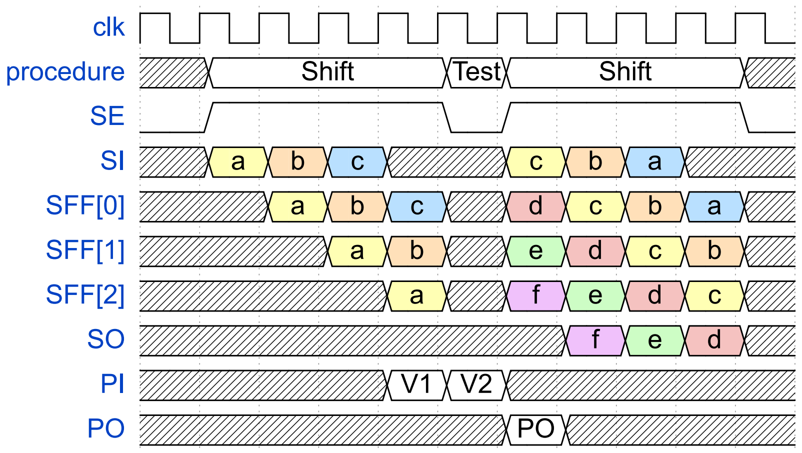


Figure 2.22: Waveform of Scan Design Launch-on-Shift Functionality

2.7.2. Transition Delay Fault Test

A test pattern used to test a TDF consists of a pair of test vectors (V1, V2). The first vector V1 is the initialization vector, and vector V2 is the launch vector. The steps for testing a TDF, illustrated in Figure 2.22, are as follows:

1. **Initialization Cycle:** The circuit is initialized to a certain state by vector V1 (and optionally scan chains), i.e. for a slow-to-rise TDF the fault site of the TDF is initialized to '0'.
2. **Launch Cycle:** Vector V2 is applied and a transition is launched at the target gate terminal, i.e. a '0' to '1' transition at the fault site of a slow-to-rise TDF. This cycle is indicated in Figure 2.22 by V2 being at the primary input (PI).
3. **Capture Cycle:** The transition is propagated and captured at an observation point. In Figure 2.22 these observation points are PO, and the scan flip-flops.

There are three TDF scan pattern generation methods: launch-on-shift, launch-on-capture, and enhanced scan. These have different timing for launching test patterns. The aforementioned methods will be briefly covered now.

Launch-on-Shift

In launch-on-shift (LoS), also called skewed-load [40], patterns the last shift of the scan chain load is used to launch the transition. A TDF scan test can be modelled as $< v_1, s_1, v_2, s_2 >$, where v_1 and v_2 are the pair of test vectors of the TDF test. Furthermore, s_1 and s_2 are sets of state variables for the initial state of the circuit (after the initialization cycle) and the captured state of the circuit respectively. Figure 2.23 shows a full-scan circuit, where $y_1 - y_3$ are the state input variables, $Y_1 - Y_3$ are the state output variables, x_1 and x_2 are input vectors, and z_1 and z_2 are primary outputs. LoS patterns shift s_1 in through the scan chain, and shift s_2 out. This means that if pattern i certain values for y_1 and y_2, pattern $i + 1$ has these values for y_2 and y_3 respectively. This results in overlap between consecutive state variable vectors, which is a limitation of LoS patterns.

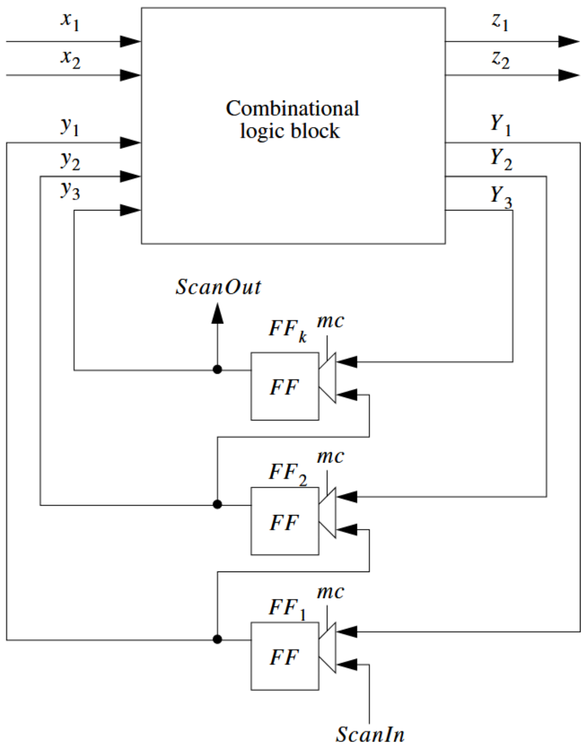


Figure 2.23: Example Full-scan Circuit from [36, p. 656]

Launch-on-Capture

Launch-on-capture (LoC) is also known as broadside [41]. This TDF pattern generation method is very similar to LoS. After pattern i applies s_1, v_1 and v_2, the state variable s_2 is equal to s_1 for the next pattern $(i+1)$. This is functional behaviour, which is why LoC is also called functional justification. But this does limit the flexibility of possible TDF patterns. Figure 2.24 shows LoC test patterns, consisting of test vector pairs V_1, V_2, V_3, V_4, and V_5, V_6. After the vector pair V_1, V_2 is applied the state is already in the correct state for the vector pair V_3, V_4. The same is the case for the next vector pair.

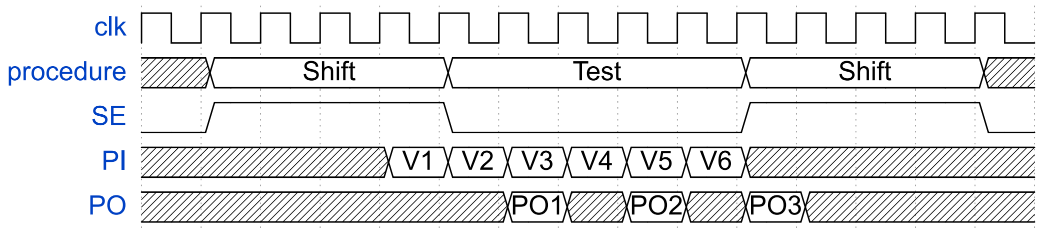


Figure 2.24: Waveform of Launch-on-Capture Pattern

Enhanced Scan

The last TDF pattern generation method is enhanced scan [42]. Enhanced scan uses double-strobe flip-flops [43], in Figure 2.25 it can be seen that a double-strobe flip-flop is a scan flip-flop with a hold latch on its output. An extra signal is added to control these hold latches. This setup enables the application of any arbitrary vector pair to test a TDF. Resulting in a higher fault coverage than LoS and LoC patterns at the cost of more area overhead.

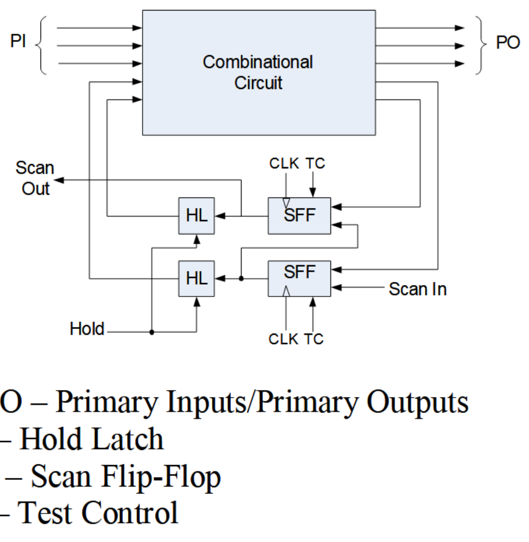


Figure 2.25: Enhanced Scan Flip-Flop [42]

2.7.3. Path Delay Fault Test

The PDF fault model has been described in Section 2.4.2. From that description and the previous section about TDF testing it can be deduced that testing is very similar when targeting PDFs and TDFs. When a TDF test pattern is executed, it will detect any TDFs along the entire propagation path. However, it will only be able to detect one PDF which consists of the cumulative delay along this path. This is one of the reasons PDF testing is significantly more expensive than TDF testing. Another reason is that there will be significantly more faults in a PDF fault list than in a TDF fault list. That is why PDF testing is not practical. Tests for PDFs can be divided into two types: robust, and non-robust PDF tests.

Robust PDF Test

A test pattern $< V_1, V_2 >$ is a robust test pattern for a PDF *“if and only if it detects a path fault independent of gate delays in the rest of the circuit.“* [44]. This means that a robust PDF test guarantees an incorrect value will be propagated to the end of the path, if the delay of the path under test exceeds a specific duration. This sounds like a desired characteristic of a test. However, it can be hard to find a robust PDF test. To simplify the ATPG process, non-robust PDF tests can be used.

Non-Robust PDF Tests

From the definition of robust PDF tests, the definition of non-robust PDF test can be inferred. Namely, non-robust PDF test guarantees the detection of the targeted PDF only if no other faults are present. Non-robust PDF tests can again be subdivided into two types, weak and strong. The former is defined as follows *“A strong non-robust test does not allow delayed transitions on off-path inputs to invalidate the test, but does not consider pulses“* [45]. So it assumes every off-path input to have a non-controlling value for the gate it drives, or pulses on off-path inputs happen early enough to not affect the propagation of the target fault. Weak non-robust PDF tests are defined as *“A weak non-robust test requires only that the path be sensitized. It allows delayed transitions and pulses on off-path inputs to invalidate a test“* [45]. This definition is self-explanatory. From both definitions it can be concluded that the weak tests give less chance of a test pattern actually succeeding in detecting a PDF.

3 Silent Data Errors

This chapter contains the literature study focusing on SDEs. This is a broad and popular topic within the testing community. That means that there are ample publications relevant to this issue. In Section 3.1, the chapter starts by introducing the problem and its newly revived relevance. This is followed by Section 3.2, diving into the nature of SDEs and its characteristics. Next, Section 3.3 discusses SDE rates and how they can be measured. Then, Section 3.4 describes techniques that can help mitigate the consequences of SDEs. Lastly, the chapter will end with Section 3.5, covering SBST in depth.

3.1. Silent Data Errors

SDEs are errors that 'silently' take place in computers, also known as silent data corruptions (SDCs). The consequence of these errors is corrupted data, meaning that somewhere in memory incorrect information is stored. This could cause be a pixel in a saved image to be a slightly different color, but it could also be something more problematic like an extra zero in the amount of money sent over a bank transfer. These errors are so called 'silent' because as they are occurring no software level or hardware level alarms are going off, and even weeks later they can still be undetected. This 'silent' nature can often lead to months of debug engineering time [5], when trying to find the root cause.

A key question in research about SDEs is: what causes silent data errors? An intuitive answer to this is soft errors due to cosmic rays [6], this is a widely known phenomenon. This phenomenon occurs when a radiation event creates a charge disturbance that causes the data content of a memory cell to flip. This is a straightforward cause of SDCs, as it happens silently and corrupts data. Adding to that, the majority of the area on most modern chips is used for memory. Meaning that cosmic rays are more likely to hit memory parts of a chip. To tackle this source of SDEs, error correction codes (ECC) are an effective method. ECC are a method to detect and correct errors upon memory accesses. The main purpose of ECC is to correct for noise that randomly occurs while reading from memory [46].

Another possible source of SDEs are physical defects due to manufacturing and ageing [47]. During the chip manufacturing process wafer inspection is applied, to identify chips with defects and avoid them from being used. So the defects causing SDEs fall under the category 'escapes', as they are defective but escaped detection at the manufacturing plant. SDEs are extremely rare and unlikely to happen. If they are caused by a defect, the defect is only prevalent in very specific conditions. It could also be assumed that the defect is not easily sensitized and observed, else it would have been detected at the manufacturing plant.

3.1.1. Calls-to-Action

In 2021 there have been two calls-to-action, in the form of papers [5][4], from the hyperscalers Meta and Google. In these papers Google and Meta illustrate the problem of SDCs in their at-scale infrastructure. They call for a collaborative effort between industry and the test research community to address this issue. In addition, they provide unparalleled information and insight into SDCs in data centres. Their access to at-scale infrastructure provides them with the unique opportunity to study a large number of SDCs cases. This infrastructure consists of millions of processors with multiple cores, making SDCs a frequent appearance.

Meta was the first to publish its paper on SDCs. They argue against the prior belief that SDCs are mainly caused by cosmic rays, and claim empirical evidence supports that SDCs occur due to device characteristics and are repeatable at scale [5]. They also shift the scope of the problem of SDCs from the prior location of memory to a new area: functional blocks. ECC are able to correct SDCs that take place in memory, but functional blocks have minimal error correction. This is problematic as it leaves room for SDCs to manifest. Meta also found that SDCs in functional blocks can depend on specific data and system conditions. Data variation can change the paths that are activated and system conditions can change the propagation speed of paths. So, they mention timing path errors as a possible cause of these SDCs. The discovery of SDCs is often delayed due to their silent nature. Meta shows the debugging flow for a SDC from application level down to assembly. This process can require months of debug engineering time. This emphasizes the importance of either detecting SDC-prone machines or correcting SDCs.

A few months later Google's paper on SDCs followed, confirming that Meta's observations are similar to what Google is observing. They then introduce a new concept as a new cause of SDCs: "silent" corrupt execution errors (CEEs). This is defined as malfunctioning instructions due to manufacturing defects, which can only be detected by checking the result of these instructions against expected results [4]. These CEEs can occur long after the machine's start of life, and on specific cores on multi-core CPUs. Google calls those cores suffering CEEs; "mercurial" cores. Specific instances of CEEs observed by Google are: violations of lock semantics, data corruptions due to memory operations, deterministic AES mis-computation, corruption affecting garbage collection. These CEEs are hard to reproduce deterministically, due to dependence on workload, frequency, voltage, and temperature. CEEs are an emerging cause of SDCs, and there are several reasons for it. CEEs are rare so large-scale operations, which is a relatively recent phenomenon, allow them to happen more frequently. Feature sizes are getting smaller and smaller, so chips become less reliable. CPU architectures are increasing in complexity, which means that there are more avenues for CEEs to manifest.

3.2. SDE Nature

The newly found perspectives on the problem of SDEs has been discussed, and the importance of the issue has been emphasized. Next, to better understand the problem, its nature will be portrayed. In this section questions about SDE nature will be examined, to guide the eventual search for solutions. Questions like, what is the root cause of modern day SDEs? What type of defects can cause SDEs? How do SDEs propagate through the layers of computer abstraction?

3.2.1. Defects

Defects are undesirable physical deviations in chips that can impact their functionality or performance. These physical deviations also impact the yield from wafers at semiconductor fabrication plants, also known as fabs or foundries. It is also important to note that defects cause different type of SDEs, than cosmic rays, noise, or droop. The latter causes 'transient SDEs', this means the SDE happens only once randomly. Defects can cause 'persistent SDEs' that occur more than once, and are at least somewhat reproducible. According to a paper by Intel, one of the world's largest chip manufacturers [48], there are no unique or novel defect modes that manifest as SDE [49]. So, SDEs are caused by widely known types of defects that escape the post manufacturing testing procedures. The Test Technical Committee of the IEEE Electronics Packaging Society estimated that 80% of SDEs are due to time-zero test escapes [50].

Figure 3.1 shows SDE rates published by intel for different defect modes, circuit types, and logic function. It can be seen that a large majority of SDEs were traced to "subtle open defects". These are resistive-open defects where the transistor is still functional, but it has a higher threshold voltage *"driven by subtle marginality"* [49]. This indicates that marginal or subtle electrical behaviour and timing tolerance in silicon play a large role in whether a defects manifests as an SDE. For the different circuit types memory arrays, logic, and clock distribution there is no one large contributor. The same story goes for functional regions, data paths and control path logic both contribute equally. However, less failures were due to faults on both data and control.

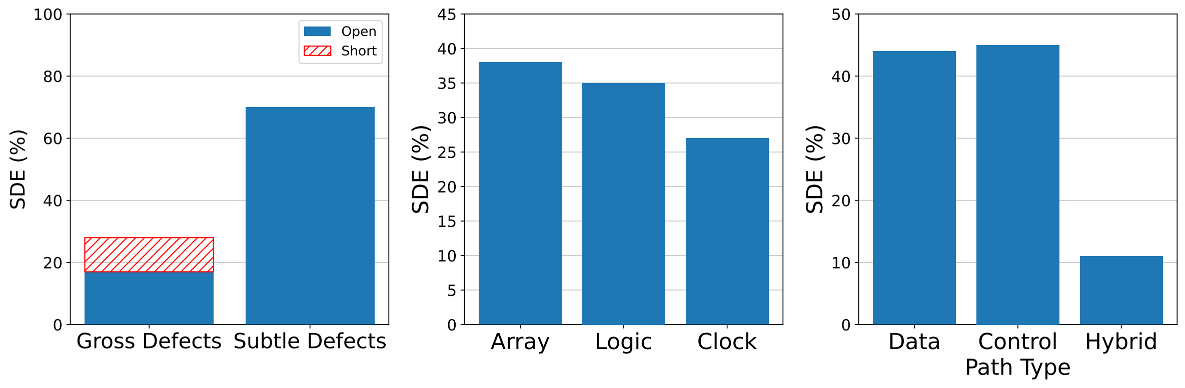


Figure 3.1: Defect Mode Classification of SDE Based on Impedance (left), SDE Distribution Across Circuit Types (middle), and SDE Distribution by Functional Region (right) from [49]

SDEs due to subtle timing marginalities induced by process variations has been further explored in the work of Adit Singh [51]. This work was prompted by published industrial data that points to timing failures caused by significant delay increases of random paths. Singh focuses on low voltage operation, because this accentuates circuit delays due to process variation. He shows that the delay of certain transistors, with a threshold voltage of multiple standard deviations from the mean, skyrockets when using lower voltage. Furthermore, simulations show that extreme outlier timing paths are virtually always caused by a single slow transistor, which becomes especially problematic at low voltage or low frequency. Another interesting insights provided by Singh [51], is that the delay increase due to increase in threshold voltage is systematically larger than the decrease in delay due to decrease in threshold voltage. So these do not cancel each other out. Finally, Singh proposes performing timing tests at lower voltage than threshold voltage to find the extreme outlier transistors as they scale differently from other transistors.

3.2.2. Propagation Through the Stack

To classify a defect as the cause of an SDE it is important to analyse the entire stack and how defects propagate through it. Figure 3.2 shows the computing abstraction hierarchy and some additional text. As can be seen, the bottom four layers are classified as hardware and the top 6 layers are classified as software. Hardware and software are connected through the ISA, this was discussed in Section 2.2.2. To the left of this classification, the stack is again divided into benign, masked, and SDE. A defect propagating through the stack is called benign when *"it exists, but it never appears at the software layer"* [52]. A defect can be benign due to hardware masking mechanisms, such as when a defect can be modelled as a SDF that does not affect functional behaviour. At the microarchitecture-level a fault has a chance to manifest as an SDE without needing to propagate all the way up to the application level. When a fault causes a wrong computation and the result of this is written to a memory this is classified as an SDE. A storage corruption like this does not get detected and corrected by ECC, because it is already corrupted before entering the memory. When a fault propagates up to the ISA level it becomes an architecturally visible fault [52], more details on this can be found in Section 3.3.1. An architecturally visible fault can propagate upwards to the application level, where it can be classified as an SDE. However, it can still be masked in software by propagating to an unused value or by causing a crash or a time-out. This prevents the defect from manifesting as an SDE. From this illustration it can be concluded that to classify a defect or faulty behaviour as a SDE the propagation through the whole stack needs to be simulated.

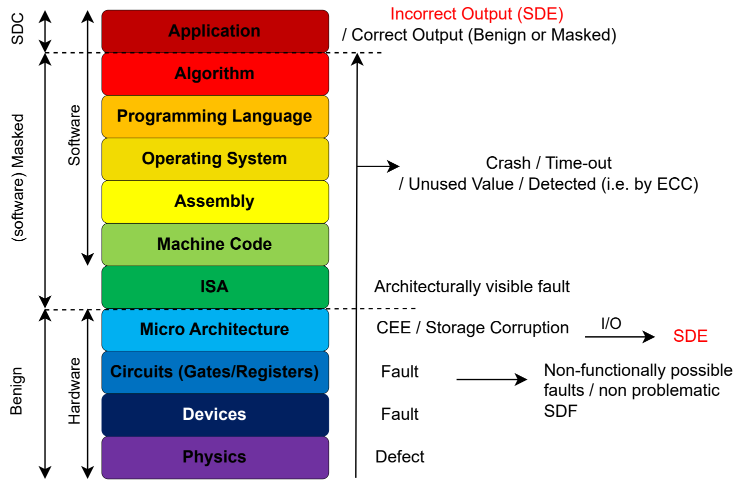


Figure 3.2: Classifying SDEs Throughout the Stack

3.3. SDE Rates

The previous section concluded by emphasising the importance of simulating the entire stack when attempting to measure SDEs. This section will discuss how SDEs are measured, and give insights into SDE rates. The most straight forward way to measure SDE rates, considering SDE occurrences are extremely rare, would be to have a fleet of processors available and run programs that can easily be checked for SDE behaviour afterwards. However, having access to enough physical faulty processors is a privilege only owners of extreme scale systems can afford. Therefore, simulations are most likely the most practical way to gain insight into SDE rates. This is the argument of Gizopoulos et al. [53] for using microarchitecture-level simulation in their work, and their other works that will be discussed in this section. Gizopoulos et al. also mention the observations of the hyperscalers discussed in Section 3.1.1, and point to processing elements having an unexpectedly large rate of SDEs. To investigate this Gizopoulos et al. claim microarchitecture-level simulation enables exploration into probabilities, rates, severities, and root causes of SDE for with high accuracy. Additionally, this measurement method has the best trade-off between speed and observability, in the context of a full stack simulation, including things like ECC mechanisms, and OS behaviour. Table 3.1 shows the methodologies they considered. They argue that from all the low cost options microarchitecture-level fault injection has the best observability for feasible speed. The only physical fault injection method they considered was beam testing. An alternative they did not mention or consider is stress testing a dozen or so processors by for example running them at low voltage or in a high temperature environment. This would induce faults, and measurements could be done to see how many of these faults manifest as SDE. Faults could not be injected in specific units, but SDE rates could be compared on varying benchmarks with different instruction profiles.

Table 3.1: SDE Rate Measurement Methodologies from [53]

Evaluation Method	Time Needed	Cost	Accessible Resources	Fault Source	Availability	Observability
Field, Lifetime data	months/years	very high	all	natural	final product	limited
Beam testing	hours	high	all	natural	final product	limited
Software-level fault injection	hours	low	limited	synthetic	early/final product	medium
Architecture-level fault injection	days	low	limited	synthetic	early	medium
Microarchitecture-level fault injection	**days/weeks**	**low**	**most**	**synthetic**	**early**	**very high**
RTL fault injection	years	low	all	synthetic	late	very high

3.3.1. Microarchitecture-level Fault Injection

Microarchitecture-level fault injection is used to estimate SDE rates. A microarchitecture-level simulator used in multiple studies that will be discussed in this section is gem5 [54]. Gem5 allows deterministic end-to-end execution of large workloads on an operating system. Gem5 simulates interactions between components like control units, execution units, and all major storage units. However, it only functionally models combinational and sequential logic. Fault injection is combined with microarchitecture-level simulation to try and induce SDEs. This is achieved using a microarchitecture-level fault injection framework like GeFIN [55]. GeFIN is built on top of gem5, and can classify outcomes of each fault injection as: SDE, timeout, crash, or assert. An SDE is measured when the simulation finished normally but the program output is different from the fault-free program output. When the simulation did not finish within a certain period of time it is categorized as a time-out. The simulation can also not output anything due to some catastrophic event, this is called a crash. Lastly, an assert is when the simulation encounters a high-level condition that the simulator can not handle, like an illegal address being used. GeFIN is able to inject different type of faults like permanent, transient, and intermittent. However, because combinational and sequential logic is functionally simulated it is limited to bit-flips in memory components, and at outputs of functional units. Additionally, in these micro-architectural studies often not all possible faults are injected. It is not feasible to inject in a reasonable time all possible errors in all locations at each clock cycle, which is why statistical fault injection (SFI) is used. During SFI a randomly selected subset of possible faults are injected, at random injection cycles. The sampling size of an SFI campaign can be computed using the population size (all possible faults at any cycle), desired margin of error, and desired confidence level [56].

Microarchitecture-level Results

This section will discuss some of the results from the microarchitecture-level papers. The goal of this is to paint a more complete picture of SDE behaviour within the entire system. Architecturally visible faults were introduced in Section 3.2.2, they are also known as corrupt execution errors (CEEs). There are five different types of CEEs:

- **Execution Time Error:** A completely correct instruction is committed in the wrong cycle.
- **Instruction Flow Change:** A different instruction is executed due to incorrect instruction fetching.
- **Instruction Replacement:** A different instruction is executed due to a corrupted opcode.
- **Operand Forced Switch:** When one or more operand fields are corrupted.
- **Data Corruption:** The correct resource is used but the content of the register is corrupted.

The work of Papadimitriou et al. [52] uses GeFIN to inject faults into on-chip storage structures of a CPU with a 64-bit Armv8 ISA. The fault model Papadimitriou et al. use is the transient single-bit fault, meaning a bit flip is injected into a storage structure once per simulation. They use SFI to inject 2000 bit flips per structure. Figure 3.3 shows the SDC probability of different hardware structures for each type of CEE. The SDC probability is the probability that a fault in a certain hardware structure manifests as a SDC. Data Corruption is by far the most common type of CEE, with a 53.1% SDC probability for the L1 data cache. When looking at the definition of data corruption this makes sense. If data in the L1 data cache is corrupted due to a fault, when loading this data into a registers its content is also corrupted. Furthermore, it is to be expected that data corruption is the most common type of CEE. This is due to the fact that data corruption is the least likely type of CEE to cause a crash, so the most likely to stay silent and become a SDE.

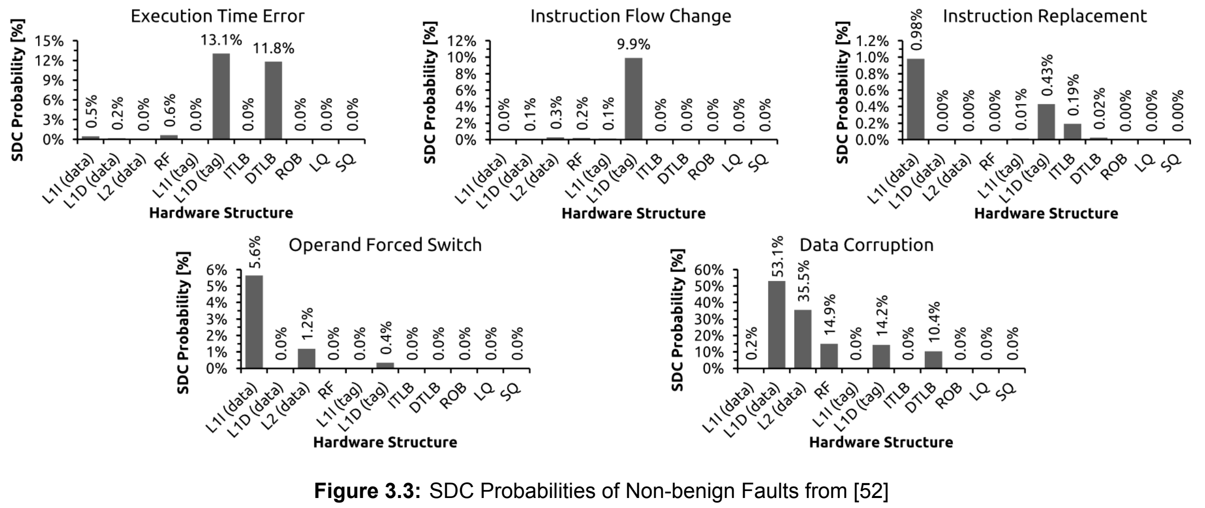


Figure 3.3: SDC Probabilities of Non-benign Faults from [52]

An earlier work of Papadimitriou et al. [57] uses the exact same setup, also using GeFIN and injecting 2000 single bit flips into each storage structure. However, they compare two different microarchitectures, and also different compiler optimization levels (O0, O1, O2, and O3). Figure 3.4 shows the FIT rate for both microarchitectures per optimization level for eight different benchmarks. FIT stands for failures in time, and is the number of failures that can be expected in one billion device-hours of operation. Furthermore, the FIT rates are also categorized into application crashes, system crashes and SDCs. It is interesting to see that the more modern architecture is more vulnerable to SDCs. This supports the claims made by Meta [5] and Google [4]. The benchmarks dijkstra and sha have the lowest SDC rates. This might be due to the fact that a graph algorithm and a cryptographic hash algorithm both use relatively little memory, and the fault are injected into memory structures.

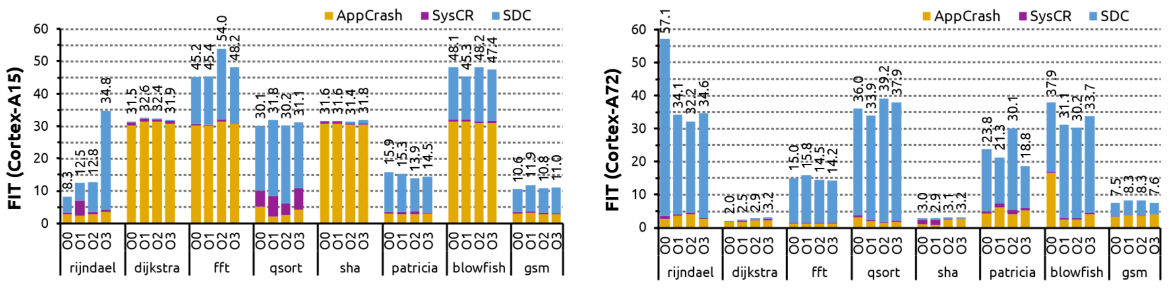


Figure 3.4: Arm Cortex-A15 FIT rates (left graph) and Arm Cortex-A72 FIT rates (right graph) from [57]

A work by Gizopoulos et al. [58] also uses GeFIN but injects different faults. Gizopoulos et al. inject transient and permanent bit-flips into storage structures, as well as permanent faults in an integer adder. However, microarchitecture-level simulation only simulates functional units functionally. Therefore, statistical models were created to describe the propagation of permanent stuck-at faults in the integer adder to its output. Essential this is a frequency of a stuck-at or bit-flip at the output. Figure 3.5 shows the SDC AVF for different ISAs per benchmark, for permanent faults in the L1D Cache and the integer adder. SDC AVF is simply the the probability of a fault leading to a SDC. The data cache has a higher SDC AVF than the integer adder, which is to be expected. Due to their statistical model many injected faults are masked in the integer adder, while there is no masking in the data cache. Furthermore, in both plots RISC-V seems to be the ISA that is most susceptible to SDCs.However, this could differ for different implementation of each ISA. This is a nuance that was not mentioned by Gizopoulos et al.

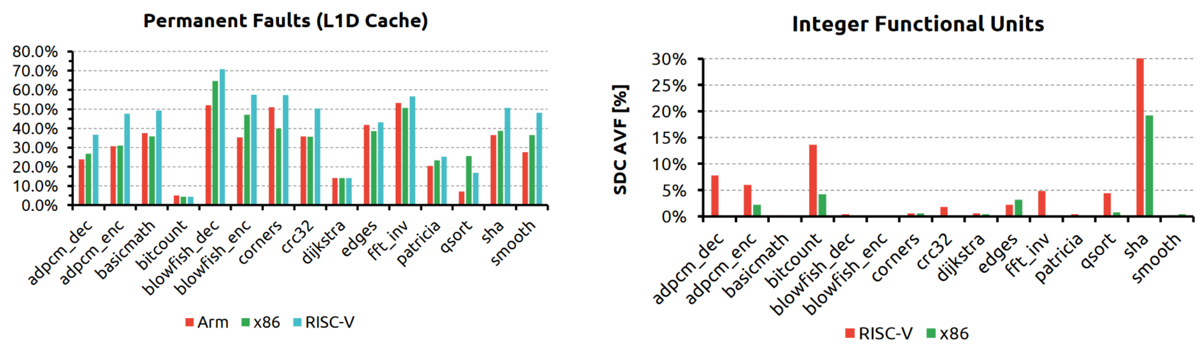


Figure 3.5: SDC AVF for Permanent Faults in L1D Cache and Integer Adder from [58]

A more recent work by Gizopoulos et al. [53] does not use GeFIN to inject fault but Gem5-marvel [59]. Gem5-marvel is a fault injection framework built on top of gem5, that is able to inject faults into functional units. It uses gate-level models of functional units in C++, on which stuck-at fault injection can be done. Besides memory structures they also inject faults into an integer adder and multiplier, and a floating-point adder and multiplier. Figure 3.6 shows the SDC probability of the aforementioned functional units for different benchmarks. As can be seen, floating-point adder and multiplier units are significantly more SDC prone than their integer counterparts. This could be due to increased logic complexity, or because the benchmarks use more floating point arithmetic than integer arithmetic. This study should have taken into account the instruction profile of the benchmarks when presenting results.

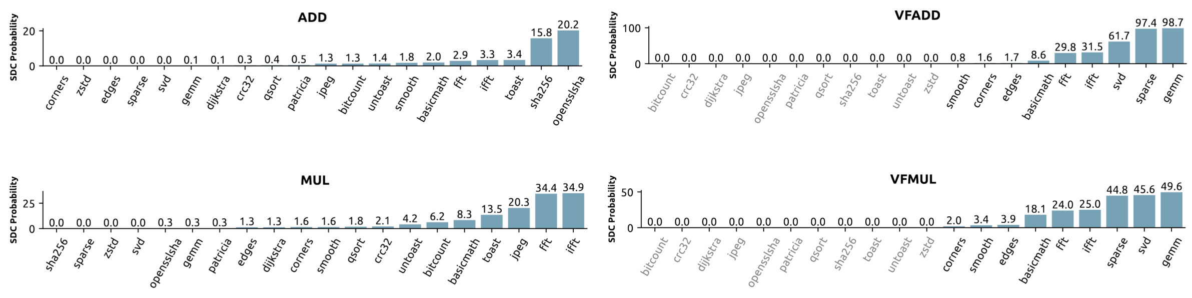


Figure 3.6: SDC Probability of Multiplication and Addition Arithmetic Units for Integers and Floating Points Numbers from [53]

The discussed microarchitecture-level works have been of the same group of authors. Therefore, some other microarchitecture-level fault injection works will briefly be covered to show some different approaches. Li et al. [60] developed a fault injection framework that models microarchitecture-level effects of gate-level permanent faults. Enabling them to compare SDC occurrences due to permanent SAFs with SDC occurrences due to permanent TDFs. As expected, of these two fault models TDFs is most likely to cause SDCs. Hyungmin Cho [61] has shown that FPGA emulation can speed up RTL-level fault injection combined with microarchitecture-level simulation. In contrast, Wibowo et al. [62] went up an abstraction layer and combined functional simulation with microarchitecture-level simulation to speed up execution.

3.3.2. Comparison

The discussed microarchitecture-level SDC works have been compared in search of their limitations and potential avenues for improvement. Table 3.2 shows the components that each work has performed fault injection in. As can be seen many functional units have not been covered.

Table 3.2: Qualitative Comparison of Fault Injected Components

Component		[57]	[58]	[52]	[53]	[60]	[61]	[62]
Memory		✓	✓	✓	✓		✓	✓
Functional Units	IntALU					✓		
	IntMultDiv							
	FP_ALU							
	FP_MultDiv							
	ReadPort							
	SIMD_Unit							
	PredALU							
	WritePort							
	RdWrPort							
	Int Add		✓		✓			
	Int Mult				✓			
	VF Add				✓			
	VF Mult				✓			
	Decoder					✓		
	AGEN					✓		

Table 3.3 shows with combination of fault types and fault models have been used for fault injection in each work. In terms of transient faults only single-bit faults have been injected, and injection of intermittent faults has not been explored whatsoever.

Table 3.3: Qualitative Comparison of Fault Models used for Fault Injection

Fault Type	**Fault Model**	[57]	[58]	[52]	[53]	[60]	[61]	[62]
Permanent	Bit-flip fault		✓			✓		
	Single-bit fault		✓		✓			
	Gate-level fault		✓		✓			
	SA fault		✓			✓		
	Delay fault					✓		
	Bridging fault							
	Transistor fault							
Transient	Single-bit fault	✓	✓	✓	✓		✓	✓
Intermittent	Any							

3.3.3. Alternative SDE Estimation Methods

There are alternatives to microarchitecture-level fault injection for estimating SDE rates. However, most works use microarchitecture-level fault injection due to its efficiency, this was discussed at the start of Section 3.3. One of these alternatives is data from deployed hardware, but there are not many entities with access to large fleets of processors. Another alternative is beam testing, this is a technique using focused ion or electron beams. These beams can be used to inject faults at specific location within circuits to observe their effects. This method can be very costly if precision of the fault location is required. There is also software-level and architecture-level fault injection, but these are not as detailed as microarchitecture-level fault injection. The more detailed approach RTL fault injection suffers from long run time. For these practical reasons, most of the works found that report SDE rates use microarchitecture-level fault injection.

3.3.4. Conclusion

To conclude this chapter about microarchitecture-level SDC works, a lot of similar studies with little additional novelty were discussed. However, these studies did help paint a picture of how faults can turn into SDCs. Except for the work of Hyungmin Cho [61], no works have compared different fault models for varying fault types. This should be investigated, maybe different fault models will have different correlations to SDCs or CEEs for various functional units or ISAs. The fault models currently used are not good enough, more complex fault models will have different propagation from gate level to microarchitecture-level. For example, transition delay faults depend on two cycles, and what they use now does not consider the transition of signals between clock cycles. Furthermore, not many functional units have been considered for fault injection, i.e. no fixed point units or dividers. This could be interesting to see how it relates to the different types of CEEs. Combining different abstraction levels to get more detailed when necessary, and switch to a faster less detailed abstraction level when applicable is very promising and should be explored more. No intermittent faults are explored at all, while Google [4] and Meta [5] point to these as a potential cause of a lot of SDCs. Microarchitecture-level fault injection of intermittent fault for different fault models would provide great insights.

3.4. SDE Solutions

Mitigating the consequences of SDEs can involve either hardware or software solutions. The goal of SDE solutions is to prevent SDEs from occurring. This can be achieved by detecting and correcting SDEs in real time. But it could also be done by identifying circuits suffering from SDE occurrences, and avoiding their usage. The software solutions that will be covered in this section include software test libraries, and software redundancy. Hardware solutions that are discussed involve continuous verification, and structural testing. Any combination of these techniques can be used to mitigate SDEs.

3.4.1. Software Test Libraries

Software test libraries (STLs) for SDEs are software libraries that can be used to detect processor defects that can cause SDEs. They are practical because they can be ran on processors that have some down time. Therefore, they are used in modern data centres as an in field maintenance tool. These tools run as full-system diagnostics and execute a wide variety of code sequences to test every CPU subsystem. An example of an STL is DCDIAGS [63], a tool that consists of a suite of tests.

The suite verifies if every computation returns correct results by using self checking mechanisms for example, multi-threaded arithmetic or cryptographic routines, and checking each result for correctness. Figure 3.7 shows some of these mechanism, this is elaborated on below:

- **Golden-Value Tests:** The tool runs a fixed sequence of operations with a known input and compares the result to a "golden" reference. For example, it may compute a checksum or a mathematical function of a predetermined value and check that the output matches the expected constant. Any discrepancy flags the core as defective.
- **Cross-Thread (Multi-Core) Comparison:** All cores in the processor execute the same instruction sequence with identical initial data. At the end of each iteration, the outputs from every thread are compared against each other. Any core that produces a deviating result is marked as faulty.
- **Inverse-Transformation Tests:** A computation is paired with its inverse (i.e. compress then decompress, or encrypt then decrypt) on a random data buffer. The final output should exactly match the original input, else an SDE is inferred.

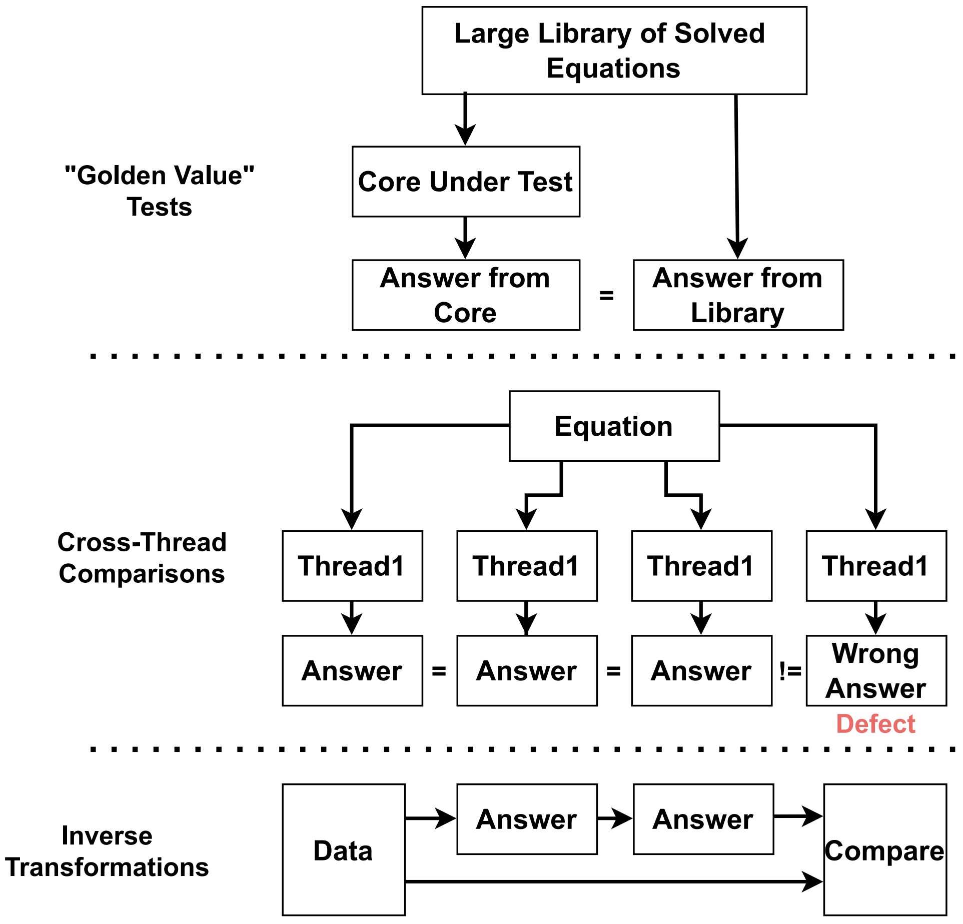


Figure 3.7: Software Self-checking Methods for Data Centre SDEs

These software tests turn the CPU into its own self-checking test engine. Intel reports that DCDIAGS tests have uncovered unique SDE causing failure mechanisms that escaped all other testing [63]. STLs are currently used both in chip manufacturing and in data centre maintenance. In production test, DCDIAGS is run on every wafer or packaged part to filter out "tails" of undetected defects. In the field, administrators typically schedule periodic offline tests: they take servers down for maintenance and run diagnostics (including DCDIAGS) on each CPU. This routine maintenance can catch processors that only reveal SDE faults after some ageing.

However, STLs have some downsides. To successfully detect SDEs, tests need to be repeated many times. This is due to the intermittent behaviour of defects causing SDEs. So individual tests consisting of specific computations will be run repeatedly. Therefore tests that target SDE defects have a huge test cost. This is characterized by time to failure (TTF), which is the length of time an individual test must run to detect a defect. Long TTFs causes large test costs. Moreover, STL development is based on stressing all architectural units of the processor and hoping to detect defects. It is a form of functional testing meaning it is not based on fault models. Therefore, test development is not systematic, but extremely ad hoc. Lastly, this also means that the thoroughness of these tests is unclear. The thoroughness can not be expressed in FC, and the only way to evaluate tests is to see how many defects are detected.

3.4.2. Software Redundancy

Software redundancy techniques replicate program computations in software to detect and correct defects that lead to SDCs. STLs, discussed in the previous section, used software self checking mechanism which in essence are also a form of software redundancy. However, the software redundancy that will be discussed in this section does online detection and correction of incorrect computations. Two primary software redundancy approaches that have been adopted in are full duplication and selective instruction duplication (SID). In full duplication, every instruction is executed twice or more and the results are compared at runtime. Figure 3.8(a) shows an example of this. While conceptually simple, this approach costs nearly 100% performance overhead and doubles the memory footprint of programs, making it impractical for large-scale or latency-sensitive workloads.

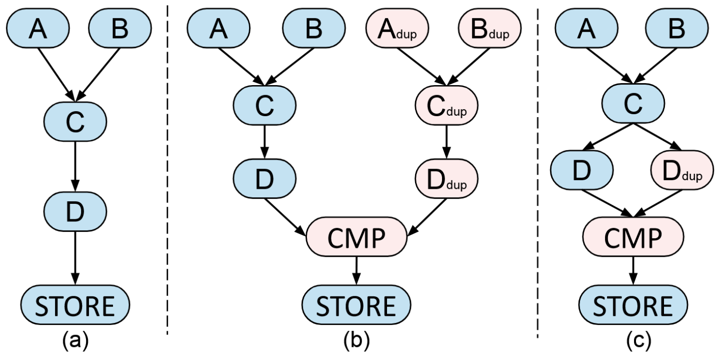


Figure 3.8: Example of (a) Original Program (b) Full Duplication (c) SID from [64]

SID reduces overhead by duplicating only a subset of vulnerable instructions. Figure 3.8(b) shows an example of this. Each candidate instruction is duplicated in the binary, and a comparison is inserted immediately before the next control□flow or synchronization point. If a transient fault flips a bit in either of the two identical instructions, the mismatch triggers an error detection event. SIDformulates the selection of instructions as a 0–1 knapsack problem, where each instruction's cost is its fraction of dynamic cycles, and its benefit is the product of that cost and the measured SDC probability from a compiler-level fault injection run. Equation 3.1 and Equation 3.2 show the formulas for cost and benefit respectively, of instruction i.

$$\text{Cost}_i = \frac{(\text{Dynamic Cycles})_i}{\text{Total Cycles}}, \tag{3.1}$$

$$\text{Benefit}_i = (\text{SDC Probability})_i \times \text{Cost}_i. \tag{3.2}$$

Within a given performance-overhead budget, SID maximizes total benefit by duplicating only the most promising instructions. Because SID measures $(\text{SDC Probability})_i$ using a single reference input, it can overlook instructions whose fault propagation behaviour changes with different data sets. Huang et al. [64] observe that ~ 16% of instructions, in typical high performance computing kernels, exhibit near-zero SDC probability under the reference input, but become highly vulnerable under other inputs.

As a result, conventional [64] may suffer up to a 37% drop in coverage when the program runs on arbitrary inputs.

To address input dependence, MINPSID [64] augments SID with an input□search engine based on a genetic algorithm and control-flow-graph (CFG) fitness metric. It:

1. Statically analyses the program to build its CFG.
2. Iteratively generates and mutates inputs, scoring each by the difference in their weighted CFG profiles (edge-execution counts) to explore diverse execution paths.
3. Performs targeted fault injection on each new input to uncover incubative instructions (instructions with which SDC coverage loss correlates).
4. Updates each incubative instruction's benefit to its maximum observed value across all inputs, ensuring they are selected in the final knapsack optimization.

This process effectively recovers the lost coverage: MINPSID reduces the average input-induced coverage drop from 37% to under 3% across 11 benchmarks. However, the protection levels that were used in the work were 30%, 50%, and 70%. The protection level *“indicates the amount of dynamic instructions need to be duplicated“* [64]. So, the protection level equals the runtime overhead, which is quite significant considering the occurrence frequency of SDCs. Laguna et al. developed IPAS [65], a SID method that exploits machine learning to find incubative instructions. This improved runtime overhead compared to the work of Huang et al. [64]. But, the overhead still ranges from 4% to 35% which is not feasible for most applications.

3.4.3. Continuous Verification in Hardware

Continuous verification in hardware entails hardware verifying its own operations to detect faulty behaviour when it happens. These self-checking functional units raise an exception or log an error when a discrepancy is detected. This ensures that a potential SDE loses the S and therefore is not silent. Of course, continuous hardware checking increases area or energy cost, so it is typically applied to the most vulnerable or safety-critical paths. However, in contrast to software redundancy, continuous verification in hardware can be designed to cause less runtime overhead.

The example that will be discussed is a low cost hardware scheme for continuous verification of large arithmetic circuits by Pan et al. [66]. The scheme is based on the relation shown in Equation 3.3. The relation considers two numbers b_1 and b_2, and the modulo n of these numbers, a_1 and a_2 respectively. It states that multiplying a_1 with a_2 is equal to multiplying b_1 with b_2 and taking the modulo n of the result.

$$\text{If } a_1 \equiv b_1 \pmod{n} \wedge a_2 \equiv b_2 \pmod{n} \implies a_1 \times a_2 \equiv b_1 \times b_2 \pmod{n} \tag{3.3}$$

The relation can be used to self-check a multiplication, and also works for other arithmetic operations like addition and subtraction. Figure 3.9 shows the self-checking circuit based on this relation.

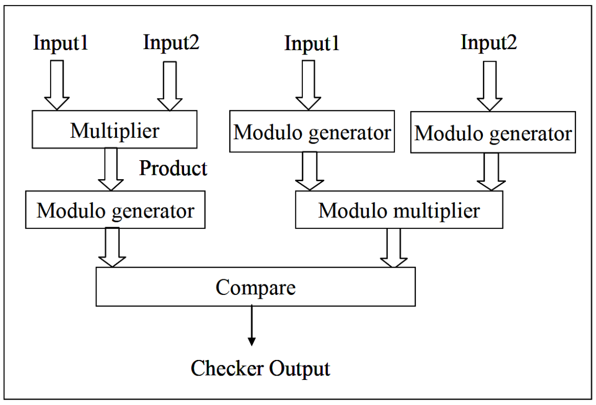


Figure 3.9: Self-Checking Multiplier Circuit from [66]

Table 3.4 shows the resulting FC and overhead for different modulo n implementations, and other works. The fault coverage is not complete, because of escapes due to ambiguous cases like $9 \pmod{3} =$

0 and $12 \pmod 3 = 0$. Using the same logic it can be explained why for higher n there is higher FC, this is because there will be less ambiguous cases. Lastly, when comparing the scheme of Pan et al. [66] with the other works, it becomes clear that a nuanced trade-off between area overhead and FC is worth it.

Table 3.4: Area Overhead and Fault Coverage Comparison for 32-bit Multiplier from [66]

Detection Scheme	Mod 3	Mod 7	Mod 31	Mod 127	Nicolaidis [67]	Marienfeld [68]
Area Overhead %	14.5	16	18.7	21.7	46 to 76	28 to 35
Fault Coverage %	93	94	97	97	100	100

3.4.4. Test

This section covers test solutions, which include techniques from the field of structural hardware testing. This is seen as a hardware solution, as structural testing is based on the underlying hardware. It can mitigate SDEs by ensuring circuits with defects, that could manifest as SDEs, are not used. The underlying cause of SDEs is thought to be *“timing errors from random process variations“* [10]. The test might well be the best solution because its objective is to detect the root cause of SDEs: defects. Moreover, the response of the industry in the past few years has been the brute force solution STLs, discussed in Section 3.4.1. Current test infrastructure made for traditional ATE, can not be used to apply STLs. This would require many test boards in parallel to deal with the long test times of functional tests, as well as cooling infrastructure which is needed to cool CUTs that operate at-speed. That is why *“new low-cost scan tests that can reliably detect timing failures caused by random process variations remain of great interest to industry“* - Adit Singh [10]. The test communities response to this will be discussed in this section.

Vmin Testing

In Section 3.2.1, testing at low voltage was established to be a way to accentuate subtle timing marginalities, and thus be able to detect them better. Adit Singh published another work on this topic [69]. Table 3.5 shows that the chance of an outlier transistor causing a problematic delay is significantly higher at V_{min} than at V_{nom}. He also showed how this can be leveraged for setting the voltage and timing of TDF and CAT scan tests to help increase the detection of these marginal parts.

Table 3.5: Gate Delay vs $\triangle$Vth and Occurrence Probability from [69]

$\triangle$Vth		**Percentage Delay Change due to $\pm\triangle$Vth**				**Probability of**
σ	**Volts**	**VDD = 1.0V**		**VDD = 0.6V**		**Vth > Vth0 + $\triangle$Vth**
0	0.000	0%	0%	0%	0%	1 in 2.0E0
1	0.025	-4.9%	+5.4%	-13.7%	+18.1%	1 in 6.0E0
2	0.050	-9.6%	+11.4%	-24.4%	+43.2%	1 in 4.4E1
3	0.075	-13.6%	+18.1%	-32.7%	+79.9%	1 in 7.4E2
4	0.100	-17.5%	+25.6%	-39.7%	+137.8%	1 in 3.2E4
5	0.125	-21.0%	+33.9%	-45.5%	+240.7%	1 in 3.5E6
6	0.150	-24.3%	+43.2%	-50.3%	+465.6%	1 in 1.0E9
7	0.175	-27.3%	+53.8%	-54.4%	+1018%	1 in 7.8E11

Singh’s approach is tried by intel in a work by Natarajan et al. [70]. They test the Timing-Aware Transition (TAT) and the Timing-Aware Cell-Aware (TACA) fault models under different temperature and frequency conditions on data centre products. To see if the V_{min} can be increased, because an increased V_{min} means that the circuit would not function correctly with the original V_{min}. Table 3.6 shows their results. In these results class units are a set of packaged units that have not been tested on a system, and SDE units are units that have been tested and failed due to SDE. The testing frequencies F1 and F4 related to each other in the following way: $F4 = 4 \times F1$. It was concluded that high frequency testing at low temperature accentuates timing marginalities the most, because the V_{min} for which the units were functional went up the most under these conditions.

Table 3.6: Distribution of Units that Showed Vmin Elevation with TA Content from [70]

Temp	Percentage of 108 Class units with at least one core having ≥ 40mV increase over baseline						Percentage of 7 SDE units with at least one core having ≥ 40mV Vmin increase over baseline					
	F1			F4			F1			F4		
	Total (TA)	TAT	TACA	Total (TA)	TAT	TACA	Total (TA)	TAT	TACA	Total (TA)	TAT	TACA
hot	26.9	20.4	14.8	0	0	0	71.4	71.4	57.1	0	0	0
cold	45.4	38.9	20.4	73.1	73.1	7.4	71.4	71.4	57.1	85.7	85.7	42.9

Process Variation Aware Testing

To attempt to tackle SDEs caused by defects induced by process variation, there has been a string of recent works on process variation-aware testing. These recent works build upon the work by Sauer et al. [71]. Process variation is the root cause of hard-to-detect marginal defects currently escaping post manufacturing testing. Therefore, current testing could likely be improved by incorporating process variation. That is the main motivation for the work of Jafarzadeh et al. [72], in which they propose a pattern generation method that targetsSDFs under process variation. Moreover, the generated tests are robust, which means that it detects the targeted SDF regardless of other delays. Jafarzadeh et al. also define a new metric for fault efficiency of SDFs under process variations. Equation 3.4 shows the fault efficiency of test set T for fault size i. The numerator is the amount of faults with size i detected by T for all circuits (c) in circuit set C. The denominator is the amount of detectable faults of size i in FL for all circuits in C.

$$FE(i,T) = \frac{\left|\bigcup_{c\in\mathcal{C}} F(c,i,T)\right|}{\left|\bigcup_{c\in\mathcal{C}} \{(f,c,i)\in FL\}\right|} \tag{3.4}$$

Figure 3.10 shows the proposed method. First, a transition fault test set is generated, as well as a training set of 100 Monte-Carlo instances of the CUT. Each instance consists of randomly sampled transistors from a 14nm FinFET transistor model [73]. Then, a SDF fault list is generated by estimating the minimum detectable fault size using a static timing analysis tool. The fourth step involves robust test pattern set generation for the fault list, and set of circuit instances. This is followed by timing-aware test set compaction. Finally, the resulting test set is validated using a validation set of Monte-Carlo circuit instances.

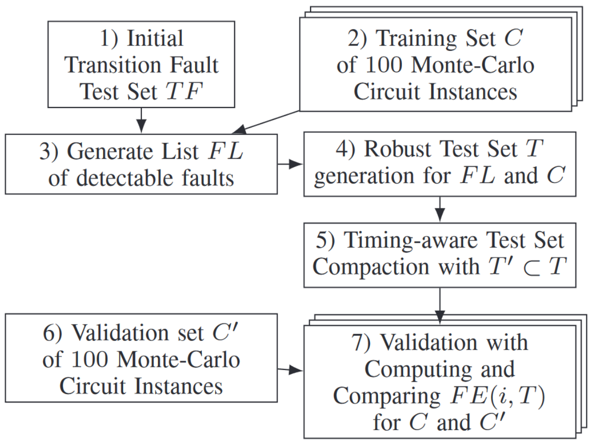

1) Initial Transition Fault Test Set TF
2) Training Set C of 100 Monte-Carlo Circuit Instances
3) Generate List FL of detectable faults
4) Robust Test Set T generation for FL and C
5) Timing-aware Test Set Compaction with $T' \subset T$
6) Validation set C' of 100 Monte-Carlo Circuit Instances
7) Validation with Computing and Comparing $FE(i,T)$ for C and C'

Figure 3.10: Overview of Proposed Method from [72]

The algorithm used for step four is shown in Algorithm 1. The inputs consist of the SDF fault list (FL) generated in step 3, and the trained set of circuit instances (C). A TDF test set acts as the base test set and will be expanded by the algorithm. The algorithm starts with fault size $i = 1$ and iterates until $i = K$ or FL is empty. Every iteration the algorithm does the following for fault size δ_i. First, timing-aware fault simulation for all SDFs of size δ_i is performed. Then, the detected faults (f, c, δ_i) (fault location, circuit instance, fault size respectively) are removed from FL as well as all faults (f, c, δ_j), where $j \geq i$. The authors make the reasonable assumption that if an SDF of a certain size is detected, all SDFs at the same location and circuit of a larger size are also detected. After this, the circuit c with the lowest

SDF FE is found, and N-detect [74] ATPG for TDFs is applied on undetected fault locations in c. From the generated patterns, ones that detect new SDFs in C are added to the test set. This process is repeated until $i = K$ or FL is empty.

Algorithm 1 Test Pattern Set Generation under Variations from [72]

```
INPUT: $FL, C$
OUTPUT: Test set $T$ for SDFs
 1: Set $i = 1$.
 2: Generate transition fault test set $\tilde{T}$.
 3: for $i \leq K$ do
 4:     Perform timing-aware fault simulation of all SDFs of size $\delta_i$ in $C$ with $\tilde{T}$.
 5:     Drop the detected faults $(f, C, \delta_j)$ with $j \geq i$ from $FL$.
 6:     Find $c \in C$ with the lowest SDFs efficiency.
 7:     Apply N-detect ATPG for transition faults on undetected fault locations in $c$ and get $T_{TF}$.
 8:     Add patterns detecting new SDFs in $C$ to $\tilde{T}$.
 9:     $i = i + 1$.
10: end for
11: $T = \tilde{T}$
```

Algorithm 2 shows the algorithm for test pattern set compaction under process variations. The inputs of this algorithm are test set T, fault list FL, and training set of monte carlo circuits C. The output is a compacted test set T' for SDFs. As the algorithm initializes two sets $FL_{detected}$ and T'. Then, the algorithm loops through each test pattern in T. Per test pattern it first performs timing-aware fault simulation to find the set of faults $F(t_j)$ in FL detected by t_j. Second, it selects subset F'(t_j) of $F(t_j)$ with minimum fault sizes, and if all faults in F'(t_j) have a matching larger fault in $FL_{detected}$, t_j is dropped from T'. Else when one or more new faults are present in F'(t_j), these additionally detected faults are found, and the corresponding larger faults are removed from $FL_{detected}$. Then the additionally detected faults are added to $FL_{detected}$. Then FL is also updated to not include any faults detected by t_j. Once this is done for every test pattern in T, there should be only one fault size in T' for each pair (f, c). So, reversed order fault simulation of $FL_{detected}$ and T' is executed. This entails fault simulation starting with the last pattern in T', and ending with the first pattern in the set. If any simulated pattern does not find any new faults it is removed from the test set.

Algorithm 2 Test Pattern Set Compaction under Variations from [72]

```
INPUT: $T, FL, C$
OUTPUT: Compacted Test set $T'$ for SDFs
 1: Set $j = 0$
 2: Initialize fault lists $FL_{undetected} = FL$, $FL_{detected} =$ empty, $T' = T$
 3: for $j =< |T|$ do
 4:     Perform timing-aware fault simulation to find set of faults $F(t_j)$ in $FL_{undetected}$ detected by $t_j$
 5:     Select subset $(F'(t_j))$ of $F(t_j)$ with minimum fault sizes
 6:     if (All faults in $F'(t_j)$ have a matching (larger) fault in $FL_{detected}$) then
 7:         drop $t_j$ from $T$
 8:     else
 9:         Determine additional detected faults by $t_j$
10:         Remove the corresponding larger faults from $FL_{detected}$
11:         Update $FL_{undetected}$
12:     end if
13: end for
14: Fault simulation of $T'$ and $FL'$ in reversed order as there is now one fault size for each $(f, c)$
```

Figure 3.7 shows the results from Jafarzadeh et al. [72]. They used six benchmark circuits to compare their robust test set with a 10-detect test set, and an M_{rob} test set. The M_{rob} test set is generated by N-detect with $N = M_{rob}$, to match the test set size of the robust test set. The provided results consist of FE as defined in Equation 3.4, and the test set size $TSize$.

Table 3.7: Results from [72]

Circuits	Robust Test Set		M_{rob}-detect Test Set			10-detect Test Set	
	FE	TSize	M_{rob}	FE	TSize	FE	TSize
b12	0.59	704	5	0.58	685	0.59	1337
b14	0.58	2533	6	0.56	2521	0.57	4091
b20	0.56	3367	5	0.54	3125	0.55	6329
b17	0.56	5498	8	0.54	5506	0.55	6870
p45k	0.57	14234	6	0.55	12962	0.56	21487
p100k	0.55	16764	8	0.53	16884	0.54	20989

Figure 3.7 shows a slight improvement in the by Jafarzadeh et al. created FE metric, for the robust test set over the 10-detect test set. This is to be expected because this metric is used in the generation of the robust test set. Furthermore, the robust test set does provide a significant test set size decrease compared to N-detect. The main idea of work from Jafarzadeh et al. [72] is that the test pattern generation specifically tries to target hard-to-detect faults. By searching of the circuit in the circuit set with the lowest FE. In addition, the method allows generating test patterns for SDFs while avoiding slow timing-aware ATPG. This was done by using non-timing-aware ATPG in combination with timing-aware GPU-accelerated fault simulation.

This variation-aware testing method was expanded by Jafarzadeh et al. [75], to also account for voltage variations. Figure 3.11 shows the proposed methodology. First, two circuit sets are generated through static timing analysis for minimal and nominal voltage, as well as the corresponding fault lists. Then, T_{min_unc} is generated based on C_{min} and FL_{min} in a similar way to [72]. Following this, the compaction algorithm from [72] is applied under V_{min} and V_{nom}, and even twice sequentially for both, creating the test sets $T"_{min}$ T_{min}, and $T'min$.

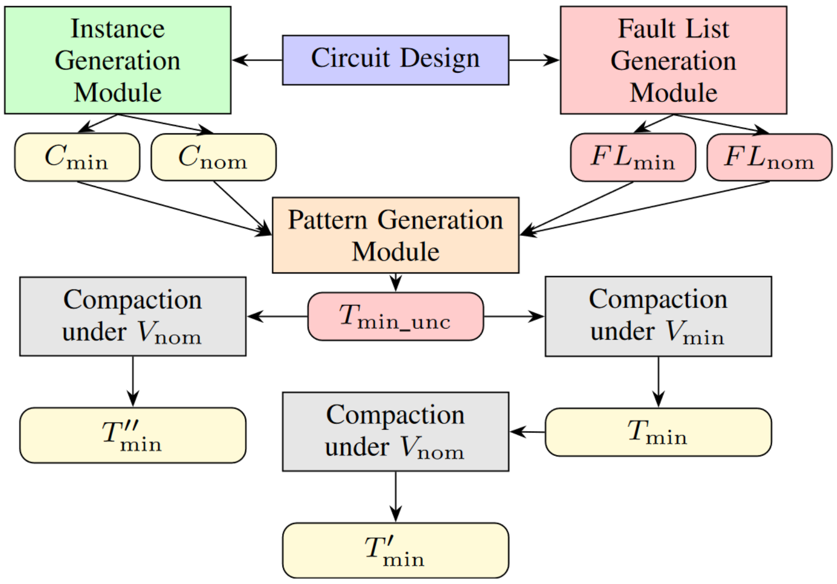

Instance Generation Module
Circuit Design
Fault List Generation Module
C_{min}
C_{nom}
FL_{min}
FL_{nom}
Pattern Generation Module
Compaction under V_{nom}
$T_{\mathrm{min\_unc}}$
Compaction under V_{min}
T''_{min}
Compaction under V_{nom}
T_{min}
T'_{min}

Figure 3.11: Proposed Method from [75]

Table 3.8 shows the results of their approach. They compare the test sets $T"_{min}$ T_{min}, and $T'min$, with the test set T_{nom} based on the approach in [72]. The comparison is done using FE ratio as defined in Equation 3.5. This ratio being not going below one means the patterns generated for V_{min} also enhance test under nominal voltage with respect to FC. An improvement in pattern count is also observed.

$$R(C_V, \delta_i, T_1, T_2) = \frac{FE(C_V, \delta_i, T_1)}{FE(C_V, \delta_i, T_2)} \tag{3.5}$$

Table 3.8: Resulting FE Ratio from [75]

Circuits	Rg(C_{min},T_{min}, T_{nom})		Rg(C_{nom},T'_{min}, T_{nom})		Rg(C_{nom},T''_{min}, T_{nom})	
	Value	**i**	**Value**	**i**	**Value**	**i**
b12	1.07	2	1.10	2	1.13	2
b14	1.02	3	1.31	3	1.33	3
b20	1.03	3	1.24	3	1.25	3
b17	1.05	2	1.32	2	1.33	2
b18	1.03	3	1.38	3	1.39	3
P45k	1.03	3	1.28	3	1.29	3
P100k	1.02	2	1.24	2	1.26	3

Two more works have been published following this promising approach. Jafarzadeh et al. [76] propose a DLBIST method that uses these process variation-aware test patterns. To apply the right test set in the right conditions they add a “DFVS control“ unit to the DLBIST setting, that can set the CUT to desired frequency and voltage. Jafarzadeh et al. [77] again expand on the process variation-aware testing work. They identify the zero temperature coefficient which is the voltage where temperature effect inversion is zero. This means that the effects of temperature-induced variability are minimized. They conclude that test generation at the zero temperature coefficient voltage increases SDF FC, and decreases test set size.

Software-Based Self-Test

SBST is another potential test solution for SDEs. SBST is a method of testing a processor core using instructions, allowing a SBST program to be executed like any other program. Chen et al. [78] compare SBST with the alternatives full scan and logic BIST. They show several advantages of SBST. It restricts itself to the available hardware, so it does not add any extra area to the design. SBST can be run like any other software program enabling at-speed test. Additionally, SBST programs are very flexible allowing in-field application of SBST tests. However, SBST also has its downsides making it a bit impractical for wide adaptation. SBST programs are specifically tailored per core, and it costs significant engineering effort to develop. This is due to the complexity of extracting functional constraints from a processor core. Also, the conversion of patterns to instructions is still actively being researched. A more detailed discussion of the current SBST state-of-the-art can be found in Section 3.5.

Other System Level Test Works

There are several other works on system level test that are worth discussing. The work by Cantoro et al. [79] shows that system level tests targeting SAFs can be modified to target TDFs. This is done by observing which TDFs are excited by the SAF test, and then observing them. Bartolomucci et al. [80] describe a technique that is used to assess small delay defects coverage of software test libraries. This technique enables measuring the quality of software test libraries with regards to small delay defects. However, it does not improve software test library generation. Lastly, there are also system level test works that do not test CPU cores, but rather GPUs [81] [82]. With the growing adoption of artificial intelligence, demand for GPUs and other accelerators is sure to keep increasing in the coming decades [83]. Therefore, demand for system level tests for GPUs is also likely to increase.

3.4.5. Conclusion

To conclude, several software and hardware SDE solutions have been covered. Mitigation through test has the most potential, due to relatively low overhead compared to the other solutions. Moreover, testing targets the root cause of many SDEs, defects. Testing under minimal voltage has been shown to find more hard-to-detect defects causing SDEs. Process variation-aware testing is also an improvement over current testing practice, in terms of FC and test set size. A combination of the two has been proven to be even better. SBST is a way of applying tests that is faster and more flexible than current methods of test application. This is exactly what is needed to tackle SDEs. It is an answer to Adit Singh’s call : ”new low-cost scan tests that can reliably detect timing failures caused by random process variations remain of great interest to industry” [10]. SBST programs are based on scan tests, and are relatively low cost. In combination with low voltage and variation-aware testing it could be exactly what the industry needs. However, it is not yet widely deployed mainly due to the large development effort associated with SBST. Nonetheless, SBST is the SDE solution with the most untapped potential to be a good solution to the issue of SDEs. This thesis will research SBST to unlock more if its potential.

3.5. SBST State-of-the-Art

This section will cover the SBST state-of-the-art. The goal of this section is to discuss existing approach, and existing unsolved challenges. Based on this discussion an avenue for researching SBST is identified. This will be the starting point for the research done in this thesis. There are different types of SBST development methods. Figure 3.12 shows an SBST taxonomy created by Psakaris et al. [84] fifteen years ago. An SBST can be developed by either a functional method, or a structural method. This relates to functional and structural testing, which has been covered already. Structural methods are superior to functional methods, because functional methods are ad hoc and structural methods are based on fault models. Therefore, this thesis will only consider structural SBST methods, which are split up into hierarchical methods and RTL-level methods. Hierarchical methods are defined as *"Such methods focus on a processor's modules one at a time, generating stimuli for each module and then extending those stimuli to the processor level"* [84]. Examples of hierarchical methods are using precomputed stimuli and CATPG. RTL-level methods are *"methods in which the test program generation process exploits structural RTL information along with ISA information to generate instruction sequence templates for justifying and propagating faults of the module under test"* [84]. Examples of RTL methods are pseudorandom test data, which consists of random instructions, and ATPG per component.

However, it is ambiguous that ATPG per component is not a hierarchical method. Additionally, this literature review will show many RTL-level method that make use of CATPG. The methods developed since Psakaris et al. [84] published have almost all used CATPG.

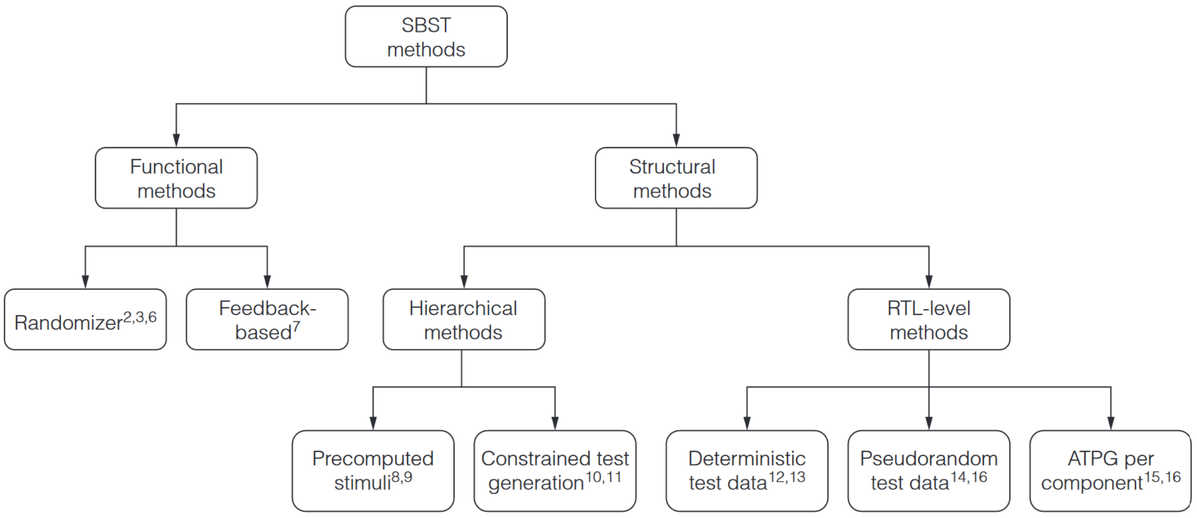


Figure 3.12: Taxonomy of Development Styles for SBST from [84]

The rest of this section is structured as follows. First, Section 3.5.1 will provide an overview of SBST and its various aspects. Then, Section 3.5.2 will discuss the state-of-the-art FCE methods. Section 3.5.3 covers pattern-to-program conversion used by state-of-the-art SBST works. Then, Section 3.5.4 discusses the combination of SBST with DFT and related works. The discussed SBST works will then be discussed in Section 3.5.5. Finally, Section 3.5.6 will conclude this section.

3.5.1. Overview

To enable a structured discussion of the state-of-the-art of a hot topic like SBST, its components need to be explained first. Given that different works focus on varying aspects of SBST, the discussion is organized by its different aspects. Figure 3.5.2 shows the main steps of SBST. First, there is FCE which can be seen as a preparatory step for CATPG. ATPG needs to be constrained (CATPG) to adhere to the functionality of the core-under-test. FCE can be implemented using different approaches for the modelling, extraction, and subsequent formatting of these constraints. This will be elaborated on in Section 3.5.2. CATPG is a very important step but not a step that gets a lot of attention in the literature. This is most likely due to the fact that most ATPG tools are proprietary, so not much can be disclosed about them. The only thing that varies for different works is the method of applying the constraint, but this is heavily tool dependent. Lastly, there is pattern-to-program conversion which turns the patterns generated by ATPG into an SBST test program. This step will be further covered in Section 3.5.3.

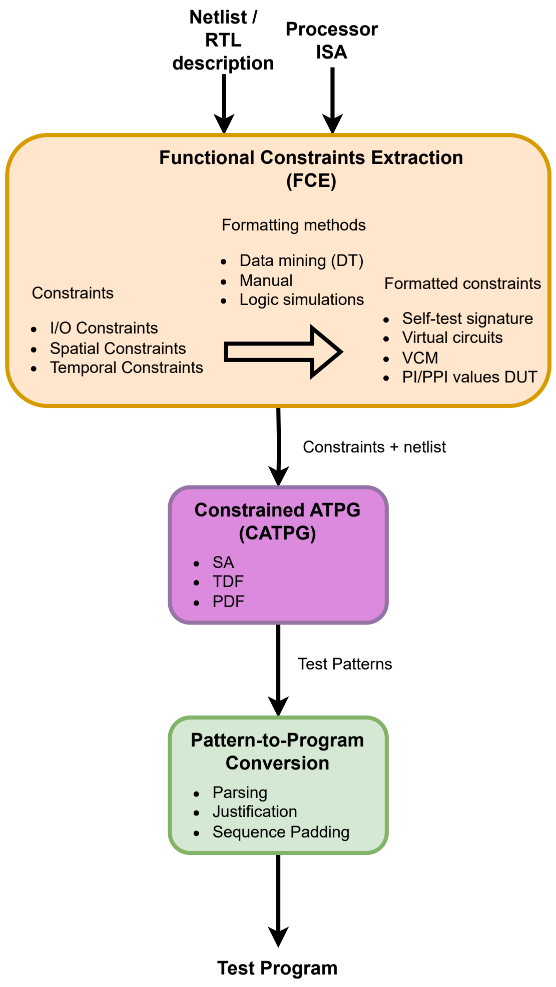


Figure 3.13: An Overview of SBST

3.5.2. Functional Constraints Extraction

FCE is one of the main challenges when developing an SBST. This is due to the ever-increasing complexity of modern processor cores. It is very challenging to automate, and *"ATPG constraints are very hard (if not impossible) to extract by analysing the design"* [85]. It is important that FCE is done well. The better the constraints are that constrain the ATPG, the easier the pattern-to-program conversion is. The ideal case is perfect FCE resulting in the output patterns of the ATPG all being functionally possible, enabling pattern-to-program conversion to perfectly convert the patterns into a test program. However, in practice FCE is never completely perfect. If the constraints are very strict, pattern-to-program conversion will be able to implement all patterns as instructions, at the cost of a lower FC due to heavily restricted CATPG. On the other hand, if the constraints are very loose, the fault coverage of the patterns generated by CATPG will be high, but a ton of patterns might not be functionally possible. Resulting in a lot of FC loss in this step. For all of these reasons improving FCE is extremely important. Works with their focus or their partial focus on the FCE process will be discussed below.

Expanded Instructions

In 2010 Zhang et al. [21] presented a template-based SBST method called automatic test instruction generation (ATIG). The ATIG Zhang et al. propose is based on expanded instructions. The additional information an expanded instruction contains is: time, instruction type, circuit states, operand values, instruction results, in/out flags, and the program counter. Figure 3.14 shows the ATIG framework. As can been simulation-enabled data mining is executed to obtain mappings between expanded instructions and signal values. These mappings are used to generate instruction-level constraints. These constraints are then converted into virtual circuits, and applied to ATPG effectively creating CATPG functionality. The test patterns generated by the CATPG are then converted into instructions, and a program is assembled. The mappings are also used in this process. The main contribution of Zhang et al. is the automatic FCE they developed. The discussed work was validated on the PARWAN [25] processor and targeted stuck-at faults (SAFs).

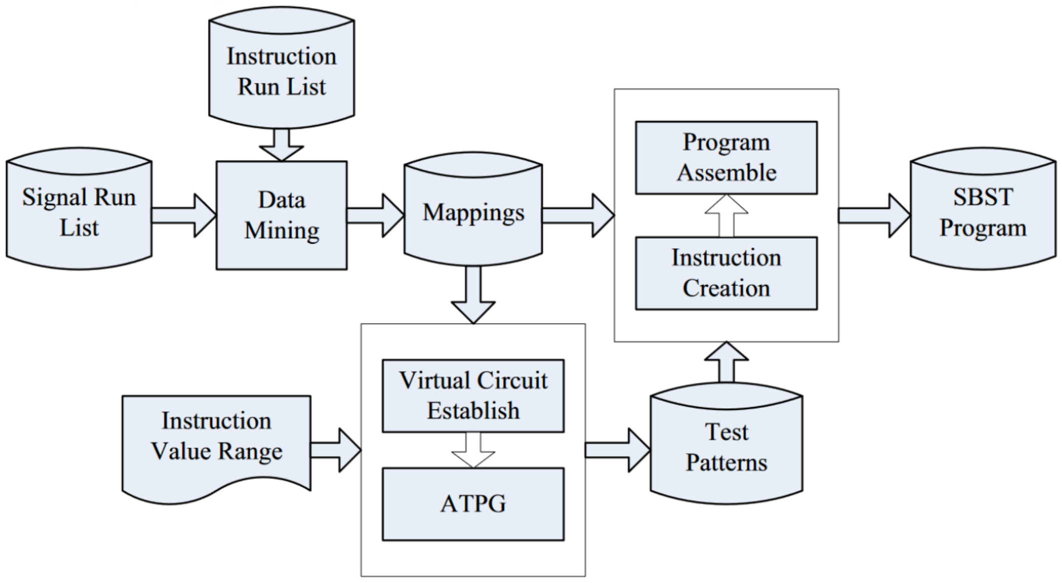


Figure 3.14: ATIG Framework from [21]

Executing-Trace-Based Constraint Extraction

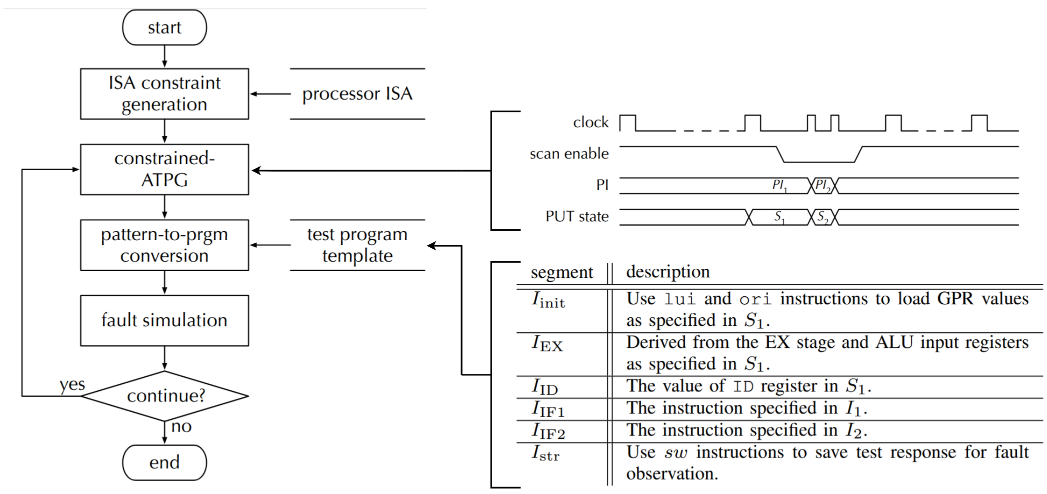
In 2013, Zhang et al. [22] proposed an advanced ATIG method based on executing-trace-based constraint extraction. Unlike the previous template-driven approach, this method leverages instruction-level simulators to collect rich execution traces, which are then used to derive expanded instruction representations containing elements such as operands, results, flags, addresses, and instruction types. These expanded instruction representations form direct mappings to the ports of processor components, simplifying the extraction of functional constraints. A decision tree algorithm with directed random checking is applied to these traces to systematically map signal-to-instruction relationships. These extracted mappings enable CATPG to generate functionally valid patterns, even for hard-to-reach hidden control logic. For sequential hidden control logic components like branch predictors, Zhang et al. introduce a method that embeds structural test patterns into functional routines derived from extended finite state machines, enhancing sequential depth and observability. Validation on the miniMIPS [86] processor demonstrated that the proposed method achieves comparable FC to full-scan, with an SBST program.

3.5.3. Pattern-to-Program Conversion

Pattern-to-program conversion is a vital step in SBST program generation. It is the process of turning the output of the CATPG, test patterns, into a sequence of instructions that forms the final SBST test program. This process is dependent on the constraints that were used in the ATPG. The more complete the constraints are, the more efficient the pattern-to-program conversion is. Pattern-to-program conversion efficiency measures how many of the test patterns are successfully converted, into instruction sequences that detect the same fault as the test patterns. A relatively easy task done in almost all pattern-to-program conversions is writing all general purpose register values to memory, to make faults observable. The biggest challenge of pattern-to-program is finding instructions that can set the core to specific FSSs. Different approaches to this and pattern-to-program conversion in general will be discussed in this section.

Test Program Template

Chen et al. [8] developed a test program template to aid in the pattern-to-instruction conversion. Figure 3.15 shows the SBST test program generation proposed by Chen et al. [8]. The constraint generation in this work was done manually, and therefore the constraints are very loose. Meaning that only obvious constraints were applied, like the instruction input of the core being constrained to the ISA, and the ALU opcode register adhering to legal ALU opcodes. This could lead to lower FC when compared to stricter constraints, but they also implemented a feedback loop from fault simulation back to CATPG which increases FC. Next, they perform CATPG creating LoC full-scan patterns. As discussed in the background LoC are functional behaviour and thus easy to convert to instructions. Then, in the pattern-to-program conversion the test program template is used. The template consists of six segments, consisting of a sequence of instructions. I_{init} initializes the general purpose registers with The values specified in S_1. I_{EX} is also based on the values in S_1, specifically the values in Ex stage, and ALU input registers. I_{ID} is the instruction that is found in the ID register in S_1. I_{IF1}, and I_{IF2} correspond to the instructions specified in I_1 and I_2. Finally, I_{str} is an instruction sequence that writes all general purpose registers to memory for fault observation. The work has been validated on a 5-stage MIPS32 core, and achieved 97.82% TDF FC.



segment	description
I_{init}	Use `lui` and `ori` instructions to load GPR values as specified in S_1.
I_{EX}	Derived from the EX stage and ALU input registers as specified in S_1.
I_{ID}	The value of `ID` register in S_1.
I_{IF1}	The instruction specified in I_1.
I_{IF2}	The instruction specified in I_2.
I_{str}	Use sw instructions to save test response for fault observation.

Figure 3.15: Proposed SBST Test Program Generation from [8]

BMC and VCM

In the work of Riefert et al. [87] [23] they present *"the first fully automated approach to functional microprocessor test"* based on Bounded Model Checking BMC. BMC is used to generate instruction sequences. By modelling the problem of achieving a system state from an initial state through a sequence of instructions as a BMC problem, it can be solved using a BMC solver. The aforementioned work also introduces a new concept: Validity Checker Module (VCM). A VCM is a type of virtual circuit that has the functional constraints imposed by the PUT encoded in it. A VCM allows DFT engineers to easily specify constraints for patterns to adhere to functional inputs. Furthermore, it enables proving the untestability of faults under the specified requirements. The work of Riefert et al. targets Small-Delay Faults and was done on the miniMIPS core [86]. Their main focus was on pattern-to-program conversion and the constraining of ATPG.

Faller et al. [13] also used BMC, and a VCM. However, they modified the VCM to enable faster development of various RISC-V processor families. This was achieved by creating generic constraints, and adding a mapping layer able to map processors with the RISC-V ISA onto the generic constraints.

Figure 3.16 shows that the mapping layer applies name mapping and bus mapping. The developed SBST specifically targets the ALU and register file of the core-under-test, and was validation on two different RISC-V cores. The smallest of the two had about half the area of the larger one, the achieved FC for the former was 82% and the latter 45.5%. Compared to other works the achieved FC is not high. Because the focus of this work was developing an easy to develop generic SBST generation method for RISC-V cores.

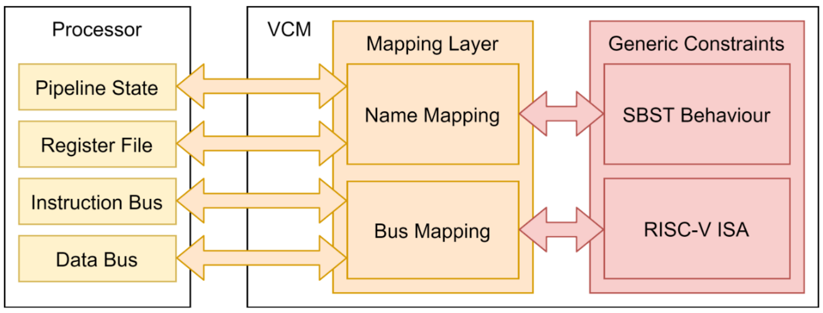


Figure 3.16: Interaction Between Processor and VCM from [13]

Reinforcement-learning-based SBST

Chen et al. [88] have proposed a reinforcement learning (RL) based SBST generation method. Chen et al. specifically exploit RL to tackle the issue of finding a sequence of instructions that can put the PUT in fault-sensitizing states (FSSs). The agent (in this case the SBST program generator) receives feedback based on simulation traces which indicate that a FSS has been reached. Finding instructions that can set a PUT to a specific FSS is the main challenge of pattern-to-program conversion. Therefore, main priority of their work lies in improving pattern-to-program conversion. Figure 3.17 the entire SBST program generation framework. There is a feedback loop from test program generation and fault simulation back to CATPG. When the FC is lower than desired CATPG is done again but with extra constraints extracted from the previous iteration’s traces. The FCE that was applied here, simply extracts the registers that are ’0’ or ’1’ more than 97.5% of the time in the collected traces as constraints for the CATPG. The developed SBST program targets TDFs, and a total FC of 94.94% was achieved on the Antares core [89].

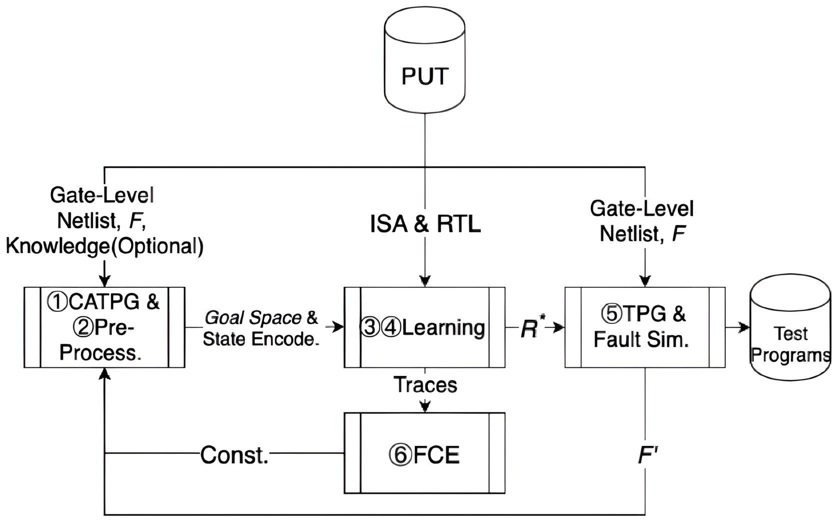


Figure 3.17: SBST Program Generation from [88]

BDD-based Justification Engine

The work of Cheng et al. [11] presents an SBST generation method that uses a BDD-based justification engine. A binary decision diagram (BDD) is an abstraction of a boolean function, and thus can be used to represent the functionality of digital circuits. A justification engine can be used to justify a sequence of instructions that can create, if functionally possible, a FSS provided by CATPG. The SBST generation method by Cheng et al. employs a hybrid pattern-to-program conversion method. It is hybrid because it uses a test program template (the same as Chen et al. [8]) and a justification engine. Figure 3.18 shows the proposed SBST program generation flow. As can be seen Cheng et al. also add a tailor-made test program for the register files. Furthermore, BDDs are used to generate the so called 'd-sequence'. Referring back to Figure 3.15, the 'd-sequence' consists of the segments I_{init}, I_{EX}, and I_{ID}, which are used to get the state in the FSS S_1. The essence of their work is research on a novel method of pattern-to-program conversion. This method was validated on a 5-stage RISC-V in-order processor, and tests for TDFs. The achieved FC is 90.44 %.

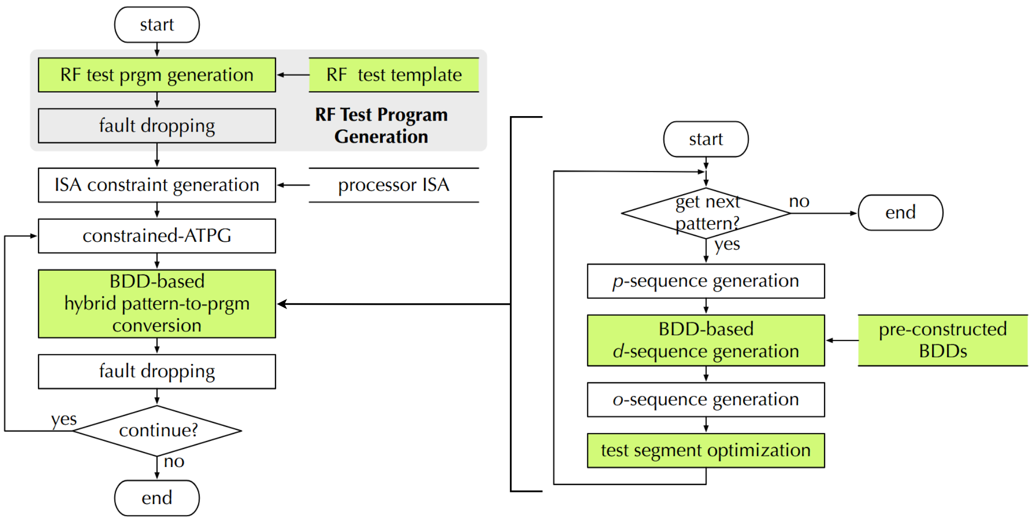


Figure 3.18: Proposed Test Program Generation and Pattern-to-Program Conversion Flows from [11]

Branch-Aware

Kuo and Huang [12] introduce a branch-aware SBST program generation method that addresses the challenge of incorporating branch instructions in SBST. Other template-based approaches struggle with the non-linear control flow introduced by branch instructions, especially in processors with branch prediction mechanisms. The novelty in this work is a method that considers both the actual and predicted outcomes of branch instructions to guarantee correct instruction execution order. The generation flow is very similar to the one shown in Figure 3.18. It starts with CATPG, followed by the derivation of preloading (P), detection (D), and observation (O) sequences. However, the difference is that after D-sequence generation, if it contains branch instructions D is rearranged. The rearrangement algorithm accounts for all four possible outcomes of branch prediction, true or false positives and negatives, using a state-aware insertion of NOPs and jump instructions to keep the intended execution path. Furthermore, the work introduces NOP block recycling, which minimizes test program size by reusing instruction slots left behind after branch-related rearrangement. An optimized test template is also proposed, reducing overhead by only loading and observing modified registers. The method was validated on a five-stage RISC-V core with branch prediction and achieved 90.68% TDF FC, rising to 92.25% with a register file-specific template (from [11]). The test program size was reduced by 76% compared to the baseline template approach. This work demonstrates that making test program generation branch-aware significantly enhances both FC and efficiency.

PDF-based SBST

The first and to this day the only work to target path delay faults (PDFs) with SBST is by Anghel et al. [14]. Anghel et al. propose a systematic method to develop SBST programs for PDFs. The proposed methodology also includes identification of functionally untestable faults. This approach was tested on the open-source RISC-V processor core Ri5cy. The main focus of this work was creating an SBST program that targets the PDF model.

3.5.4. SBST Combined with DFT

The SBST works discussed thus far have all adhered to functional constraints imposed by the CUT and the instruction set architecture (ISA). Theoretically these constraints could be bypassed or adjusted by adding DFT hardware to the CUT. This would entail introducing area overhead, and negating one of SBSTs selling points: zero area overhead. However, the main appeal of SBST is that it can be performed at-speed and has high flexibility, making it ideal for in-field testing. Relatively little research has been done on the trade-off of area versus SBST performance or SBST development effort. A short introduction of the works that have combined SBST with some type of DFT hardware addition will follow.

Test Instructions

Lai et al. [24] propose adding DFT hardware that enables SBST to use test instructions. They apply testability analysis to find registers with low controllability or observability. The “testability analysis“ is done by manually analysing the micro-architecture of the core. Then, they add test instructions to increase the controllability and observability of those registers. Like an instruction that moves data from status registers to a general purpose register, and an instruction that does the same thing but the other way around. Furthermore, they add an instruction that drives the exception signals from a general purpose register. Lai et al. found that they were able to decrease SBST test time, while increasing FC to 100%. At the cost of a 4.7% and 1.6% area overhead on the the PARWAN core [25], and the DLX core respectively. Their work targets “path delay faults“, but they do not define them and only mention them once. Due to the fact that Lai et al. report a 100% FC of PDFs it is likely they had a different interpretation of PDFs than what is known today.

Added Observation Points

Nakazato et al. [26] propose a DFT method to enhance SBST by adding observation points. However, they work under the assumption that each register in the core can be set to 0 or 1 by some function. To ensure this assumption holds they add hardware that carries out this function. They manage to increase the template level fault efficiency to 100%, Equation 3.6 shows their definition of template level fault efficiency. Where $\#MT$ is the number of faults detected in the module-under-test by “test generation“ [26], and $\#TP$ is the number of faults detected by the test template. However, they do not mention for what fault model they are generating tests for.

$$FE_T = \frac{\#TP}{\#MT} \tag{3.6}$$

Reaching full template level fault efficiency, comes at an area overhead cost of 26.68% and 13.49% for the SAYEH core and DLX core respectively. Furthermore, the work by Nakazato et al. has some contradictions, it claims: *“A function to initialize the value of each register in the processor to ‘0’ or ‘1’ is added“*. But it also says: *“the proposed method adds only observation points to the original design“*. Additionally, there is no explanation on how these “functions“ are implemented. Another thing missing from the work is any kind of nuance when choosing which observation points to add.

Functional DFT

Irith Pomeranz published a work on a functional DFT [27]. This work is more theoretical than the previous two discussed works. It was not tested in combination with an SBST program, but SBST is the context in which the idea for the functional DFT can be used. Therefore, there are no area results, only fault coverage results. In short, the idea is to add multiplexers in front of specific flip-flops. These multiplexers swap state variables (flip-flop content) so that this swap propagates a fault towards an observation point in case of a faulty circuit, and in case of a fault-free circuit the state does not change. This DFT requires a very specific situation to be effective. Therefore, Pomeranz ALSO outlines a procedure to find out for which combination of sequential elements inserting the DFT is effective. The approach is based on the observation that faulty circuits have different states than fault-free circuits.

3.5.5. Comparison

A comparison of a selection of the discussed SBST works is presented in Table 3.9. The considered FCE methods are ’test engineer’, ’data mining’, and ’logic simulation’. Data mining has been previously explained in this chapter. ’test engineer’ means the constraint are manually extracted by analysing the design, and logic simulation means they are extracted by analysing logic simulation. As can be seen most works still do FCE manually because their focus is on pattern-to-program conversion, and automatic FCE is not easy to develop. In terms of constraint formats the most used in simply constraining the (pseudo) primary inputs of the PUT with the built-in functionality of the ATPG tool. When looking at fault models a subtle shift towards the more thorough delay fault models can be observed. The conversion methods used by the various works differs a lot, and no trend over time can be established. Most of these methods have been previously discussed. Except for parser mapping, this means that values from test patterns are mapped to registers and inputs and test engineers manually find fitting instructions. Lastly, a very clear shift in the ISA of the PUT can be observed. Recent works all use the most modern of the three ISAs, namely RISC-V.

Table 3.9: Comparison of Various SBST Works

SBST Work		2001 [78]	2010 [21]	2019 [88]	2021 [8]	2023 [13]	2023 [11]	2023 [14]	2024 [12]
FCE Method	Test Engineer	✓		✓	✓	✓	✓		✓
	Data Mining		✓						
	Logic Simulation			✓				✓	
Constraint Format	Signature	✓							
	VCM					✓			
	Virtual circuits		✓						
	PI/PPI Values			✓	✓		✓	✓	✓
Fault Model	SAF	✓	✓			✓			
	TDF			✓	✓		✓		✓
	PDF							✓	
Conversion Method	Parser Mapping		✓					✓	
	Justification						✓		
	Padding				✓				✓
	Included in CATPG	✓		✓		✓			
ISA	PARWAN	✓	✓						
	MIPS32			✓	✓				
	RISC-V					✓	✓	✓	✓

3.5.6. Conclusion

In conclusion, state-of-the-art research on SBST focuses mainly on two sub-problems: pattern-to-program conversion and FCE. Improving these two aspects indirectly increases FC of the SBST program, which is the goal of most of the research. However, this is not the main focus for the work from Faller et al. [13]. They try to make SBST development easier (for RISC-V cores). This might be as important as improving FC, as the high development effort is one of the largest reasons SBST is not widely adopted. Furthermore, none of the recent state-of-the-art works consider adding extra hardware to increase FC or ease development efforts. They limit themselves to the hardware already available on the core, DFT overhead on processor cores is considered the norm [28]. Furthermore, the works that did combine SBST with a form of DFT, are very outdated. Moreover, they do not consider the area vs FC trade-off or how a DFT could decrease SBST development effort.

Recently, the target fault model of SBST works has shifted from SAFs to TDFs. This more effectively targets SDEs because SAFs fall short in modelling the timing issues that are causing SDEs. However, it can also be observed that FC results are lower for TDFs than for SAFs. This is likely due to the fact that the TDF model is more complicated, and thus harder to create test patterns for. To potentially reach similar TDF coverage with SBST as with full scan, test pattern generation needs to be assisted by increasing the testability of the core. A low cost DFT addition to complement SBST programs would enable high quality in-field testing, and could potentially replace full scan completely. This would justify the area cost of the DFT. Therefore, this work explores a hardware DFT addition to complement SBST. The goal of this addition is to increase FC at minimal area overhead.

4 Software-Based Self-Test

This chapter covers the SBST framework developed in this thesis. Section 4.1 discusses the implementation of the SBST program generation. This includes the PUT, synthesis process, FCE, constraint generation, CATPG, and pattern-to-program conversion. Then, Section 4.2 presents the experimental setup. The setup consists of the test pattern file generation, and fault simulation. Section 4.3 reports the results of the synthesis, CATPG, and SBST. These results are discussed in Section 4.4. Finally, the chapter is concluded in Section 4.5

4.1. Implementation: SBST Program Generation

The proposed SBST program generation framework is developed for the CV32E40P core [90], and consists of many components. Therefore, before going into detail, a high-level overview of the framework will now be presented, covering the large components and their inter operation. Figure 4.1 shows the high-level overview of the proposed framework.

There are three main inputs to the framework, namely design constraints, a technology cell library, and the core's behavioural design. The design constraints and technology cell library are necessary for synthesis. Synthesis processes these inputs, together with the core's behavioural design, and produces a gate-level netlist that will represent the core in the rest of the flow. Except for FCE, FCE also uses the core behavioural design and generates a scan cell configuration and constraint files. The scan cell configuration and the netlist are used for scan chain insertion. Then, the scan cell report that specifies, which registers are connected to which scan chains and scan cells is processed by the constraint command generation. The constraint command generation, generates TCL procedures with commands in them that add the right constraints to the correct scan cells. During the CATPG process, before ATPG is performed, the aforementioned commands are used to apply constraints to the scan cells. The CATPG then produces test patterns that adhere to the constraints. The ATPG tool used in this thesis is Tessent by Siemens. These patterns are then processed by the pattern-to-program conversion, which uses a test program template inspired by [8]. The pattern-to-program conversion then outputs a sequence of instructions which make up the test program. The rest of this section will discuss the core-under-test, synthesis, FCE, constraint application, scan chain insertion, CATPG, and pattern-to-program conversion.

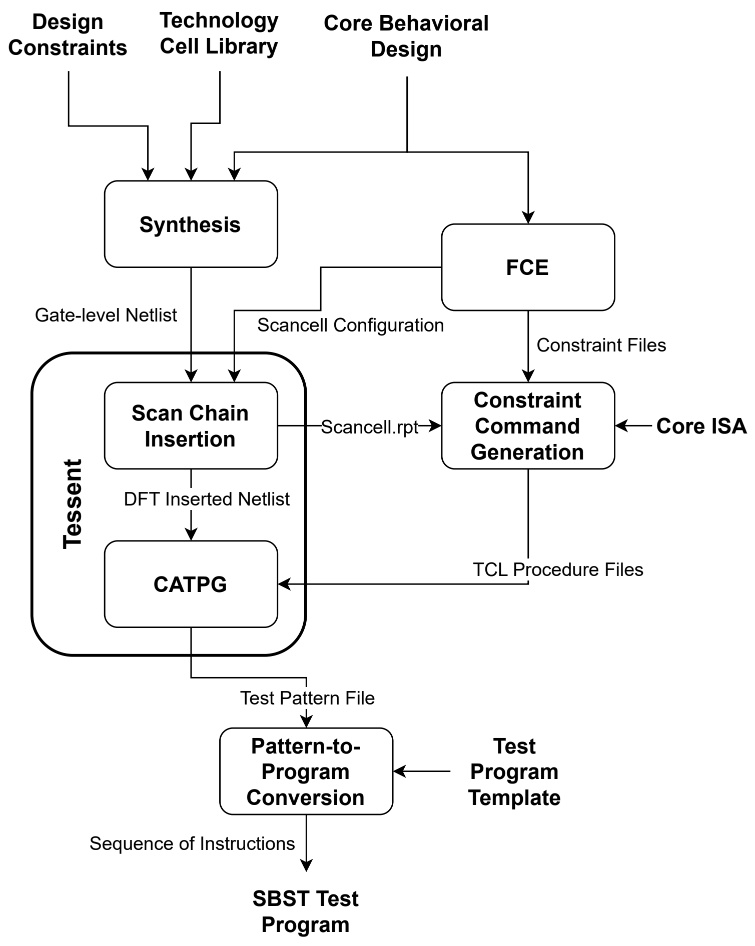


Figure 4.1: Proposed SBST Program Generation Framework

4.1.1. CV32E40P Core

The core for which the SBST program generation framework is developed is the CV32E40P core, formerly known as RI5CY, by ETH Zurich [90]. This core is chosen because it uses the RISC-V instruction set, which is a widely used [33] modern instruction set. Additionally, compared to other SBST works the core is relatively large, and it can execute multiplication. The latter is important because in Chapter 3 it is established that SDEs manifest in functional units. ETH Zurich has also developed a single-core microcontroller architecture around the CV32E40P called Pulpissimo [91]. Figure 4.2 shows the Pulpissimo architecture. It adds peripherals, a debug unit, a clock generator, JTAG, memory, and a possible hardware processing engine (HWPE). However, for the purpose of this thesis much of this hardware is not used and thus excluded. The scope of the testing is the core, so components such as a clock and reset generator, a timer, and the debug unit are removed. SBST involves testing a processor core, in this case the CV32E40P. For this purpose some data memory (RAM) is needed to save results and make the faults observable. By writing data to the data memory during an SBST program, the data memory can be inspected at a later stage to see if any faults occurred. An SBST program, similar to any other program, consists of a sequence of instructions. Therefore, program memory (ROM) is also necessary to read the program from. Figure 4.2 shows the simplification that follows from this, outlined by the red dotted lines. More details on how the memory is implemented can be found in Section 4.2. For now, it is important to understand that the core is utilized within the program generation framework, assuming that both data memory and instruction memory will be available when the SBST program runs.

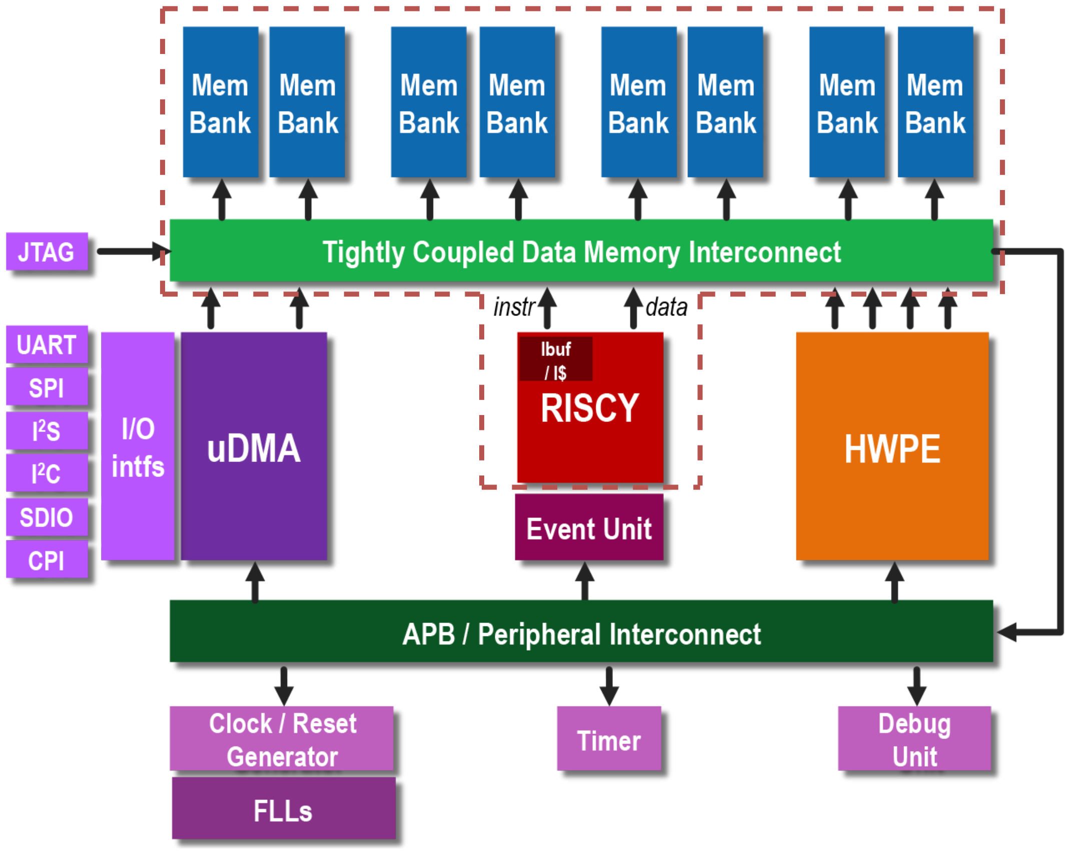


Figure 4.2: Simplification of the Pulpissimo Architecture [92]

Figure 4.3 shows the CV32E40P core. CV32E40P is an open-source 32-bit in-order RISC-V core with 4 pipeline stages, described in SystemVerilog [93]. It is an implementation of the RV32IMC_Zicsr ISA, which includes RISC-V 32-bit integer, multiplication, compressed, and CSR instructions. Optionally, the hardware processing engine can be used to add RV32F instructions, enabling floating point operations. As can be seen in Figure 4.3, the four pipeline stages are instruction fetch (IF), instruction decode (ID), execute (EX), and write back (WB). The instruction decode stage has 32 general purpose registers (GPRs), a decoder, and a controller. The execution stage consists of compute units, and control status registers (CSR). The main inputs of the core are indicated on the left and right as `rdata_i`, these are the instruction data and data from the data memory respectively. These inputs provide data from the location indicated by the core through the `addr_o` output, also on the left and right hand side of Figure 4.3. Lastly, there is the output `wdata_o`, used to write to data memory.

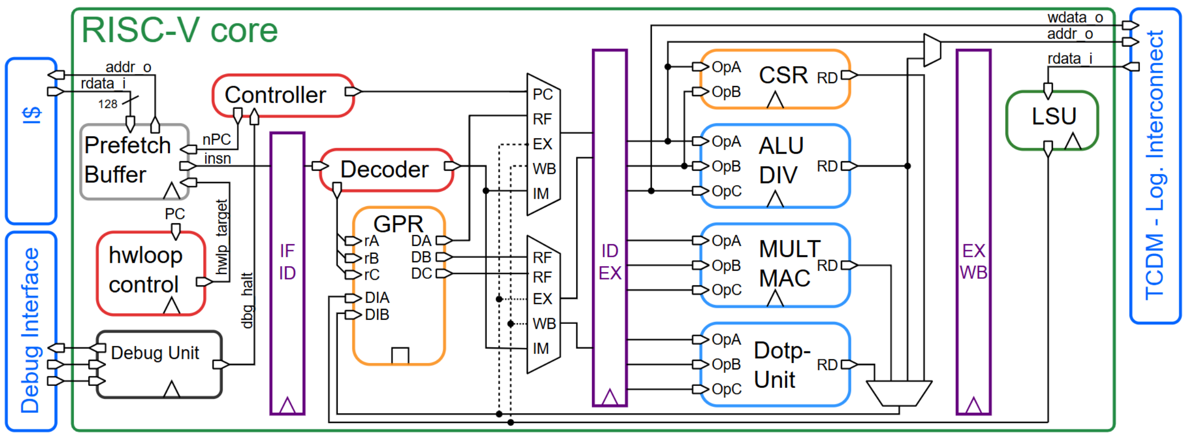


Figure 4.3: CV32E40P Diagram from [93]

4.1.2. Synthesis

The CV32E40P core covered in the previous section is synthesized into a gate-level netlist, using a 40nm technology, so CATPG can be performed on it, and so scan chains can be inserted. The inputs of the synthesis are design constraint files, a technology cell library, and the core RTL design. The design constraint files are used in the synthesis process to set timing goals like the maximum length of the critical path, and setup or hold constraints. The design constraints also set analog constraints like the capacitance of an input, which is key when planning on actually manufacturing a chip at a foundry. This information is used to optimize the synthesis according to these constraints. However, in this thesis synthesis is done to generate a gate-level netlist for ATPG targeting gate-level fault models. Therefore, the specifics of the design constraints files are not critical. For SDF these constraint files would matter because timing-aware ATPG is necessary, but for TDFs this is not the case. So for TDFs and SAFs changing the constraint files might cause some gate level or wiring changes, but the results will not significantly change.

The technology cell library that is used is the 40nm technology of TSMC. Genus, the synthesis tool used, combines the technology, RTL design, and design constraints into a netlist. The `synthesis.tcl` script is set up for this purpose. The clock gating module in the CV32E40P core had to be replaced for synthesis with a clock gating cell of the specific technology used. Clock gating is a power management technique used in digital design, that removes the clock signal when it is not being used. This is done by latching the clock with an enable signal. HDL could describe clock gating functionally, but only technology-specific clock gating cells can implement it safely and efficiently in silicon. These cells are tuned to the electrical, timing, and physical characteristics of the specific node, so that any potential clock glitches are avoided.

4.1.3. Functional Constraints Extraction

The FCE method applied in this work is manual extraction by a test engineer. The choice for this method will first be justified, then the method is explained. The SBST state-of-the-art has shown current works either use manual FCE or (semi) automated FCE enabled through logic simulation and data mining. It can be observed that only the works with their main focus being FCE have gone through the efforts of automating it. This is plainly because setting up automatic FCE requires tremendous effort, and is not worth it if this is not the focus of the research. This is exactly the reason why a form of manual FCE is used in this work.

FCE is one of the most challenging aspects of SBST, and requires great effort to develop. Due to the focus of this work lying elsewhere, this process has been simplified as much as possible. Before diving into the simplification it is key to formulate the goal of FCE. The constraints that FCE extracts should ensure that the patterns produced by CATPG are functionally possible. This means that the instruction input of the core should be restricted to the instructions of the implemented ISA (RV32IMC). Additionally, the CATPG uses scan-based patterns. So, the internal FSSs specified by these patterns should also be functionally possible. This is the main challenge of FCE. The input constraints are straightforward to implement, but internal states of the core are very hard to constrain. This is due to the complexity of modern processor cores.

The simplification of the FCE is applied for the FSS constraints. The issue is that the CV32E40P, when applying full-scan scan insertion, has 2130 scan cells. So, when considering full-scan patterns the FSS has the size of 2130 bits. Therefore, it is not feasible to do a complete manual FCE. The work by Chen et al. [8] identifies only *"apparent register value constraints"* to reduce development efforts. With apparent register value constraints they mean registers that save instructions or ALU opcodes. In a way this is also done in this thesis. However, as opposed to full-scan (used by Chen et al. [8]), this work uses partial scan. This is a form of simplification. If there are no constraints on certain registers, and they can not be easily controlled, they are removed from the scan configuration used in the CATPG process. Effectively avoiding the situation in which an unconstrained state variable creates a functionally not possible FSSs. Also, the main goal of this work is to add some form of DFT. An estimate about the DFT addition can be made by adding more controllable or observable scan cells to this partial-scan setup. To make a good estimate about the DFT addition, the constraints should be as strict as possible. This will increase the pattern-to-program conversion efficiency, and improve this estimate.

Partial Scan Configuration

The reason for choosing a partial scan configuration has been explained above. The decision-making process for which specific scan cells to include in this partial scan configuration will now follow. The main criterion is that the included scan cells should be relatively easy to extract functional constraints from. The partial scan configuration selection methodology consists of the following main steps:

1. Remove all registers except the registers in the IF stage and ID stage.
2. Remove registers in the control unit, which are not directly driven by an instruction entering the ID stage or take a constant value in functional operation.
3. Remove the performance counter registers.
4. Remove registers that depend only on signals from pipeline stages other than the IF and ID stages.

With the goal of simplifying the FCE process in mind the steps are elaborated below. The first step removes all registers from the EX and WB stages. This is because these are stages further along the pipeline, which makes it harder to trace register states back to instructions. Step two involves removing registers from the control unit in the ID stage. Registers are removed that cannot be directly driven by an instruction entering the ID stage, and which are not constant in functional operation. For example, certain registers only change when in debug mode which will not happen during SBST program execution. Third, the performance counters in the ID stage are removed, as they are driven in a very convoluted way, making it difficult to set proper constraints for them. Lastly, the fourth step removes registers that depend only on signals from the EX and WB stages. It is hard to analyse which instructions set these registers to which values, as the signals driving these registers come from the back of the pipeline. Lastly, it is important to note that general purpose registers 1-9 are reserved for usage to aid in pattern-to-instruction conversion. So, general purpose registers 10-31 are included, as these are easily controllable and observable. Figure 4.1 shows the resulting partial scan configuration, excluding the general purpose registers.

Table 4.1: Registers Included in Partial Scan Configuration

Register Name	Range	Register Name	Range
/id_stage_i/alu_operand_a_ex_o_reg	31:0	/id_stage_i/controller_i/debug_reg_q_reg	0
/id_stage_i/alu_operand_b_ex_o_reg	31:0	/id_stage_i/controller_i/debug_reg_entry_q_reg	0
/id_stage_i/alu_operand_c_ex_o_reg	31:0	/id_stage_i/regfile_waddr_ex_o_reg	4:0
/id_stage_i/mult_operand_a_ex_o_reg	31:0	/id_stage_i/regfile_alu_we_ex_o_reg	0
/id_stage_i/mult_operand_b_ex_o_reg	31:0	/id_stage_i/regfile_we_ex_o_reg	0
/id_stage_i/mult_operand_c_ex_o_reg	31:0	/id_stage_i/prepost_useincr_ex_o_reg	0
/id_stage_i/alu_en_ex_o_reg	0	/id_stage_i/regfile_alu_waddr_ex_o_reg	4:0
/id_stage_i/mult_signed_mode_ex_o_reg	1:0	/id_stage_i/branch_in_ex_o_reg	0
/id_stage_i/mult_en_ex_o_reg	0	/id_stage_i/csr_access_ex_o_reg	0
/id_stage_i/mult_operator_ex_o_reg	2:1	/id_stage_i/csr_op_ex_o_reg	1:0
/id_stage_i/data_we_ex_o_reg	0	/id_stage_i/data_sign_ext_ex_o_reg	0
/id_stage_i/controller_i/jump_done_q_reg	0	/id_stage_i/data_req_ex_o_reg	0
/id_stage_i/controller_i/illegal_insn_q_reg	0	/id_stage_i/data_type_ex_o_reg	1:0
/if_stage_i/instr_rdata_id_o_reg	31:2	/if_stage_i/instr_valid_id_o_reg	0
/if_stage_i/illegal_c_insn_id_o_reg	0	/if_stage_i/is_compressed_id_o_reg	0

Manual FCE

As mentioned before, the FCE was done manually. The constraints are modelLed as a finite set of possible circuit states, reachable with one instruction. So, for each instruction entering either the IF or ID stage, the state following from it is established as one of the possible states. In the pattern-to-program conversion a state coming from the CATPG can then be linked to its corresponding instruction. This is done twice, once for the IF stage, and once for the ID stage. These constraints are created manually by analysing the RTL design as well as logic simulations. For example, when an ADD instruction enters the core through simulation it can be seen what values the registers in the IF stage have. In the next cycle, when the ADD instruction enters the ID stage it is decoded by the decoder, which sets the values of several registers in the ID stage. This can easily be viewed by analysing the RTL design and confirmed through simulations.

4.1.4. Constraint Command Generation

The constraint command generation is the process of applying the extracted constraints to the ATPG through Tessent commands. This process depends upon the capabilities and functionality of the ATPG tool that is used. This thesis uses the ATPG tool Tessent by Siemens. Tessent allows setting ATPG constraints, and cell constraints [94]. Setting ATPG constraints restricts ATPG generated patterns to adhere to them. These constraints can restrict specific pins to a value. Alternatively, an ATPG function can be added as an ATPG constraint. An ATPG function can be any logical combination of pins, specified with AND's, OR's, and NOT's. ATPG constraints can be static or dynamic. A static constraint is always applied, in contrast a dynamic constraint is only applied during ATPG patterns. A cell constraint can restrict any scan cell to '0', or '1'. Besides that, it can make specific scan cells only observable or only controllable. Below is a list of the constraint sets created:

- All instruction constraints *dynamic (1)*
- Scan enable constraint *dynamic (0)*
- Cell constraints *static*

The dynamic constraint sets are the all instruction constraints and scan enable constraint. The all instructions constraint set restricts the instruction input of the core to implemented instructions, and the scan enable constraint restricts scan enable to 0. These constraints are dynamic because they only need to be applied during the ATPG cycles, not necessarily during load and shift cycles. Lastly, there are cell constraints that specify if scan cells are observable or controllable. With the exception of the general purpose registers, all of the scan cells in the scan configuration are set to be only controllable and not observable. This will ease pattern-to-program efforts as any flip-flop that is not part of a general purpose register is hard to make observable through instructions.

The state constraints consisting of the constraints on the IF and ID stages are static so that they also hold during load and shift cycles. However, these constraints have not been applied to the CATPG because they resulted in extremely long test pattern generation time and low FC. The effect of this will be a lower pattern conversion rate.

The application of constraints has been automated. Because when the scan configuration changes, the flip-flops will be linked to different scan chains and scan cells. Therefore, the scan cell report can be processed together with the csv files containing the constraints, to create scripts with procedures consisting of Tessent commands that apply the constraints.

4.1.5. Scan Chain Insertion

Scan chain insertion converts a selection of flip-flops in the netlist into scan cells, distributed over a number of scan chains. First, the scan insertion process is set up. This involves setting the maximum number of scan chains, and the maximum length of scan chains. Then, in case of partial scan, the sequential elements that should not be turned into scan cells are removed from the scan list. Setting up the scan insertion process also should get rid of any rule violations. There are certain rules that must be satisfied to ensure proper functionality of scan chains, more on this below. The first step of the scan insertion itself is distributing the memory elements from the scan list into scan chains (`analyze_scan_chains`). Then, the scan chains are inserted using `insert_test_logic -write_in_tsdb On -replace`. This command inserts the scan cells, stitches them up into scan chains, and writes out the scan inserted netlist and a TCD file. The TCD file contains scan configurations and is used during ATPG.

To ensure the scan chains can be reliably used to apply test, Tessent automatically analyses all memory elements to see if they adhere to certain rules. The sequential instances that pass these rules are considered scannable, and are added to the scan list. One of these rule sets is the scannability rules (S rules), which check if the tool can turn off all set and reset lines, and control all clock inputs of sequential cells from primary inputs. If a sequential element does not pass one of these rules it is no longer considered for scan insertion. One of the scannability rules that blocks scan insertion on the CV32E40P core is the S1 scannability rule. The S1 rule states that all the clock inputs (including sets and resets) of each scannable non-scan memory element need to be able to be turned off. This ensures that non-scan elements that could be converted to scan can be controlled to hold their data. The S1 rule violation was resolved with the Tessent command: `set_test_logic -set on -reset on -clock on`. This inserts test logic that can control the set, reset, and clock.

Another set of rules, which are check after insertion are the scan chain trace rules (T rules). To check these rules Tessent injects values at control locations to facilitate scan chain tracing. When tracing fails the corrupt scan cells highlighted and an error occurs. After inserting scan chains in the CV32E40P core, two T rules were violated (T5 and T3). The T5 rule states that no undefined values must be placed on a clock input or a non zero value on a set or reset of a memory element in the scan path. The T3 rule checks if the shift procedure creates a sensitisable path from the scan chain output back to the scan chain input. There were exactly the same amount of violations for both rules, so it was one issue caused violations of both rules. Figure 4.4 shows one of the T5 violations viewed in Tessent visualizer. It can be seen that the latch in the TSMC clock gating cell CKLHQD1BWP is uninitialized at the start of the trace sequence. This causes a T5 violation and indirectly also a T3 violation.

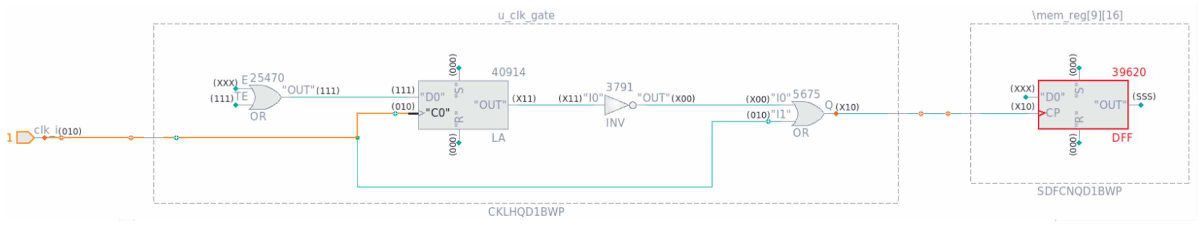


Figure 4.4: T5 Rule Violation in Tessent Visualizer

All clock gating cells available in the TSMC cell library have the same design, but different electrical properties. Additionally, Tessent did not provide a way to solve these violations in their manual or online. Therefore, an alternative clock gating cell is implemented using TSMC cell library primitives. Figure 4.5 shows the original clock gating cell from the library on the left, and on the right the alternative clock gating cell is shown. The OR gate driven by the clock gate enable and test enable inputs is moved behind the latch. The test enable signal passes the latch, going straight into the OR gate together with the uninitialized value from the latch. This ensures there is no undefined signal being outputted from the clock gating cell into any scan cells.

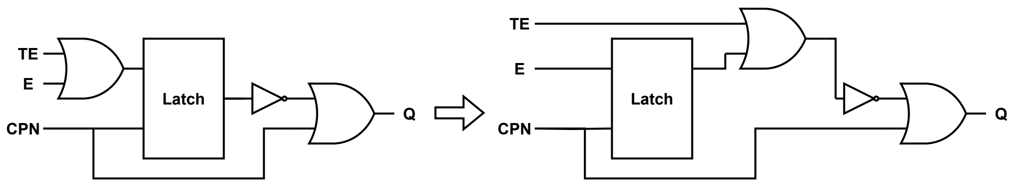


Figure 4.5: T5 Rule Violation Fix

4.1.6. CATPG

CATPG is the generation of test patterns restricted to specified constraints. This step is essential to SBST test program generation, and it is what takes the most computation time. Figure 4.6 shows the entire flow of the script that runs CATPG in Tessent. This flow is equivalent to the "Tessent" block in Figure 4.1. Step one ensures the specified settings are used in the rest of the flow, and creates output folders named according to these settings. The settings include the scan configuration, scan chain length, top module, fault model, and the sequential depth. The scan configuration contains all the sequential elements that should be converted into scan cells, and the scan chain lengths specifies how many scan cells are placed in each scan chain. However, to maximize FC results this is always set to zero. The downside of this is that many extra inputs and outputs are added to the design, but this scan setup is only used in simulation so the amount of inputs and outputs does not matter. The fault model dictates what faults the ATPG target, and the sequential depth sets the maximum amount of cycles a test pattern can be. Step two reads the netlist of the specified top module, specifies the top module to Tessent, and reads the cell library files. To simulate the netlist, Tessent needs behavioural descriptions of the cells from the used technology library. Steps three and four have been discussed in Section 4.1.5. Step five writes out a scan cell report, that is used in step six (see Section 4.1.4) to generate the constraint scripts. Step seven uses the aforementioned scripts to add constraints to the tool. Next, step eight sets the fault type, the sequential depth, and creates patterns. Finally, step 9 writes out any relevant reports.

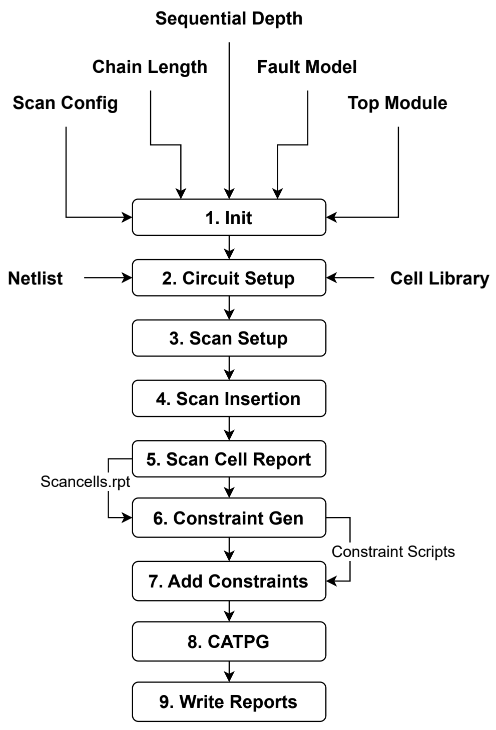


Figure 4.6: Tessent CATPG Script Flow

The output of the CATPG are test pattern files in a STIL [95] format. Figure 4.7 shows the expected behaviour of these test patterns. As can be seen the scan enable is only high during load/unload and shift procedures, ensuring no scan chain usage during sequential and capture cycles. This is due to the dynamic scan enable constraint mentioned in Section 4.1.3. Lastly, the input sequential depth n (shown in Figure 4.7) sets the maximum sequential depth of the test patterns. This decides how many clock sequential cycles there can be within a pattern.

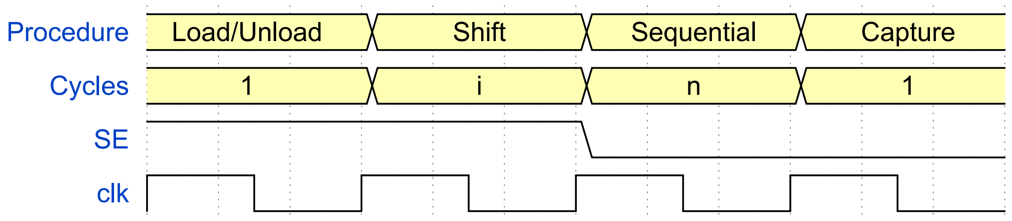


Figure 4.7: CATPG Output Pattern Inspired by Tessent Manual [94]

4.1.7. Pattern-to-Program Conversion

Pattern-to-program conversion processes the test pattern file outputted by CATPG, and creates an SBST program. The implemented pattern-to-program conversion uses a similar test template as used by Chen et al. [8], but it is tailored to partial scan. Figure 4.8 the flow of the pattern-to-program conversion process. The inputs are the ScanCells.rpt and TestPatterns.stil files, based on the information present in these files an SBST program can be generated. The scan cell report contains the information about which flip-flops are linked to which scan cells and scan chains. The test pattern file contains values for each primary input or scan cells, for each cycle in every test pattern. These files are parsed creating a list of instances of the `Pattern` class, and a list of instances of the `ScanChain` class. The `Pattern` class saves all the relevant information of a pattern. The `ScanChain` class consists of one or more instances of the `ScanCell` class, which links a scan cell with its corresponding flip-flop. The two aforementioned lists together are used to create a `Mapper` instance. The `Mapper` class can be used to retrieve values for a specific pattern, including flip-flop values, and instruction input values for specific sequential or capture cycles in the pattern. Then, the `Mapper` instance is used to instantiate the `InstrGen` class. The `InstrGen` class has the method `generate_program()` which is used to generate the SBST program.

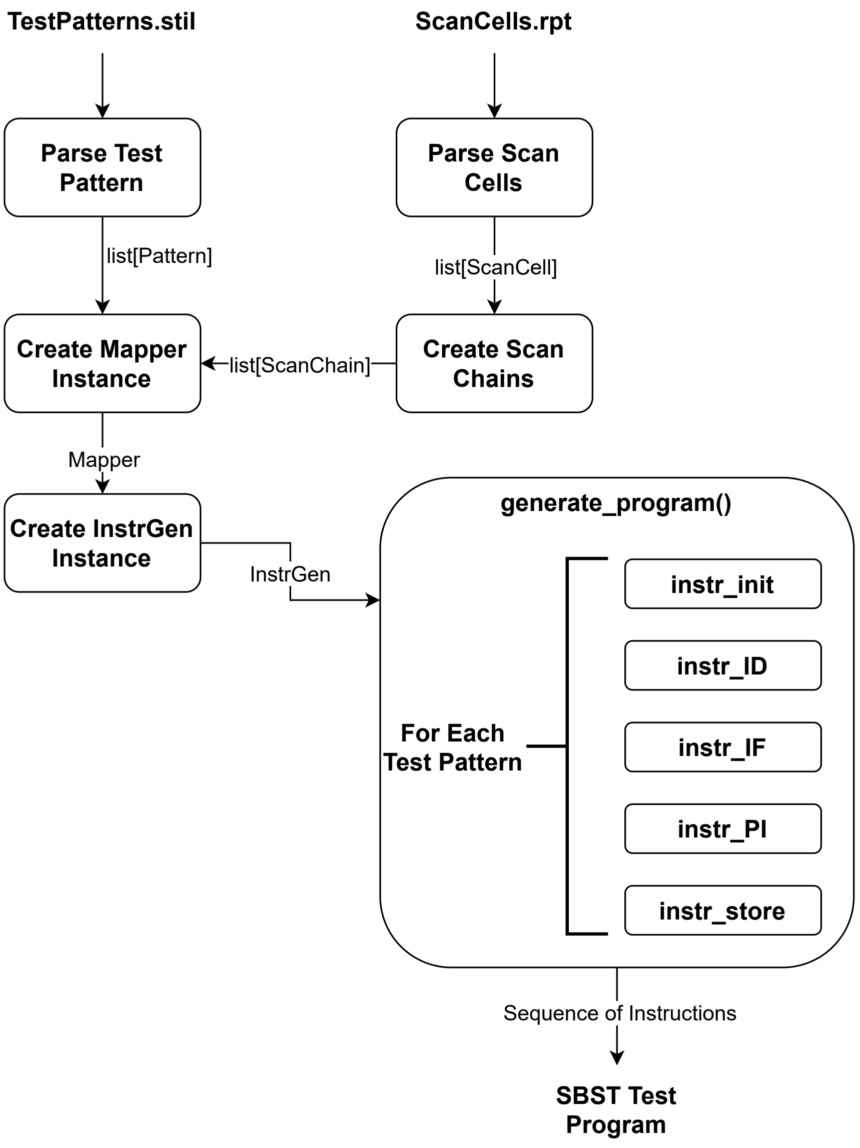


Figure 4.8: Pattern-to-Program Conversion Flow

The `generate_program()` method uses the test template to convert each pattern. The customized test template is shown in Figure 4.9, and as can be seen the template outlines the steps taken in `generate_program()`. The upper three signals represent a scan LoC test pattern, and the lower two signals show how this is applied via SBST. The I_{PI} instructions can be read directly from the test pattern file. The I_{str} instructions save the values of all used general purpose registers (x10-x31). The instruction sequences used to set the core to the FSS (S_1) are I_{init}, I_{ID}, and I_{IF}. The correct values are loaded into the used general purpose registers with the instruction sequence I_{init}. The instruction that sets the IF stage in the right state is I_{IF}. The instruction sequence I_{ID} attempts to set the ID stage to the FSS. However, for this some justification may be necessary.

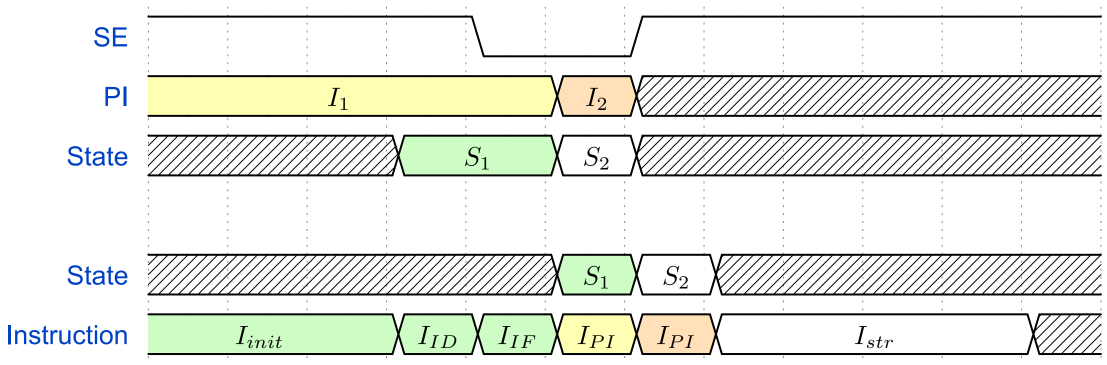


Figure 4.9: Modified Test Program Template based on [11]

Justification of Fault Sensitizing State

Justification of the FSS entails finding instructions that set the core in the right state. As can be seen in Figure 4.9, these instructions consist of the instructions sequences I_{init} I_{ID}, and I_{IF}. The sequence I_{init} consists of instructions loading the GPRs with scan in values from the CATPG test patterns. I_{IF} is the instruction that is loaded into the instruction register of the IF stage in the test patterns. Lastly, the instruction sequence I_{ID} sets the ID stage to the correct FSS. This is a bit more complicated than the other instruction sequences.

The state variables set by I_{ID} include the ALU or multiplier operator, operands, write addresses, and other state variables in the ID state. The FSS is recreated as much as possible, but this process is not perfect. This is due to the fact that the values which the scan cells are set to are not constrained. Applying the manually extracted state constraints resulted in very low fault coverage, so the state constraints have not been applied to the CATPG. Resulting in test patterns with FSSs that are not functionally possible. Therefore, these states have been recreated as much as possible by checking if the ALU or MULT operators are legal. If one of these operators is legal an instruction for the ID stage pipeline can be found. If not, the hamming distances between the states each instruction causes and the FSS specified in the test pattern are computed. The instruction that most closely recreates the FSS is then picked. This is the last instruction of the I_{ID} instruction sequence before the instructions I_{IF}, and I_{PI}. However, there are also some instructions before this in I_{ID}. These instructions load the MULT operands to the correct value is the last instruction in I_{ID} is found to be an ALU instruction, and vice versa. Additionally, some instructions are added that set GPRs to the values of the ALU or MULT operands in the test pattern, depending on what the last instruction in I_{ID} is. This final instruction then used the prepared registers.

For example, if the ALU operator (in the test pattern file) indicates that there is an ADD instruction, the test pattern in parsed for the values of the ALU's a and b operands. Two registers are then set to these values with lui and ori instructions. The same is done to two other registers for the multiplication operands. Then, MULT instruction is applied to set the multiplication operands, followed by an add instruction that sets the ALU operands and operator.

Register File Test Program

The register file of any core has a repeating structure consisting of 32 registers (flip-flops) and their corresponding read and write logic. When a test pattern is found for one register it can be applied iteratively on all registers. So, for this specific module, the manual generation of test patterns is more efficient than ATPG. This has been shown in the work of Cheng et al. [11]. The implemented register file test program is inspired by this work, but it is an improved version. It is shown in Section 4.3 that the implemented RF test program has higher fault coverage than the one described in [11].

There has been one work on specifically testing register files using SBST, by Tuna et al. [96]. The mentioned work applies march tests on a register file through instructions. A march test consists of a sequence of march elements, which itself is a sequence of memory reads and writes. The second column in Table 4.2 shows different march tests and the march elements they apply. Memory reads of

1 and 0 are specified by $r1$ and $r0$ respectively, and writes are specified by $w0$ and $w1$. Furthermore, each march element is applied to all memory entries iteratively before moving on to the next march element. The upwards and downwards arrows before all march elements, specify if the iteration goes from the lowest address to the highest or the highest address to the lowest respectively. The ad hoc march algorithm that is proposed uses a slightly modified notation. The upwards and downwards arrows still indicate the order in which the march sequences are applied to each register. However, for each operation it is also specified if the operation is done on the register of the current iteration (i), or another register.

Table 4.2 shows the different march algorithms that were tried. The march algorithm with the highest fault coverage of the register file, see Section 4.3.4, is chosen and used. The chosen march algorithm is the ad hoc algorithm, tailored for the implementation of the register file of the CV32E40P core.

Table 4.2: March Algorithms Partly from [96]

March Algorithm	**March Sequence**
MATS	$\{ \uparrow (w0); \uparrow (r0,\, w1); \uparrow (r1) \}$
MATS+	$\{ \uparrow (w0); \uparrow (r0,\, w1); \downarrow (r1,\, w0) \}$
MATS++	$\{ \uparrow (w0); \uparrow (r0,\, w1); \downarrow (r1,\, w0,\, r0) \}$
MARCH X	$\{ \uparrow (w0); \uparrow (r0,\, w1); \downarrow (r1); \downarrow (r0) \}$
MARCH C-	$\{ \uparrow (w0); \uparrow (r0,\, w1); \uparrow (r1,\, w0); \downarrow (r0,\, w1); \downarrow (r1,\, w0); \uparrow (r0) \}$
MARCH SR	$\{ \uparrow (w0); \uparrow (r0,\, w1,\, r1,\, w0); \uparrow (r0,\, r0); \uparrow (w1); \downarrow (r1,\, w0,\, r0,\, w1); \downarrow (r1,\, r1) \}$
Ad hoc	$\{ \uparrow (w_i0,\, w_i1,\, r_i1,\, r_{i-1}1,\, w_i0,\, w_i1,\, r_{i+1}0,\, r_i1,\, w_{i-1}0,\, w_i0,\, w_{i+1}0,\, w_i1,\, r_{i+1}0,\, r_{i-1}0,\, r_i1) \uparrow (w_i0,\, w_i0,\, r_i0,\, r_{i-1}0,\, w_i1,\, w_i0,\, r_{i+1}0,\, r_i0,\, w_{i-1}1,\, w_i1,\, w_{i+1}1,\, w_i0,\, r_{i+1}1,\, r_{i-1}1,\, r_i0) \}$

4.2. Experimental Setup

The goal of the experimental setup is to test the FC provided by a generated SBST program. This is done through fault simulation in Tessent. Figure 4.10 shows a high-level overview of the experimental setup. To be able to test the SBST program, it needs to be formatted into a test pattern file that can be read and simulated by Tessent. This section will present the method that is used to generate test pattern files from a sequence of instructions. Then, an elaboration on the fault simulation in Tessent is given.

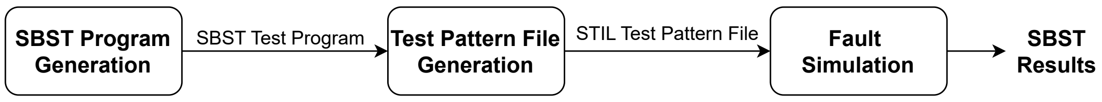


Figure 4.10: Experimental Setup

4.2.1. Test Pattern File Generation

The test pattern file generation consists of a SystemVerilog testbench that runs the sequence of instructions, making up the SBST program, and writes out a test patterns file in the STIL [95] format. Figure 4.11 shows the general fault simulation framework used for SBST fault simulation. Everything shown in the figure except the fault simulation was developed as part of the experimental setup for this thesis. The three main parts of the developed framework are discussed below.

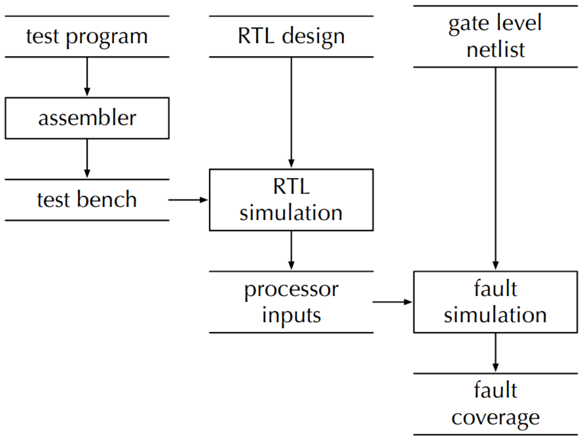


Figure 4.11: SBST Fault Simulation Framework from [78]

Data and Instruction Memory

The SystemVerilog testbench that is implemented mocks the data and instruction memory. The core communicates with both memories via the OBI (Open Bus Interface) protocol. At the start of the testbench the data memory is initialized with random values. The idea of SBST is that the faults are propagated to the data memory, and checked at a later stage. For the sake of generating a test pattern file it is enough to write it to memory, and capture the data write output of the core to be used in the test pattern file. The instruction memory consists of the instructions from the SBST program. The data read address output of the core is not considered, whenever the core requests the next instruction the next instruction from the SBST program is provided. This ignores possible branch instructions that could change the program counter. However, the SBST program could be saved in memory to account for branch instructions. This is outside of the scope of this work, and would not change the results.

Core I/O

Every cycle of the testbench, the core inputs are captured. These input values are used to write a sequential cycle into the test pattern file. When the testbench detects four NOP instructions in a row, it also captures the core outputs. Four NOP instructions in a row are inserted by the SBST program generator to indicate the end of a pattern. The captured core outputs, as well as the captured core inputs of the previous cycle are used to write a capture cycle into the test pattern file.

STILVerify

After the test pattern file has been generated it is verified with the STILVerify software. STILVerify is a verification tool from Siemens for checking the conformiry of STIL files. It ensures that the syntax of the STIL file is correct. Furthermore, STILVerify can be used to convert a STIL file into Verilog testbench files. Simulation of this testbench is the same as the simulation Tessent does internally when running fault simulation of the STIL test pattern file. This enables test engineers to debug and verify STIL-based test development.

4.2.2. Fault Simulation

Fault simulation is executed in Tessent to check which faults are detected by the SBST program. Figure 4.12 presents the flow of the developed Tessent fault simulation script. Steps one and two are the same as in the CATPG script discussed in Section 4.1.6. Step three flattens the design by removing all modular hierarchy, replacing all module instances with their internal logic. Step four adds constraints to the instruction inputs of the core ensuring functional behaviour. Then, step five sets up the simulation by setting the fault model, and sequential depth (to the maximum). Finally, step 6 reads in the patterns and simulates them. The results of step 6 are written out in step 7, including any failures if they occur. Failures are when the good simulation does not match the expected values specified in a test pattern. This effectively checks if the test pattern content is valid.

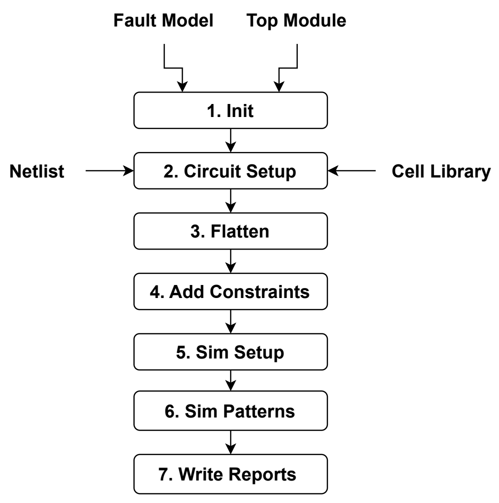


Figure 4.12: Tessent Fault Simulation Script Flow

Implications of Parallel Fault Simulation

A key insight about fault simulation in Tessent is that it can only do parallel fault simulation. There is no option to run sequential fault simulation. Parallel fault simulation means that multiple test patterns are simulated in parallel. Sequential fault simulation would entail that the test patterns are simulated sequentially, carrying over the state the core is left in by the previous test pattern as the starting point of the current test pattern. Running parallel fault simulation means that the core needs to be reset at the start of each test pattern. Otherwise, sequential elements will be uninitialized and store undefined values. This is no issue as the state of the circuit at the start of each test pattern does not contribute to fault sensitization or propagation.

Modified Core for Fault Simulation

Another limitation of Tessent is that it only allows one capture cycle per pattern. A test pattern file with patterns containing more than one capture cycle is syntactically correct according to STILVerify. But when attempting to simulate such a test pattern file in Tessent an error occurs. This is an issue because the test program template ends each pattern by writing out the general purpose registers to memory. Each time a value from a register reaches the core's data write output a capture should take place to detect any corrupt values that were propagated into the register. It is also not possible to spread the memory write instruction over the test patterns that follow, because at the start of each test pattern to core has to be reset.

To tackle this issue an alternative design of the CV32E40P core has been implemented, namely the "CV32E40P_core_faultsim". To create this core used for fault simulation the component `cv32e40p_faultsim_outputs` has been added, as well as 1024 additional primary outputs (`logic [1023:0] faultsim_o`). This component is connected to these additional outputs, and whenever a data write is executed the component copies this data onto the `faultsim_o` output. The component effectively acts as a ring buffer of 1024 bits, allowing 32 32-bit writes. When the buffer is full it will wrap around and overwrite the first data that was written out. Having these extra outputs allows capturing all data that was written out in one capture cycle. This comes at the cost of 1024 extra primary outputs, but as this alternative core design is only used for fault simulation this is negligible. The only information that gets lost is the write addresses that correspond to all the data in the ring buffer. However, the original CATPG patterns also expect to observe these address outputs.

4.3. Results

This section briefly presents the results of the SBST implementation. First, the results from synthesising the CV32E40P [90] in Genus with the TSMC 40nm cell library are shown. Then, the results from the

CATPG are presented, for the scan configuration discussed earlier in this section. Finally, the results of the generation SBST program are shown. These results will be discussed in Section 4.4.

4.3.1. Experiments

Results are presented for a series of experiments, which will be briefly outlined. First, the instruction implementations of the different march algorithms, discussed in Section 4.1.7, are tested through fault simulation. FC results of these experiments is provided. Then, the FC results of the developed SBST programs are shown for both SAFs and TDFs. These are compared with a baseline of full scan FC results. All of those results are given for sequential depths ranging from 2 to 8. For the CATPG and full scan results this means that the pattern generation was run with the corresponding sequential depths. The SBST results for various sequential depths are all based on CATPG patterns of corresponding sequential depths. More sequential depth will allow for achieving higher FC, but it increases test generation time which is also reported. Lastly, the hamming distance estimation discussed in Section 4.1.7 is compared to just picking a random instruction.

4.3.2. Metrics

The main metric used in the results is FC, which is defined in Section 2.5.2. The program size of the SBST programs is reported in megabytes (MB). Lastly, a metric that has not been introduced yet is test coverage loss (TCL). Test coverage loss, defined in Equation 4.1, is the percentage of test coverage that is lost by undetected faults. Test coverage differs from fault coverage as it excludes faults that are proven to be untestable by ATPG patterns. It reflects the effectiveness of the ATPG.

$$Test\ Coverage\ Loss = 1 - \frac{number\ of\ detected\ faults}{number\ of\ total\ faults\ -\ number\ of\ untestable\ faults} \tag{4.1}$$

4.3.3. Synthesis

Table 4.3 shows the results from synthesising the CV32E40P core [90] using the TSMC 40 nm Cell Library. As can be seen, the total area, consisting of the cell and interconnect area, is $38{,}476$ µm^2. The total power consists of the leakage power, internal power, and switching power. The leakage power is the static power consumption of the core. The internal power is the switching power inside cells, and the switching power is the power due to load capacitance. Finally, the maximum clock frequency is reported to be around 103.5 MHz.

Table 4.3: Synthesis Results of CV32E40P [90] with the TSMC 40 nm Cell Library

Metric	**Value**
Cell Count	$12{,}399$
Sequential Elements	$2{,}130$
Cell Area	$25{,}866.77$ µm^2
Net Area	$12{,}256.66$ µm^2
Total Area	$38{,}123.43$ µm^2
Total Power	2.25 mW
Leakage Power	17.33 µW (0.77%)
Internal Power	1.46 mW (64.85%)
Switching Power	0.78 mW (34.38%)
Clock Period	$10{,}000$ ps
Slack	353 ps
Max Clock Frequency	103.5 MHz

4.3.4. Register File Test Programs

The register file test program has been explained in Section 4.1.7. Different march algorithms have been tried, and the ad hoc program is picked because it provides the highest FC. Table 4.4 shows the fault coverage of the register file for SAFs and TDFs. It can be seen that SAF FC is similar for all algorithms. Large differences in FC can be observed when considering the TDF results. Notably, march SR performs better than the other march algorithms, and the ad hoc program outperforms the rest.

Table 4.4: March Tests Fault Coverage of Register File Module

March Algorithm	SAF FC	TDF FC
MARCH C-	72.77%	22.71%
MARCH X	73.56%	22.71%
MATS ++	63.37%	20.66%
MARCH SR	74.33%	47.46%
Ad hoc	74.80%	61.37%

4.3.5. SBST

The CATPG patterns have been converted to SBST programs which are tested by fault simulation. Figure 4.13 shows the fault coverage of the generated SBST programs and the CATPG patterns they are based on. As can be seen the CATPG patterns perform worse than the SBST programs they are converted into for the lowest sequential depths. This is happening due to padding instructions added in the SBST program, and the RF test program that is added. Both of these will detect faults that might not be detected by the original CATPG patterns. Furthermore, when sequential depth increase there is a significant drop between CATPG FC and FC of corresponding SBST programs. This is most likely due to the imperfect pattern-to-program conversion, failing to recreate certain FSSs. Lastly, it is clear that these SBST programs do not come close in FC to the full scan baselines.

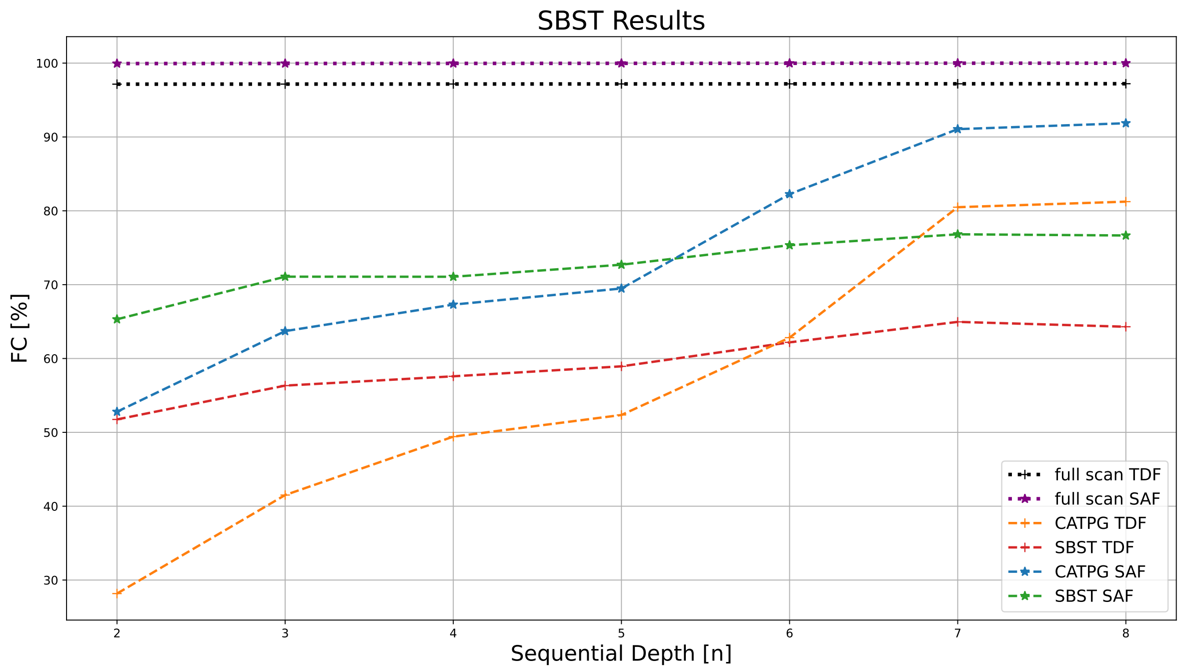


Figure 4.13: SAF and TDF Results of the SBST Programs and CATPG Patterns

To see how the pattern-to-program conversion performs, ideally one would check each CATPG pattern and corresponding SBST instructions. Then, compare if all faults detected by the CATPG pattern are also detected by the instructions. However, due to limits imposed by the ATPG tool this calculation of pattern conversion rate is not possible. Therefore, the detected faults conversion ratio is calculated. This entails the ratio of faults detected by the CATPG patterns that are also detected by the SBST program. This metric does not measure how many of the test patterns are successfully converted, but it does indicate the ratio of faults that are intended to be detected, based on the CATPG patterns, by the SBST program. Figure 4.14 shows the detected faults conversion ratio for various sequential depths and two fault models. As expected there are many patterns likely not converted perfectly. It can be observed that the larger the sequential depth the higher the conversion ratio, this is likely due to the fact that the more cycles a test pattern lasts the less effect the initial FSS has on the capturing of faults. Therefore, the larger the sequential depth the less it matters if the FSS is recreated successfully.

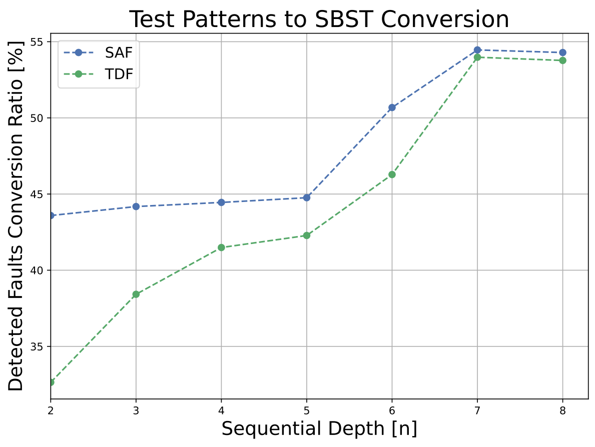


Figure 4.14: Test Pattern Conversion Results

The CATPG runtime is shown in Figure 4.15 for SAF and TDF CATPG. As expected, when the sequential depth starts to increase the runtime goes up. However, from sequential depth five onward the runtime decreases slightly for increasing sequential depth. This is likely caused by the fact that the search space of the CATPG algorithm increased. So, certain faults that the CATPG algorithm timed out on before can be detected now, saving a significant amount of time.

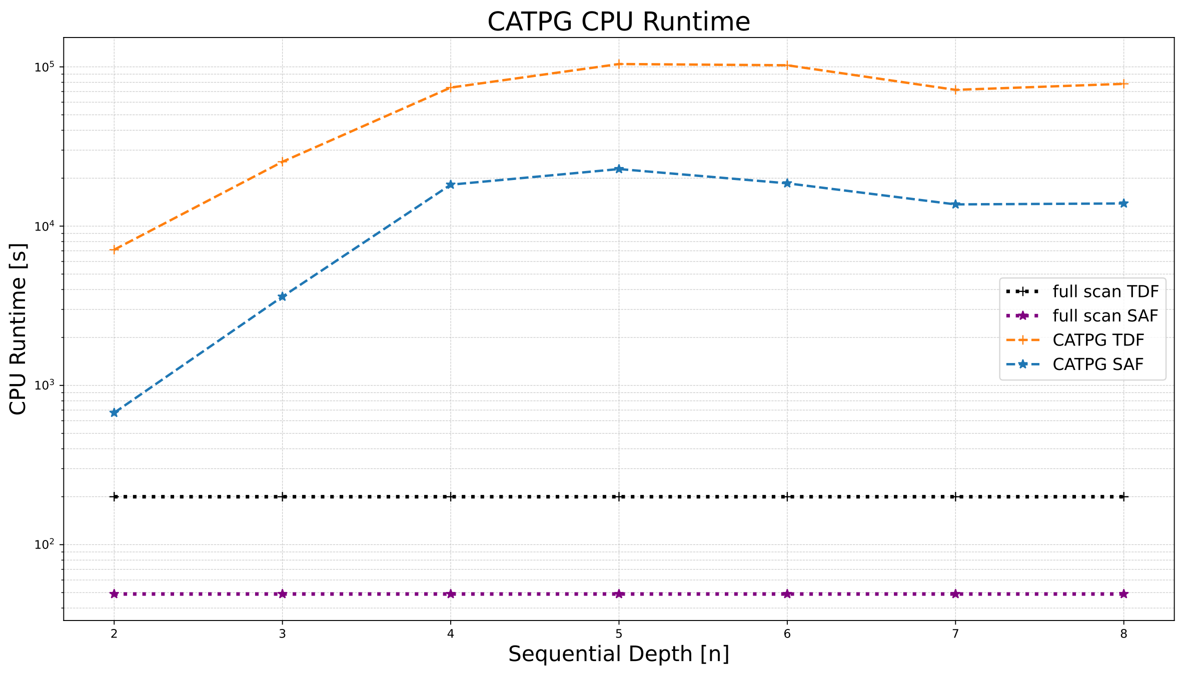


Figure 4.15: CATPG CPU Runtime

A detailed FC report for the sequential depth 8 SBST program is shown in Table 4.5. Compared to Table 4.4 the TDF FC of the register file module has increased by about 24 % by the conversion of the CATPG patterns. However, there is still significant TDF coverage loss in the register file as well as the EX stage. Moreover, the FC for both fault models is extremely low in the “cs_registers“ module.

Table 4.5: SAF and TDF Results based on TDF Patterns with Sequential Depth 8

Module	**#Faults**	**SAF FC (%)**	**SAF TCL (%)**	**TDF FC(%)**	**TDF TCL (%)**
`id_stage`	50330	90.98	4.00	78.91	9.37
`register_file`	36468	97.42	0.83	85.30	4.37
`controller`	2030	47.32	0.94	27.09	1.31
`decoder`	1894	71.54	0.48	63.20	0.62
`int_controller`	604	0.17	0.53	0.00	0.53
`ex_stage`	29914	86.99	3.40	70.56	7.73
`alu`	15834	77.84	3.08	62.77	5.18
`mult`	13502	98.15	0.20	79.70	2.40
`cs_registers`	16386	5.57	13.66	0.01	14.47
`if_stage`	10688	83.83	1.53	71.84	2.66
`aligner`	1784	88.45	0.18	78.20	0.34
`compressed_decoder`	1978	92.82	0.13	92.01	0.14
`prefetch_buffer`	4130	92.99	0.26	83.37	0.61
`load_store_unit`	5522	85.95	0.69	72.98	1.32
`sleep_unit`	42	41.67	0.02	19.05	0.03
`faultsim`	0	0	0	0	0
Total	113292	76.57	23.40	64.29	35.68

Figure 4.6 shows the program sizes of the SAF and TDF SBST programs. This program size is calculated by only counting effective test patterns from the SBST program. Effective test patterns are test patterns that detect undetected faults. As can be seen the SBST program is significantly larger when it targets TDFs compared to SAFs. This is to be expected as the TDF model is the more intricate model.

Table 4.6: SBST Program Size Results for Sequential Depth 8

Fault Model	SAF	TDF
Program Size (MB)	0.167	0.249

The SBST results shown in Table 4.5 are of an SBST program based on CATPG test patterns that target TDFs. This program was then fault simulated for both fault models. Table 4.7 shows the SAF results for an SBST program based on CATPG test patterns that target SAFs. As can be seen the resulting FC is slightly lower. Therefore, in the rest of the work all CATPG test patterns target TDFs, and these are then converted and simulated for both fault models.

Table 4.7: SAF SBST Results based on SAF Patterns

Module	# SAF Faults	SAF FC (%)	SAF TCL (%)
id_stage	50330	89.93	4.47
register_file	36468	96.37	1.17
controller	2030	46.43	0.96
decoder	1894	70.43	0.49
int_controller	604	0.17	0.53
ex_stage	29914	81.57	4.83
alu	15834	76.59	3.25
mult	13502	87.55	1.47
cs_registers	16386	4.95	13.75
if_stage	10688	84.74	1.44
aligner	1784	88.34	0.18
compressed_decoder	1978	92.16	0.14
prefetch_buffer	4130	92.99	0.26
load_store_unit	5522	86.06	0.68
sleep_unit	42	41.67	0.02
faultsim	0	0.00	0.00
Total	113292	74.68	25.29

Lastly, Table 4.8 shows the results of an SBST which does not use the hamming distance estimation for justification of the FSS. Instead it uses a random instruction. When comparing these results with Table 4.5 it can be seen that SAF FC is slightly higher with the random instruction, while TDF FC is slightly lower. However, the differences are very small so nothing conclusive can be said about it. Likely the hamming distance estimation functions like a random instruction picker.

Table 4.8: SBST Results for Random FSS Generation based on SAF and TDF Patterns

Module	# Faults	SAF (%)	SAF TCL (%)	TDF (%)	TCL (%)
id_stage	50330	89.00	4.88	76.86	10.28
register_file	36468	94.63	1.73	82.42	5.66
controller	2030	46.53	0.96	25.71	1.33
decoder	1894	77.67	0.37	71.28	0.48
int_controller	604	0.17	0.53	0.00	0.53
ex_stage	29914	80.36	5.15	72.32	7.26
alu	15834	75.03	3.47	64.06	5.00
mult	13502	86.66	1.57	81.96	2.13
cs_registers	16386	10.94	12.89	0.69	14.37
if_stage	10688	85.10	1.41	72.06	2.64
aligner	1784	88.34	0.18	72.03	0.44
compressed_decoder	1978	91.46	0.15	92.57	0.13
prefetch_buffer	4130	94.39	0.20	85.01	0.55
load_store_unit	5522	86.13	0.68	73.05	1.31
sleep_unit	42	41.67	0.02	19.05	0.03
faultsim	0	0.00	0.00	0.00	0.00
Total	113292	74.88	25.09	63.99	35.97

4.4. Discussion

A key observation from these results is that an increase in pre-converted CATPG FC translates to a higher post-conversion SBST FC. This observation will play an essential part in the design space exploration in Section 5.1. Moreover, the results also show that increasing sequential depth of the test patterns will increase SBST FC at the cost of long CATPG runtime. The works by Cheng et al. [11] and Kuo et al. [12] use a feedback loop from fault simulation back to CATPG to improve their FC results. However, they only consider using a sequential depth of 2, combining a larger sequential depth with this feedback loop might result in higher FC or less iterations of the feedback loop.

The developed SBST clearly does not have a high test pattern conversion rate, resulting in relatively low FC. This could be resolved by either implementing a justification engine to recreate FSSs more successfully or by implemented the aforementioned feedback loop. Furthermore, for TDFs the “cs_registers“ module has 0 % FC. This may indicate that this module is hard to test for delay faults, while the core is in functional mode.

4.5. Conclusion

In conclusion, an SBST generation framework for RISC-V cores has been developed. The resulting SBST program achieved 76.57 % SAF FC, and 64.29 % TDF FC on the CV32E40P core. The detailed FC results show that the EX stage, ID stage, and “cs_registers“ module suffer from low FC. This indicates the need to improve FC, especially for TDFs, through the addition of a DFT. It is also shown that pre-converted CATPG test pattern results give an indication of post-conversion SBST program results. Therefore, a design space exploration will be carried out by tweaking the partial scan CATPG setup.

5
DFT Design

This chapter contains the design of the DFT addition to the SBST created in the experimental setup. Section 5.1 covers the design space exploration, including the methodology and results. The results of the DSE are discussed in Section 5.2, and an avenue for the DFT is picked. Then, Section 5.3 considers various design options and explains the choices made in picking one. The DFT implementation is covered in Section 5.4. Finally, the results are presented in Section 5.5 and discussed in Section 5.6. This is followed by a conclusion in Section 5.7

5.1. Design Space Exploration

The DSE to identify avenues for DFT addition that complements SBST is one of the novelties of this thesis. The main idea of the DSE is to use partial scan patterns for SBST generation, so that scan cells can be added to see how FC is effected by increased observability or controllability. Adding observability and controllability to flip-flops, not considered in the pattern-to-program conversion, mimics the effect a DFT with similar functionality. The premise is that the CATPG test pattern FC, provides an indication of the FC after being converted to an SBST program. This was concluded in Section 4.5. A DSE can be performed based on this indication of potential FC increase. Also included in the DSE is the consideration of the trade-off between area increase and FC increase. The amount of flip-flops being made observable or controllable indicates the amount of area overhead. Moreover, the capabilities added to these flip-flops are also accounted for. Making a flip-flop only observable will cause less area overhead, than making it controllable and observable.

5.1.1. Methodology

The DSE methodology determines which scan configurations are explored in the DSE. A scan configuration is defined as the configuration of scan cells, it specifies which scan cells are added to the baseline scan configuration used for the SBST generation covered in Chapter 4. Aside from which scan cells are added to the partial scan setup, the scan configuration also conveys if they are observable, controllable, or observable and controllable.

The DSE should cover all possible abilities of scan cells that it adds, to be able to make the most educated choice about the DFT addition. Furthermore, there should be some systematic way of deciding which scan cell combinations are considered for scan configurations that are simulated during the DSE. Algorithm 3 outlines the DSE methodology. The inputs are the baseline scan configuration $C_{baseline}$, a set of ability scan cells can have A, and parameter x. The outputs are the optimal scan configuration $C_{optimal}$, a set of combined scan configurations S_{comb}, and a set of scan configurations S_{high} which perform x percent better than the baseline. The first step generates set M consisting of one entry per core module, each entry consists of all the potential scan cells in the corresponding module. Then, some variables are initialized and the baseline scan configuration is added to the set with scan configurations to be simulated S_{DSE}. Next, the cartesian product of A and M is computed to create scan configurations. These scan configurations are each combined with $C_{baseline}$ and added to S_{DSE}. All scan configurations in S_{DSE} are now simulated for sequential depth 2 to 8. Then, based on parameter x a set is made of a selection of high FC scan configurations S_{high}. After, the set S_{comb} is created, consisting of any possible combinations of the scan configurations in S_{high}. Once created, all scan configurations in S_{comb} are now simulated for sequential depth 2 to 8. Based on the results of these simulations the optimal scan configuration according to Equation 5.1 is found $C_{optimal}$. Finally, the algorithm returns this optimal scan configuration, S_{comb}, and S_{high}.

$$AFC = \frac{FC}{additional\_area} \tag{5.1}$$

Algorithm 3 DSE Methodology for Finding DFT Avenue to Complement SBST

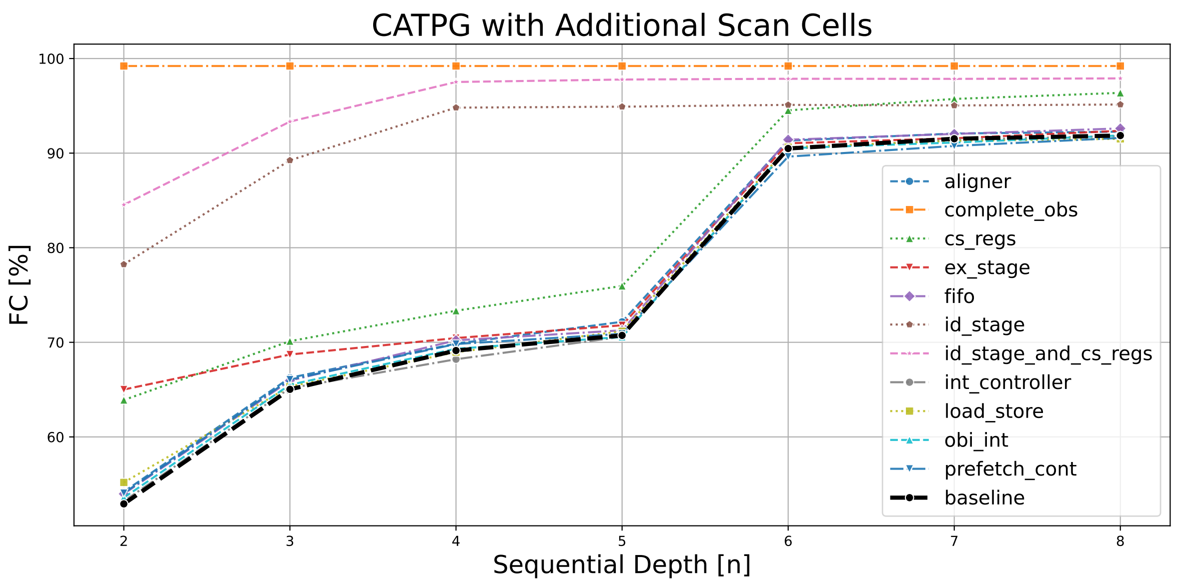
```
INPUT: $C_{baseline}$, $A \in \{obs, obs\&cont\}$, $x$
OUTPUT: $C_{optimal}$, $S_{comb}$, $S_{high}$
 1: Create Set $M$ with selection of scan cells $SC_{module}$ per core module.
 2: Set $i = 0$, $K$ = size($A$), $S_{high} = \emptyset$, $S_{DSE} = C_{baseline}$, $S_{comb} = \emptyset$.
 3: for $i \leq K$ do
 4:     For each entry in $M$ create a scan configuration where all scan cells have ability $A_i$ .
 5:     Complement $C_{baseline}$ by combining with created scan configurations and add to $S_{DSE}$.
 6:     $i = i + 1$
 7: end for
 8: Set $i = 2$, $K = 9$
 9: for $i \leq K$ do
10:     Run CATPG with sequential depth $i$ for all scan configurations in $S_{DSE}$.
11:     $i = i + 1$
12: end for
13: Add scan configurations to $S_{high}$, for which the best run had at least $x$% FC more than best baseline
    run.
14: Add all possible combinations of the configurations in $S_{high}$ to $S_{comb}$.
15: for $i \leq K$ do
16:     Run CATPG with sequential depth $i$ for all scan configurations in $S_{comb}$.
17: end for
18: For the best run of every configuration in $S_{comb}$ and $S_{high}$ calculate the compound metric of FC and
    additional area.
19: The configuration with the highest compound metric score is $C_{optimal}$.
```

5.1.2. Results

This section will briefly present the results from the DSE following Algorithm 3. The results will be interpreted and discussed in Section 5.2.

Fault Coverage for Controllable and Observable Scan Cells

Figure 5.1 shows the SAF results of the CATPG run with additional observable and controllable scan cells for each module. The two modules which seem to benefit most from additional observability and controllability at sequential elements are “cs_regs“ and “id_stage“. From this, it follows that a combination of the two configurations also has a FC increase compared to the baseline, this can also be seen in Figure 5.5. The plot also shows that the “complete_obs“ configuration that includes all sequential elements, performs like full scan when these elements are given controllability and observability.



w

Figure 5.1: SAF Results with Additional Controllable and Observable Scan Cells

Figure 5.2 shows the results for the same simulations as Figure 5.1 but for TDFs. It shows similar trends to the SAF results, but the FC is systematically lower. However, it can be observed that for a sequential depth of 8 the FC results of the two configurations “cs_regs“ and “id_stage“ are about the same. This is expected as the “cs_regs“ module is hard to control, so it benefits more from extra controllability and observability than just observability.

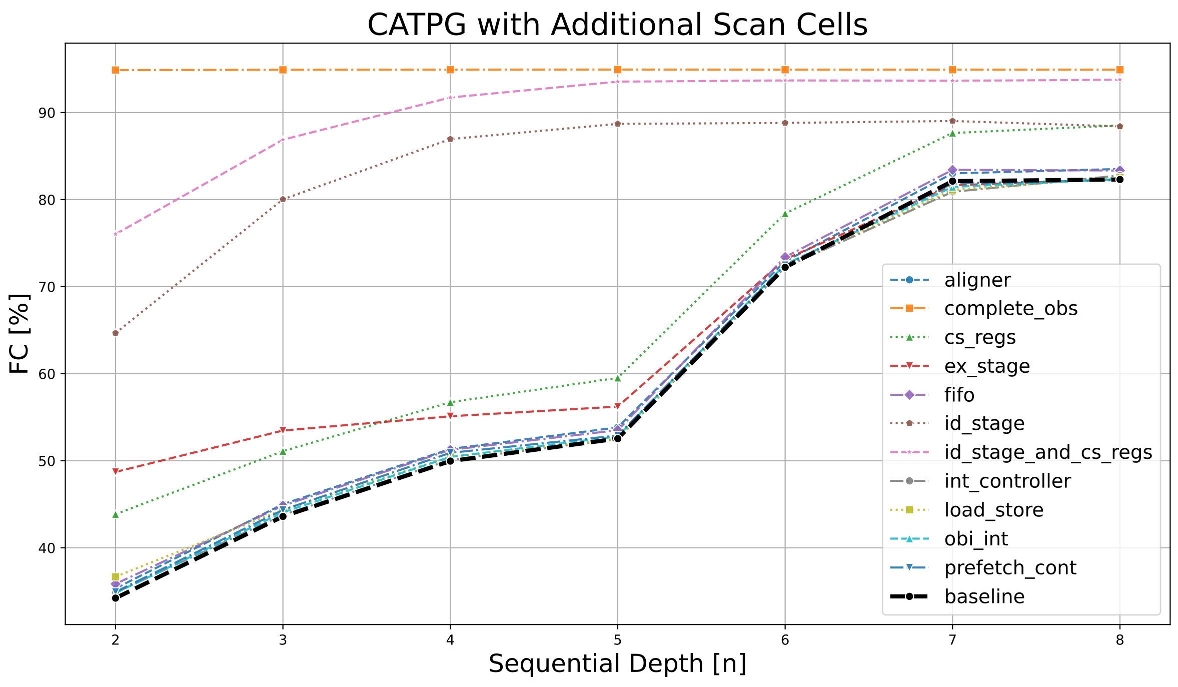


Figure 5.2: TDF Results with Additional Controllable and Observable Scan Cells

Fault Coverage for Observable Scan Cells

Figure 5.3 shows the SAF FC for CATPG with additional observation points in each module. The results showing FC of observable only additional scan cells indicate that this is less of an improvement over the baseline, compared to observable and controllable scan cells. This is expected as the capabilities of the additional scan cells are decreased, and thereby the testability of the core. The three best performing configurations are still the same three that perform best for additional controllable and observable scan cells.

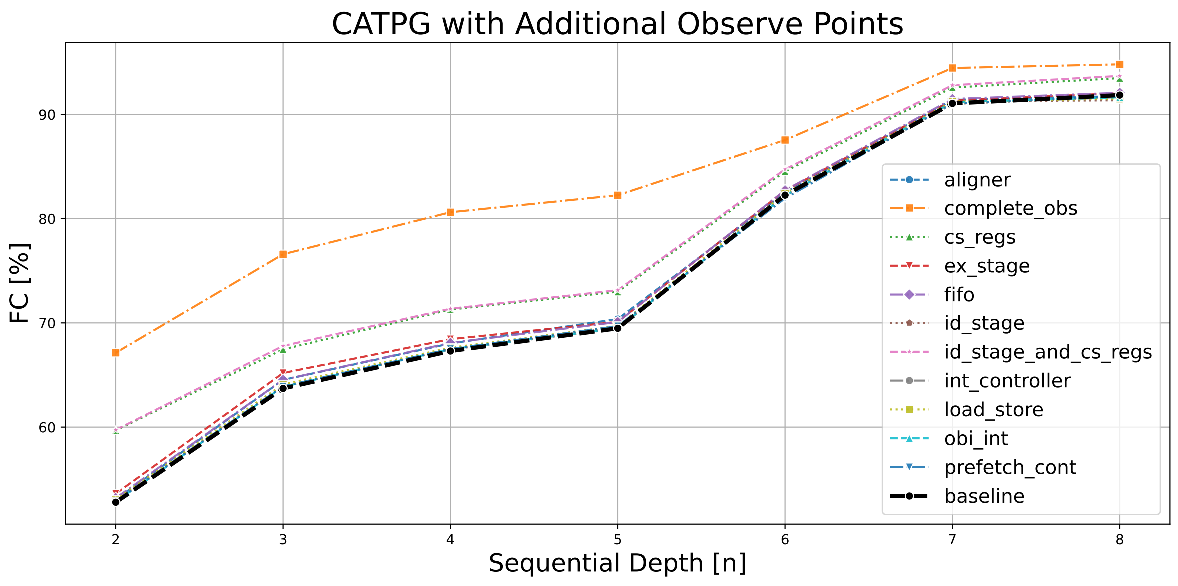


Figure 5.3: SAF Results with Additional Observe Points

Figure 5.4 shows the TDF fault coverage for CATPG with additional observation points in each module. The exact same simulations with additional observable scan cells, give different results. More details on this can be found in Appendix B. The FC results for the two fault models SAFs and TDFs show similar results. Specifically trends seen for increasing sequential depth for both fault models are the same, but the FC for SAFs is systematically higher.

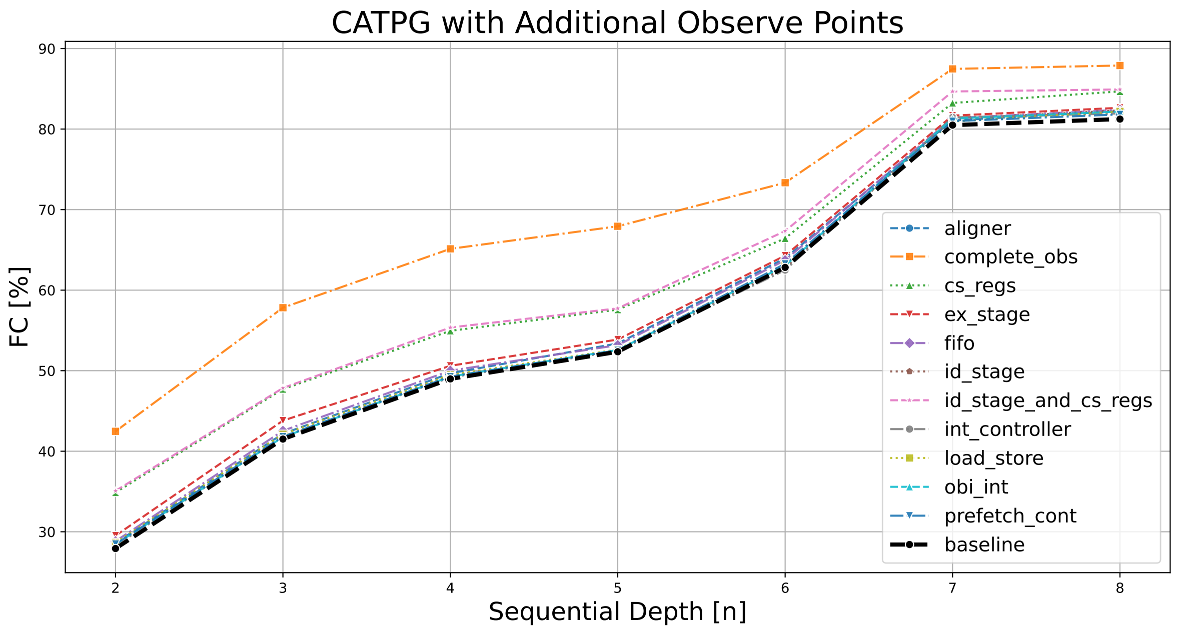


Figure 5.4: TDF Results with Additional Observe Points

Area

This section covers the same results as the previous section, but area costs are included, in terms of additional scan cells. Additionally, only the FC is provided for the runs with sequential depth 8, which has the highest FC results.

Area Controllable and Observable Scan Cells

Figure 5.5 shows the SAF results for the run with additional observable and controllable scan cells. The additional scan cell numbers give an estimate of the increase in area if the functionality of the additional scan cells would be added to the design through a functional DFT. As expected a linear trend can be observed, meaning that more additional scan cells provide more FC increase. Furthermore, the “id_stage“ configuration stands out due to it having little additional scan cells for significant FC increase.

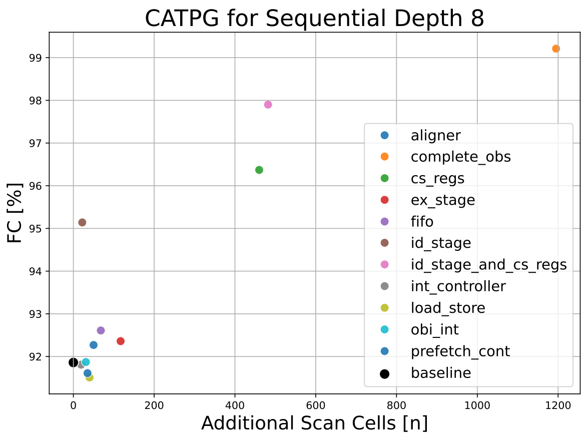


Figure 5.5: SAF Results with Additional Controllable and Observable Scan Cells and Sequential Depth 8

Figure 5.6 shows the TDF results for the run with additional observable and controllable scan cells. Once again, the TDF results look similar to the SAF but the overall FC is lower.

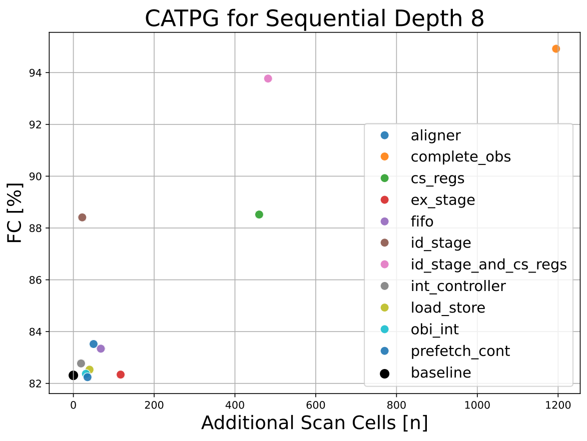


Figure 5.6: TDF Results with Additional Controllable and Observable Scan Cells and Sequential Depth 8

Area Observable Scan Cells

Figure 5.7 shows the additional number of observable scan cell for each configuration and the corresponding SAF FC. A linear trend can once again be observed.

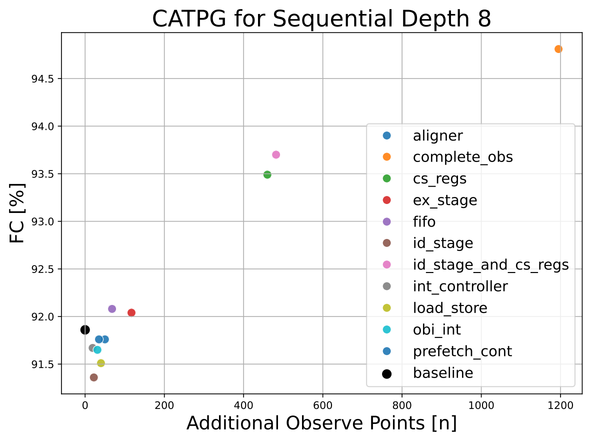


Figure 5.7: SAF Results with Additional Observe Points and Sequential Depth 8

Figure 5.8 shows the additional number of observable scan cell for each configuration and the corresponding TDF FC. Compared to the SAF results more of a spread can be observed for the configurations with the least amount of additional scan cells. This is the case because the percentage point difference between these configurations and baseline is larger than for the SAF results. This indicates that increased observability is more essential for detecting TDFs than SAFs.

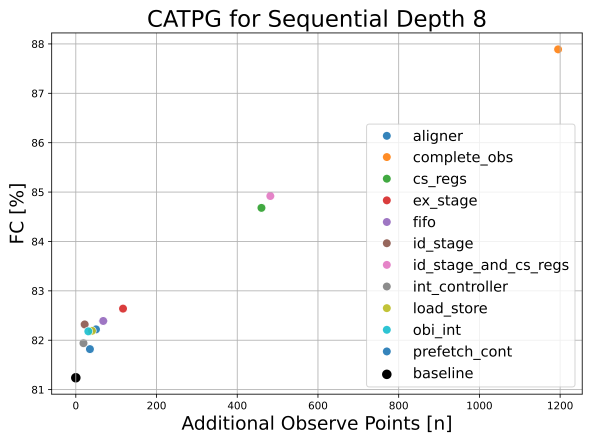


Figure 5.8: TDF Results with Additional Observe Points and Sequential Depth 8

5.1.3. Discussion

The results of the DSE presented previously show that adding additional functionality to the core could potentially increase FC of SBST programs that make use of this. Additional controllability and observability is shown to provide more FC increase than just observability. However, What is not included is that for the controllable and observable scan cells, each additional scan cell will add more area than for observable only scan cells. This follows from the fact that extra functionality requires extra hardware. A general trend is observed which shows that more additional scan cells provide more FC. This is the main trade-off that should be considered when choosing an avenue for a DFT addition.

5.2. Design Choices

The design choices made about the flip-flops to include for the DFT addition are based on the DSE runs. These are the runs with additional observable only scan cells and observable and controllable scan cells. The set with contenders S_{high} according to Algorithm 3, with parameter $x = 2$, will be considered. From the scan cell configurations with only additional observable scan cells “complete_obs“, “cs_regs“, and “id_stage_and_cs_regs“ are contenders. From the scan configurations with additional observable and controllable scan cells the same configurations are contenders, but and additionally “id_stage“ is also a contender.

Making a flip-flop controllable and observable will cost more area than just making a flip-flop observable. So, area overhead is considered taking this into account as well as the amount of additional scan cells for the contender configurations. The configurations that immediately seem less favorable when taking into account area overhead, are “complete_obs“, “id_stage_and_cs_regs“, and “cs_regs“ (controllable and observable). Because they provide relatively little extra FC for the amount of extra area overhead.

The scan configurations that are left for consideration are “id_stage“ (controllable and observable), and “cs_regs“ (observable). Due to the fact that Tessent does not allow setting scan cell constraints, the pattern-to-program conversion does not perfectly recreate the FSSs from the test patterns. For this reason it is expected that a few extra id_stage flip-flops that are controllable will not necessarily translate to an increase in SBST FC. As the rest of the FSS might not completely match the pattern, and these additional flip-flops might depend on this. However, adding observable flip-flops to the control registers will likely not suffer the same problem. Additionally, the baseline SBST results from Table 4.5 show that most FC loss happens in the “cs_regs“ module. For those reasons adding observability to the flip-flops in the “cs_regs“ module is chosen as the goal of the DFT addition. Besides that the small “id_stage“ configuration for observable only scan cells has also been implemented for comparison.

5.3. DFT Method

This section will cover the high-level design of the chosen DFT addition. First, a list of criteria is presented. Then, based on these criteria several options are considered and the option that meets the criteria the best is picked.

5.3.1. Criteria

Table 5.1 shows the criteria that are defined for the DFT addition. These criteria can be used to measure the success of the design. Criteria `AR` and `TI` express that the time and area overhead should be minimized. The `IO` criterion specifies that the DFT cannot add any extra inputs or outputs to the core. The criteria `RE`, `OB`, and `SS` are functional criteria and are explained in the criterion description column.

Table 5.1: Criteria for the DFT Design

Criterion Code	Criterion Description
`AR`	The design must minimize area overhead.
`TI`	The design must minimize the increase in SBST program run time.
`IO`	The design has to make use of existing primary inputs and outputs.
`RE`	The design must be able to reliably detect defects.
`OB`	The design must be able to propagate state variables to an observable output of the core.
`SS`	The design must be able to take a snapshot of a selection of state variables at a specific cycle.

5.3.2. Possible Options

For the trade-off analysis, multiple design options are considered. Table 5.2 shows the different options that are considered for the DFT addition. Option 1 is a standard scan design that is only used for observation. Option 2 is a scan chain like structure that can be used in functional mode. So, the register values are shifted out through sequential elements that do not effect the rest of the functionality of the circuit when used. Options 3 and 4 are explained in Table 5.2. Lastly, option 5 uses a clock that is faster than the core’s clock to serialize the state variable data, thereby reducing the amount of extra memory elements needed. Additionally, this option would require the core to have an extra primary output to output this serialized data to.

Table 5.2: Options for the DFT Design

Option Number	Option Description
1	Standard scan design, but it is only used for observation.
2	Multiple shift register structures inspired by scan chains, but they can be used in functional mode.
3	Wires from the flip-flops straight to (by the SBST template) reserved registers, with a MUX that controls it.
4	Add shadow D latches to the flip-flops that start holding their value at a specific cycle. Complemented by an FSM that writes the values to (by the SBST template) reserved registers.
5	Use a fast clock to partially serialize the state variables, reducing extra memory elements.

5.3.3. Trade-off

The trade-off table is shown in Table 5.3. All the options score well on the functional criteria, except for option 5. This design might not be reliable due to the fact that it requires clock domain crossing, so it scores a one on this criterion. Furthermore, the ideas that do not add any extra primary inputs or outputs score a 5 on the `IO` criterion, else the given score is one. The `AR` criterion score is an estimation of the area overhead each option will bring. Option 2 scores a 1 on this criterion, as it would require one extra flip-flop for each flip-flop that needs to be made observable. Options 1 and 3 would add a significant amount of wiring and multiplexers, so these options score a 2. An option that likely has a bit less area overhead than this is option 4, and the lowest area option is option 5. For the timing criterion `TI` options 2 and 5 score the best, because these designs would not lengthen the SBST program run time. Options 3 and 4 would require some extra instructions that write the registers with the state variables to memory, so they score a 4. Finally, option 1 scores the worst because it would require shifting out the state variables with a slow clock, while pausing the functionality of the rest of the core. The option with the overall highest score is saving state variables in latches, and writing them into registers with an FSM.

Table 5.3: DFT Design Trade-off Table

Criterion Code	Option 1	Option 2	Option 3	Option 4	Option 5
`AR`	(2/5)	(1/5)	(2/5)	(3/5)	(5/5)
`TI`	(1/5)	(5/5)	(4/5)	(4/5)	(5/5)
`IO`	(1/5)	(1/5)	(5/5)	(5/5)	(1/5)
`RE`	(5/5)	(5/5)	(5/5)	(5/5)	(1/5)
`OB`	(5/5)	(5/5)	(5/5)	(5/5)	(5/5)
`SS`	(5/5)	(5/5)	(5/5)	(5/5)	(5/5)
Total Score	19	22	26	27	22

5.4. Hardware Implementation

In total three DFT additions are implemented for extra observability in the control and status registers (CSR) module and the ID stage module. Figure 5.9 shows the high-level design of the CS DFT, which is implemented using design option 4. Only one extra module has been added to the core: “cs_buffer“. This module has the same amount of latches as the control and status registers module has flip-flops that are accessible in functional mode. These latches are used to capture the contents of these flip-flops at a specific cycle. Therefore, there is also some logic that counts down to the capture cycle. A test instruction is added to the core by modifying the decoder unit, this instruction informs the “cs_buffer“ when to start counting down and from what number to count. Another functionality of the “cs_buffer“ is that it periodically writes the captured values to a reserved register. This register is saved in memory by instructions inserted into the SBST program. Lastly, some logic has been added to the register file to be able to write the captured values into a register without disturbing the normal functionality of the core.

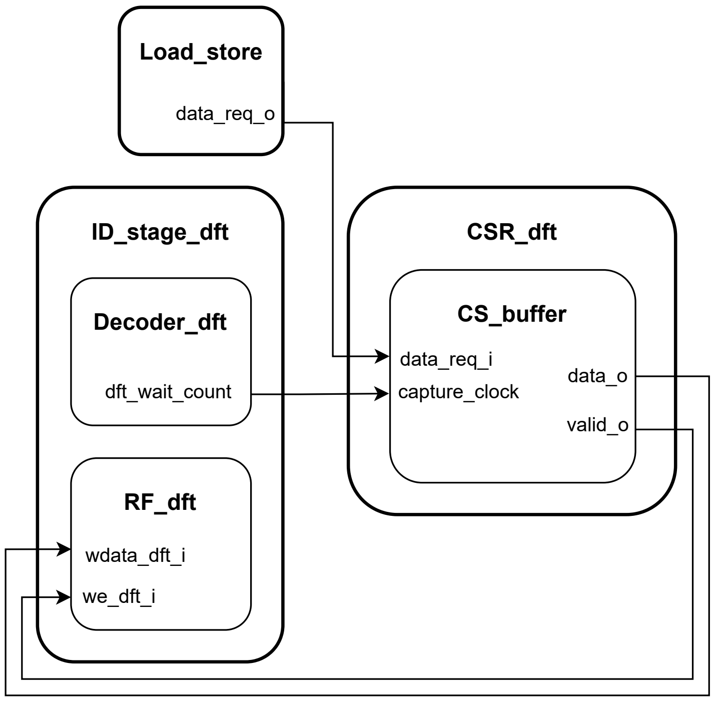


Figure 5.9: DFT CS Design

Control and Status registers are auxiliary registers present in most modern cores, micro-controllers, and I/O devices. These registers serve many different purposes like saving the configuration of how cores should respond to interrupts and providing debugging capabilities [97]. Therefore, in functional mode these registers are hard to control and observe through instructions. Appendix A shows which registers are accessible in functional mode, and what the functions of all these registers are. Due to the convoluted and hard to control nature of the status and control registers ATPG patterns result in low fault coverage in this unit. Therefore, a manual test program has been developed by writing to these registers one by one, thereby sensitising TDFs in them. This allowed for the implementation of an optimised version of the CS DFT design. As only one register is tested per test pattern, only one register's value needs to be saved at a time. This allows for the removal of the latches, by adding a mux and directly routing the value of the selected register to a GPR.

Effectively, the optimised CS DFT design implements option 3 from Table 5.2. However, because a maximum of 32 bits are written to at a time the bus from the "cs_buffer" to the register files only needs to be 32 bits wide. So, under these circumstances this option scores a lot better on the area criterion AR. Figure 5.10 shows the optimised CS design, which includes an additional connection between the decoder and the "cs_buffer". This connection sends the information found in the DFT instruction on which register should be saved, to the MUX in the "cs_buffer".

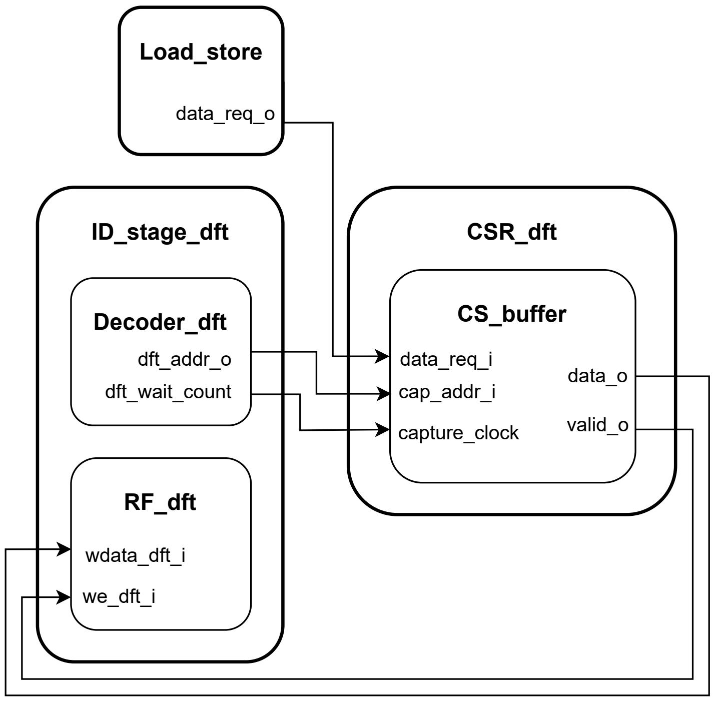


Figure 5.10: DFT CS Design Optimised for Area

Another DFT has been implemented in the ID stage module. This includes 22 flip-flops that were not included in the baseline partial scan configuration. This DFT is placed close to the register files, and decoder. It does use ATPG patterns, but because there are only 22 flip-flops there is no need to use latches to save the values of the flip-flops. So this design also uses option 3 from Table 5.2. The bit values are routed straight to a general purpose register, and written at the right cycle using the write enable signal.

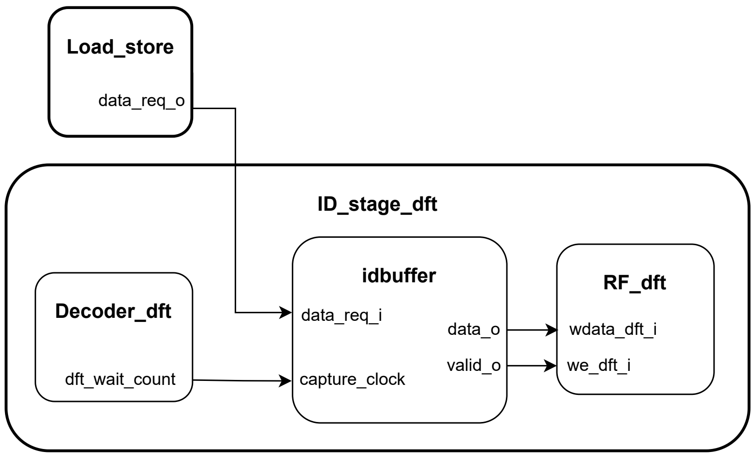


Figure 5.11: DFT ID Design

5.5. Results

This section shows the results of the three implemented DFT designs in terms of FC increase compared to baseline, and area overhead compared to baseline. Then, the proposed work is compared to state-of-the-art works. No new metrics are introduced in this section.

5.5.1. Fault Simulation Results

Table 5.4 shows the results for the CS DFT, in terms of number of faults, FC, and TC loss. Results are shown for two fault models: SAF and TDF. The results in Table 5.4 show that compared to baseline (see Table 4.5) the CS DFT addition achieves a FC increase of 6.29 percentage points for SAFs and 2.4 percentage points for TDFs, at the cost of 0.84% area increase. It should be noted that the total number of faults is slightly higher than the baseline number of faults, due to the added hardware. So, the FC results are low estimates because the test patterns were not generated to detect the faults in the DFT hardware. Lastly, most of the additional FC, as expected, is in the “cs_registers“ module.

Table 5.4: SAF and TDF Results for the CS DFT and Optimised CS DFT

Module	#Faults	SA FC (%)	SA TCL (%)	TDF FC(%)	TDF TCL (%)
`id_stage`	51126	91.42	3.82	79.08	9.33
`register_file`	36906	97.44	0.82	85.28	4.74
`controller`	2016	48.09	0.91	25.89	1.30
`decoder`	2058	77.14	0.41	68.83	0.56
`int_controller`	604	11.09	0.47	0.00	0.53
`ex_stage`	30758	87.14	3.41	69.41	8.17
`alu`	16004	77.71	3.09	62.93	5.15
`mult`	14176	97.75	0.26	76.26	2.92
`cs_registers`	16216	46.94	7.51	18.06	11.60
`if_stage`	10558	82.77	1.59	70.31	2.74
`aligner`	1772	88.66	0.18	78.44	0.33
`compressed_decoder`	1978	92.82	0.12	92.01	0.14
`prefetch_buffer`	3940	91.64	0.29	82.41	0.61
`load_store_unit`	5522	85.66	0.69	72.67	1.32
`sleep_unit`	42	41.67	0.02	21.43	0.03
`faultsim`	0	0	0	0	0
Total	114612	82.86	17.10	66.72	33.24

Table 5.5 shows the results for the ID DFT, again in terms of number of faults, FC, and TC loss. The ID DFT addition increases the FC with 1.01 percentage points for SAFs and 1.92 percentage points for TDFs, for an area overhead of 0.65%. Similar to the CS DFT the total number of faults is slightly higher than baseline due to the hardware addition. Lastly, increases in FC can be seen in the ID stage, EX stage, and load store unit. This follows from the fact that the ID stage is connected to both the EX stage and the load store unit. So, adding observability to the ID stage helps detect faults in the aforementioned modules as well.

Table 5.5: SAF and TDF Results for ID DFT

Module	#Faults	SAF FC (%)	SAF TCL (%)	TDF FC(%)	TDF TCL (%)
`id_stage`	50732	91.92	3.58	81.25	8.30
`register_file`	36696	97.87	0.68	87.58	3.98
`controller`	2024	50.20	0.88	28.21	1.27
`decoder`	2004	76.50	0.41	68.46	0.55
`int_controller`	604	0.17	0.53	0.00	0.53
`ex_stage`	30778	89.21	2.87	73.51	7.09
`alu`	16020	80.94	2.65	67.08	4.59
`mult`	14180	98.98	0.11	80.84	2.36
`cs_registers`	16406	5.48	13.56	0.00	14.34
`if_stage`	10558	83.03	1.57	70.94	2.68
`aligner`	1772	88.71	0.17	79.06	0.32
`compressed_decoder`	1978	93.63	0.11	92.97	0.12
`prefetch_buffer`	3940	91.76	0.28	83.12	0.58
`load_store_unit`	5522	85.71	0.69	74.97	1.21
`sleep_unit`	42	41.67	0.02	19.05	0.03
`faultsim`	0	0	0	0	0
Total	114440	77.58	22.38	66.21	33.76

5.5.2. Area Results

All area results are presented in µm^2. In Table 5.6 the detailed area of the core with no DFT additions is shown as a baseline. It can be observed that the register file is the largest module, and the next largest is the control status registers module. This is likely due to the fact that most of the sequential elements present in the core can be found in these two modules. Lastly, it is noticeable that roughly a third of the area is interconnect wiring.

Table 5.6: Area Results Core Baseline

Instance	Cell Count	Cell Area	Net Area	Total Area
cv32e40p_core	12399	25866.767	12256.664	38123.431
id_stage	5210	11129.605	4498.194	15627.799
register_file	3558	8114.576	3305.877	11420.454
controller	270	334.102	181.906	516.008
decoder	265	268.834	171.240	440.074
int_controller	80	138.121	33.314	171.435
ex_stage	3431	6711.314	3028.965	9740.280
alu	1898	3115.400	1610.312	4725.712
mult	1484	3479.137	1367.019	4846.156
cs_registers	1772	4211.903	1510.576	5722.479
if_stage	1201	2708.269	938.809	3647.078
aligner	187	496.213	117.459	613.673
compressed_decoder	257	264.776	184.539	449.315
prefetch_buffer	468	1230.919	335.763	1566.683
load_store_unit	737	1060.164	502.457	1562.621
sleep_unit	6	13.759	1.902	15.661

Table 5.7 shows the core area results with the CS DFT addition before optimization. Compared to baseline there is a 7.35% increase in area. This is mainly caused by the "csbuffer" module that contains many latches. Increases in the decoder and register file modules can also be seen due to

added hardware. A small increase in the EX stage can be seen as well, this could be because the CS registers module is connected to the EX stage module. So, changes in one of them might slightly effect the other during synthesis.

Table 5.7: Area Core CS DFT with Total Area Increase

Instance	Cell Count	Cell Area	Net Area	Total Area	Increase
cv32e40p_core_cs_dft	13481	27815.281	13111.547	40926.829	+7.36%
id_stage	5279	11242.678	4529.083	15771.760	+0.92%
register_file	3626	8199.072	3339.825	11538.897	+1.04%
controller	259	332.867	177.140	510.007	–1.16%
decoder	274	278.712	171.520	450.232	+2.32%
int_controller	80	138.121	33.314	171.435	+0.00%
ex_stage	3482	6677.975	3109.124	9787.099	+0.45%
alu	1858	3136.745	1599.438	4736.183	+0.22%
mult	1575	3424.453	1458.051	4882.504	+0.75%
cs_registers	2789	6102.205	2238.833	8341.039	+45.76%
csbuffer	1034	1880.071	681.864	2561.935	
if_stage	1145	2684.455	915.917	3600.372	–1.28%
aligner	186	494.626	116.521	611.146	–0.41%
compressed_decoder	257	264.776	184.539	449.315	+0.00%
prefetch_buffer	409	1205.341	307.959	1513.300	–3.41%
load_store_unit	737	1060.870	502.457	1563.326	+0.04%
sleep_unit	6	13.759	1.902	15.661	+0.00%

Table 5.8 shows the core area results with the optimised CS DFT addition. Compared to baseline there is a 0.84% increase in area. The optimization of the CS DFT has decreased the area of the “csbuffer“ module significantly by removing the latches. It can also be seen that the decoder total area increase doubled, this is because the DFT instruction for the optimised design also contains an address that needs to be decoded.

Table 5.8: Area Core Optimised CS DFT with Total Area Increase

Instance	Cell Count	Cell Area	Net Area	Total Area	Increase
cv32e40p_core_cs_dft_opt	12608	25998.361	12444.665	38443.026	+0.84%
id_stage	5333	11232.446	4563.299	15795.745	+1.07%
register_file	3628	8196.955	3340.545	11537.500	+1.03%
controller	265	335.513	179.992	515.505	–0.10%
decoder	280	286.474	174.141	460.615	+4.67%
int_controller	80	138.121	33.314	171.435	+0.00%
ex_stage	3518	6645.164	3134.759	9779.924	+0.49%
alu	1891	3115.577	1607.666	4723.243	–0.05%
mult	1578	3412.811	1475.458	4888.268	+0.87%
cs_registers	1828	4332.384	1538.758	5871.142	+2.60%
csbuffer	52	119.599	39.385	158.984	
if_stage	1145	2683.220	915.917	3599.137	–1.31%
aligner	186	494.626	116.521	611.146	–0.41%
compressed_decoder	257	264.776	184.539	449.315	+0.00%
prefetch_buffer	409	1204.459	307.959	1512.418	–3.47%
load_store_unit	737	1060.693	502.457	1563.150	+0.03%
sleep_unit	6	13.759	1.902	15.661	+0.00%

Table 5.9 shows the core area results with the ID stage DFT addition. Compared to baseline there is a 0.65% increase in total area. Most of the area increase is caused by the addition of the “idbuffer“,

and the extra decoding capabilities added to the decoder.

Table 5.9: Area Core ID_DFT with Total Area Increase

Instance	Cell Count	Cell Area	Net Area	Total Area	Increase
cv32e40p_core_id_dft	12541	25961.317	12409.802	38371.120	+0.65%
id_stage	5319	11300.890	4575.550	15876.439	+1.59%
register_file	3587	8175.787	3320.822	11496.609	+0.67%
controller	265	337.982	170.204	508.186	–1.52%
decoder	280	286.474	174.141	460.615	+4.67%
int_controller	80	138.121	33.314	171.435	+0.00%
idbuffer	52	119.070	39.385	158.455	
ex_stage	3516	6657.865	3119.681	9777.546	+0.38%
alu	1890	3116.635	1609.044	4725.679	–0.00%
mult	1577	3424.453	1459.002	4883.455	+0.77%
cs_registers	1776	4212.432	1514.818	5727.250	+0.09%
if_stage	1145	2683.220	915.917	3599.137	–1.31%
aligner	186	494.626	116.521	611.146	–0.41%
compressed_decoder	257	264.776	184.539	449.315	+0.00%
prefetch_buffer	409	1204.459	307.959	1512.418	–3.47%
load_store_unit	737	1061.399	502.457	1563.855	+0.07%
sleep_unit	6	13.759	1.902	15.661	+0.00%

To conclude the optimised CS DFT performs the best for slightly more area overhead than the ID DFT. This indicates the relation between area overhead and FC increase, and the corresponding trade-off that can be made.

5.5.3. Comparison with Other Works

The results of the proposed combination of SBST and DFT will be compared with other relevant works. The area overhead will be compared with the two other works that have complemented SBST with DFT. Furthermore, FC results for SAFs and TDFs will be compared.

Stuck-at Faults

The SAF FC results are compared with other works in Table 5.10. Three different results from this thesis are shown in Table 5.10. The “CS“ and “ID“ implementations refer to the implemented optimised CS DFT design and the ID DFT design respectively. The “ID (adjusted)“ entry is added for a better comparison with the other RISC-V work by Faller et al. [13]. They have removed the csr registers, decoder, and load store unit from the fault list, “ID (adjusted)“ is the ID DFT but the aforementioned modules are removed from the fault list.

Table 5.10: SAF FC Comparison with Other Works

Work	Proposed				Faller et al.				Riefert et al.
Year	2025				2023 [13]				2016 [23]
Implementation	SBST	CS	ID	ID (adjusted)	–				
SAF FC (%)	76.57	82.86	77.58	90.27	75.85	82.42	79.18	45.40	95.02
#Faults	113292	114612	114440	90106	22810	34039	35645	64557	54181
Area Overhead (%)	0	0.84	0.65	0.65	0	0	0	0	0
Program Size (MB)	0.167	0.168	0.216	0.216	0.016	0.026	0.018	0.046	–
Generation Time (h)	22	22	22	22	16.78	46.80	54.18	96.39	–
Instruction Set	RV32IMC_Zicsr				RV32E	RV32I	RV32I_Zicsr	RV32I	MIPS32

Table 5.10 shows that the proposed work has higher SAF FC than the other RISC-V work. However, this is only the case after the addition of a DFT. When the three hard-to-test modules mentioned in Section 5.5.3 are removed from the fault list, the proposed work has relatively high FC. The generation time of this work is also lower than that of the other RISC-V work. However, it is important to note that the program of Faller et al. [13] are significantly smaller. This is due to the fact that this is not the focus

of this work, and no attempts have been done to decrease program size. Lastly, it can be seen that the work that is done on a MIPS32 core has higher FC. This trend is also present for TDF results. The MIPS32 ISA is more complicated than the RISC-V ISA, so when a fault occurs, many processes are affected and the chance is higher that the fault becomes observable in some way. This could be an explanation for why it is easier to test MIPS32 cores.

Transition Delay Faults

The TDF FC results are compared with other works in Table 5.11. Again three different results from this thesis are shown in Table 5.11. The “CS“ and “ID“ implementations refer to the two DFT designs that were implemented. The “ID (no csr)“ implementation is added for a better comparison with the other works. None of the other works have a CSR module in their results, but they also do not mention removing it. To provide a more fair comparison “ID (no csr)“ is the same as “ID“ but the CSR module is removed from the fault list.

Table 5.11: TDF FC Comparison with Other Works

Work	Proposed				Kuo et al.	Cheng et al.	Chen et al.		
Year	2025				2024 [12]	2023 [11]	2021 [8]		2019 [88]
Implementation	SBST	CS	ID	ID (no csr)	–				
TDF FC (%)	64.29	66.72	66.21	77.29	92.25	90.44	86.40	97.82	94.94
#Faults	113292	114612	114440	98034	55190	50486	50486	90240	88258
Area Overhead (%)	0	0.84	0.65	0.65	0	0	0	0	0
Program Size (MB)	0.249	0.250	0.348	0.348	2.17	–	–	4.26	0.195
Generation Time (h)	22	22	22	22	–	39	26	153.41	–
ISA	RISC-V							MIPS32	

The comparison of the TDF results shows that the proposed work has lower TDF FC than state-of-the-art SBST works. The difference in results between SAF and TDF results stems from the SBST generation. Instruction justification has been A low priority in this work, and this is more essential for detecting TDFs than SAFs. Most likely this is causing the large difference in results.

Area

The area overhead of the two implemented DFT designs is compared with two other works in Table 5.12. As indicated both works have implemented their DFTs on a small core and a larger core. When observing the total area results it should be considered that this thesis uses 40nm technology, and Nakazato et al. [26] have not specified what technology was used. Lastly, the achieved FC is not compared because both works use outdated fault models.

The comparison in Table 5.12 is incomplete due to missing data in some of the works. However, it still provides some insights. The area comparison shows that the implemented DFT designs are relatively small compared to existing DFT designs. When comparing the increase of total gates, it must be noted that the increase in interconnect is not taken into account. Furthermore, the comparison of total area increase is incomplete due to the fact that this is technology independent, and most likely a different technology was used.

Table 5.12: Area Comparison with Other Works

Work	**Total Gates (increment)**	**Total Area (μm^2) (increment)**
Proposed (CS)	12608 (+1.68%)	38443 (+0.84%)
Proposed (ID)	12541 (+1.14%)	38371 (+0.65%)
Lai et al. (small core) [24]	1810 (+4.7%)	–
Lai et al. [24]	19165 (+1.6%)	–
Nakazato et al. (small core) [26]	–	15695 (+26.68%)
Nakazato et al. [26]	–	66613 (+13.49%)

5.6. Discussion

The FC results are slightly inaccurate because the total number of faults for both additions is slightly higher than the baseline number of faults due to the added hardware. Ideally, test patterns should also be generated and added to find these faults. It can also be seen that the CS DFT performs significantly better than the ID stage DFT for SAFs, while there is little difference for TDFs. This is because the baseline SAF coverage in the “cs_registers“ module is very low because observing these registers is a very convoluted process. The DFT addition enables the observation of these registers. The baseline SAF coverage in the modules where the ID stage DFT adds FC is already quite high so adding observability will not increase the FC much. The reason the TDF FC does increase, is because the place and timing of observing a fault is more essential for TDFs than for SAFs.

Although the DFTs seem to be more effective for TDFs than for SAFs, TDF FC is relatively low compared to other works. This is caused by the SBST generation framework which has a very low test pattern conversion ratio. The work by Chen et al. [8] reports about a 22 percentage point increase in TDF coverage compared to the SBST baseline of this work. However, Chen et al. have also not implemented a through justification engine. The main difference between the work by Chen et al. [8] and the proposed work, is that Chen et al. [8] have implemented a feedback loop from fault simulation back to CATPG. This indicates that the baseline SBST can likely be significantly improved by implementing such a feedback loop. The cost of this would be that the generation time would increase considerably.

5.7. Conclusion

In conclusion, it is shown that DFT additions that complement SBST can increase FC for very little area overhead. The hard to control and observe CS registers can be made partly functionally testable by adding DFT hardware. Other modules like the ID stage can also benefit from DFT hardware. To improve SBST testing and create comparable FC to full scan, multiple of these small targeted “functional“ DFT additions could be combined. These results show that SBST complemented by DFT potentially could posses similar testing capabilities to full scan for a smaller area overhead. This is possible by reusing as much hardware as possible, which is intrinsic to SBST.

Compared to state-of-the-art works the SAF results are very promising. However, to be feasible the program size would need to be decreased. Currently, all registers included in the partial scan CATPG setup are being written to and read from in every pattern. This accounts for about 75% of the test program. An easy solution would be checking which registers change during a test pattern and only interacting with those. The comparison of the TDF results show that the SBST generation needs to be further improved to match state-of-the-art coverage.

6 Conclusion

This chapter concludes this thesis. A summary of all previous chapters is provided in Section 6.1. Then, Section 6.2 draws conclusions from the thesis, and Section 6.3 discusses the limitations this thesis faced. Lastly, Section 6.4 contains recommendations for future work.

6.1. Summary

This section summarises the contents of the chapters that form the main body of this thesis.

6.1.1. Chapter 2

Chapter 2 (Background) provides the necessary background on digital IC design and test. It starts with an overview of combinational and sequential logic, including gate and flip-flop level constructs, and describes how transistors are structured. The chapter then covers the ASIC design flow: RTL design, synthesis, and place-and-route, and introduces modern processor microarchitectures, highlighting pipeline stages, and data and control flow. The chip life cycle from the design stage to the end-of-life is also discussed. Finally, it covers fault modelling and test generation techniques, from SAFs and delay fault models to ATPG algorithms and fault simulation, setting the stage for both the literature study and the combination of SBST with DFT.

6.1.2. Chapter 3

Chapter 3 (Silent Data Errors) surveys the phenomenon of SDEs and their implications for IC reliability. It paints a complete picture of SDEs, classifies underlying defect mechanisms, and reviews injection-based and field-observed SDE rates across microarchitecture-level components. It is concluded that marginal timing failures are a large cause of SDEs, and computation units are especially vulnerable. The chapter evaluates existing software solutions i.e. software test libraries, redundant execution. As well as hardware solutions such as continuous hardware verification and test. Then, the chapter focuses on SBST methods. It analyses state-of-the-art functional constraints extraction and pattern-to-program conversion methods. Additionally, it covers the limited work on SBST complemented by DFT. The chapter ends on the notion that SBST targeting TDFs needs to be improved, and a DFT addition is a promising way to achieve this.

6.1.3. Chapter 4

Chapter 4 (Software-Based Self-Test) details the implementation of an SBST framework for the CV32E40P RISC-V core. The program generation flow is presented. It starts by outlining an FCE method for a partial scan setup. Then, it covers the application of constraints in the ATPG tool Tessent, and the process of inserting scan chains into the CV32E40P core. Following this, the CATPG process is discussed, and the implementation of pattern-to-instruction conversion is presented. An experimental setup is established to generate test pattern files, and perform fault simulation using Tessent. The chapter concludes with results of synthesis, CATPG, and the SBST. The resulting SBST program achieved 76.57 % SAF FC, and 64.29% TDF FC on the CV32E40P core. The detailed FC results show that the EX stage, ID stage, and “cs_registers“ module suffer from low FC. This is improved with a DFT in the next chapter.

6.1.4. Chapter 5

Chapter 5 (DFT Design) explores the addition of DFT hardware to improve SBST FC. It begins with a DSE performed to justify the location of the DFT insertion. Then, a design method is chosen based on a trade-off analysis of criteria and candidate design options. A hardware implementation is described, detailing the integration of observe logic to the CV32E40P core. Results show a FC increase of 6.29 percentage points for SAFs and 2.4 percentage points for TDFs, at the cost of 0.84% area increase. These results indicate that DFTs which complement SBST can add FC capabilities for relatively small area overhead. Potentially, this could improve current SBST functionality to match full scan FC at a lower area overhead. However, it was also shown that the TDF FC results are relatively low compared to state-of-the-art works.

6.2. Conclusion

In conclusion, this thesis presented a methodology that determines avenues for a DFT addition to complement SBST programs. Moreover, multiple DFT designs are proposed and implemented on the CV32E40P RISC-V core. These additions are shown to increase the FC of the SBST program, at the cost of relatively little area overhead compared with results of state-of-the-art works. Based on these results it can be concluded that DFT complementing SBST can aid in the detection of hard-to-detect faults, and thereby increase FC. In contrast to the recent direction of state-of-the-art SBST works, which limit themselves to the available hardware on the core.

6.3. Limitations

The results of this work are limited by the following aspects.

- The **FCE** performed in this work was a limiting factor, as it is not comparable with state-of-the-art FCE methods. Because manual FCE takes great effort, and this is not the main focus of this work, it was greatly simplified. Moreover, automating FCE is almost exclusively done by works for which this is the main focus. The result of this is that the constraints set on the ATPG were not complete and thus lowered the test pattern conversion rate.
- The **SBST generation** framework provides lower results than current state-of-the-art works. Due to the main focus of the work being the DFT addition not the SBST program generation framework. This has limited the conclusions that could be made based on the results. Ideally, what should be proven is that a baseline SBST program with high FC can be complemented by a DFT, and result in the same FC increase and area overhead. Based on this a strong claim could be made that SBST complemented by a DFT is an alternative to full scan.
- The **ATPG tool** Tessent did not have all the capabilities that would have been ideal to have for SBST generation, and fault simulation. One of Tessent's limitations was that Tessent is only able to perform parallel fault simulation, and not sequential fault simulation. Additionally, Tessent is not able to handle more than one capture cycle per test pattern, even though its supporting tool STILverify does not indicate this as problematic. This has necessitated a fix to be able to read out all general purpose registers in one test pattern. Lastly, Tessent is not capable of proving if faults are untestable based on scan cell constraints. It is only able to find untestable faults based on input constraints. Resulting in potentially functionally untestable faults being in the fault list, and lower FC results.

6.4. Future Work

Recommendations for future work are listed below.

- To increase the FC of the SBST program, a feedback loop could be created from the fault simulation back to the CATPG. A list of undetected faults would need to be provided, together with the test patterns that were not successfully converted into instruction sequences. This feedback loop could be repeated until the desired FC is reached or no FC increase is observed any longer. This is one of the main reasons the SBST generation framework is not as effective as current state-of-the-art works.
- This work focused on adding observability, solving the issue of error masking. However, an analysis can be performed comparing how well extra observability, controllability, and observability

and controllability complement SBST programs. In particular, it would be interesting to see how this effects TDF FC.

- N-detect CATPG could be used in an attempt to find more functionally possible test patterns.
- One of the state-of-the-art pattern-to-program conversion or FCE methods could be replicated to improve the SBST program results.
- It should also be explored if combining multiple local DFT additions will provide the similar FC and area trade-off.
- No efforts have been made in this work to reduce SBST program size. A quick improvement would be detecting which registers change in each test pattern, and only writing to and reading from those registers. Currently, initializing registers and storing register values in memory accounts for about 75% of the program size.

References

[1] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits* (Frontiers in Electronic Testing), en, V. D. Agrawal, Ed. Boston, MA: Springer US, 2002, vol. 17, ISBN: 978-0-7923-7991-1. DOI: `10.1007/b117406`. [Online]. Available: `http://link.springer.com/10.1007/b117406` (visited on 05/09/2025).

[2] IEEE International Roadmap for Devices and Systems, *Executive Summary 2022*, en, 2022. DOI: `10.60627/C13Z-V363`. [Online]. Available: `https://irds.ieee.org/images/files/pdf/2022/2022IRDS_ES.pdf` (visited on 05/17/2025).

[3] E. J. Marinissen, H. Dattatraya Dixit, S. Blanton, *et al.*, “Silent data corruption: Test or reliability problem?” In *2024 IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2024, pp. 1–7. DOI: `10.1109/ETS61313.2024.10567773`. [Online]. Available: `https://ieeexplore.ieee.org/document/10567773` (visited on 09/29/2024).

[4] P. H. Hochschild, P. Turner, J. C. Mogul, *et al.*, “Cores that don’t count,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, Ann Arbor Michigan: ACM, Jun. 2021, pp. 9–16, ISBN: 978-1-4503-8438-4. DOI: `10.1145/3458336.3465297`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3458336.3465297` (visited on 09/11/2024).

[5] H. D. Dixit, S. Pendharkar, M. Beadon, *et al.*, *Silent data corruptions at scale*, Feb. 22, 2021. arXiv: `2102.11245[cs]`. [Online]. Available: `http://arxiv.org/abs/2102.11245` (visited on 09/11/2024).

[6] R. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep. 2005, Conference Name: IEEE Transactions on Device and Materials Reliability, ISSN: 1558-2574. DOI: `10.1109/TDMR.2005.853449`. [Online]. Available: `https://ieeexplore.ieee.org/document/1545891` (visited on 03/28/2025).

[7] Y. Zhu, S. Krishnan, K. Karanasos, *et al.*, “KEA: Tuning an exabyte-scale data infrastructure,” 2021, Publisher: arXiv Version Number: 1. DOI: `10.48550/ARXIV.2106.11445`. [Online]. Available: `https://arxiv.org/abs/2106.11445` (visited on 05/17/2025).

[8] K.-H. Chen, B.-Y. Yang, J.-R. Liang, H.-L. Chen, and J.-L. Huang, “Automatic test program generation for transition delay faults in pipelined processors,” in *2021 IEEE International Test Conference in Asia (ITC-Asia)*, ISSN: 2768-069X, Aug. 2021, pp. 1–6. DOI: `10.1109/ITC-Asia53059.2021.9808811`. [Online]. Available: `https://ieeexplore.ieee.org/document/9808811` (visited on 11/29/2024).

[9] M. Tehranipoor, K. Peng, and K. Chakrabarty, *Test and Diagnosis for Small-Delay Defects*. New York, NY: Springer, 2012, ISBN: 978-1-4419-8296-4. DOI: `10.1007/978-1-4419-8297-1`. [Online]. Available: `https://link.springer.com/10.1007/978-1-4419-8297-1` (visited on 12/09/2024).

[10] A. D. Singh, “Silent error corruption: The new reliability and test challenge,” in *2023 IEEE 24th Latin American Test Symposium (LATS)*, ISSN: 2373-0862, Mar. 2023, pp. 1–2. DOI: `10.1109/LATS58125.2023.10154487`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10154487` (visited on 09/29/2024).

[11] H. Cheng, C.-J. Li, H.-L. Chen, and J.-L. Huang, “BDD-based self-test program generation for processor cores,” in *2023 IEEE International Test Conference in Asia (ITC-Asia)*, ISSN: 2768-069X, Sep. 2023, pp. 1–6. DOI: `10.1109/ITC-Asia58802.2023.10301167`. [Online]. Available: `https://ieeexplore.ieee.org/document/10301167` (visited on 12/12/2024).

[12] L.-A. Kuo and J.-L. Huang, “Branch-aware self-test program generation for processor cores,” in *2024 International VLSI Symposium on Technology, Systems and Applications (VLSI TSA)*, Apr. 2024, pp. 1–4. DOI: `10.1109/VLSITSA60681.2024.10546455`. [Online]. Available: `https://ieeexplore.ieee.org/document/10546455` (visited on 12/12/2024).

[13] T. Faller, N. I. Deligiannis, M. Schwörer, M. S. Reorda, and B. Becker, "Constraint-based automatic SBST generation for RISC-v processor families," in *2023 IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2023, pp. 1–6. DOI: 10.1109/ETS56758.2023.10174156. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10174156 (visited on 12/13/2024).

[14] L. Anghel, R. Cantoro, R. Masante, M. Portolan, S. Sartoni, and M. S. Reorda, "Self-test library generation for in-field test of path delay faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 4246–4259, Nov. 2023, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/TCAD.2023.3268210. [Online]. Available: https://ieeexplore.ieee.org/document/10104111 (visited on 12/12/2024).

[15] E. Kaja, N. Gerlin, J. A. Halabi, *et al.*, "An automated and effective approach for SBST generation targeting RISC-v CPUs," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, ISSN: 2765-933X, Oct. 2024, pp. 1–4. DOI: 10.1109/DFT63277.2024.10753552. [Online]. Available: https://ieeexplore.ieee.org/document/10753552 (visited on 12/12/2024).

[16] J. Seo and H. Cho, "Generating efficient instruction sequence for software-based self-testing of processor cores using reinforcement learning," *IEEE Access*, vol. 12, pp. 189 288–189 296, 2024, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3516389. [Online]. Available: https://ieeexplore.ieee.org/document/10794784 (visited on 02/03/2025).

[17] R. Cantoro, P. Girard, R. Masante, S. Sartoni, M. S. Reorda, and A. Virazel, "Self-test libraries analysis for pipelined processors transition fault coverage improvement," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, ISSN: 1942-9401, Jun. 2021, pp. 1–4. DOI: 10.1109/IOLTS52814.2021.9486711. [Online]. Available: https://ieeexplore.ieee.org/document/9486711 (visited on 01/14/2025).

[18] Y. Zhang, K. Chakrabarty, Z. Peng, *et al.*, "Software-based self-testing using bounded model checking for out-of-order superscalar processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 3, pp. 714–727, Mar. 2020, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2890695. [Online]. Available: https://ieeexplore.ieee.org/document/8599062 (visited on 01/10/2025).

[19] A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, "Implementation-Independent Functional Test for Transition Delay Faults in Microprocessors," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Aug. 2020, pp. 646–650. DOI: 10.1109/DSD51259.2020.00105. [Online]. Available: https://ieeexplore.ieee.org/document/9217857 (visited on 11/29/2024).

[20] 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*, vol. 30, no. 11, pp. 1677–1690, Nov. 2022, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2022.3186946. [Online]. Available: https://ieeexplore.ieee.org/document/9834321 (visited on 01/15/2025).

[21] Y. Zhang, H. Li, and X. Li, "Software-based self-testing of processors using expanded instructions," in *2010 19th IEEE Asian Test Symposium*, ISSN: 2377-5386, Dec. 2010, pp. 415–420. DOI: 10.1109/ATS.2010.77. [Online]. Available: https://ieeexplore.ieee.org/document/5692282 (visited on 12/12/2024).

[22] Y. Zhang, H. Li, and X. Li, "Automatic test program generation using executing-trace-based constraint extraction for embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 7, pp. 1220–1233, Jul. 2013, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2012.2208130. [Online]. Available: https://ieeexplore.ieee.org/document/6265421 (visited on 01/15/2025).

[23] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker, “A Flexible Framework for the Automatic Generation of SBST Programs,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3055–3066, Oct. 2016, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2016.2538800. [Online]. Available: https://ieeexplore.ieee.org/document/7440859 (visited on 01/15/2025).

[24] W.-C. Lai and K.-T. Cheng, “Instruction-level DfT for testing processor and IP cores in system-on-a-chip,” in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, ISSN: 0738-100X, Jun. 2001, pp. 59–64. DOI: 10.1145/378239.378282. [Online]. Available: https://ieeexplore.ieee.org/document/935477/citations?tabFilter=papers#citations (visited on 02/03/2025).

[25] Z. Navabi, *VHDL : analysis and modeling of digital systems*, eng. New York ; London : McGraw-Hill, 1996, ISBN: 978-0-07-112732-5. [Online]. Available: http://archive.org/details/vhdlanalysismode0000nava (visited on 05/02/2025).

[26] M. Nakazato, S. Ohtake, M. Inoue, and H. Fujiwara, “Design for testability of software-based self-test for processors,” in *2006 15th Asian Test Symposium*, ISSN: 2377-5386, Nov. 2006, pp. 375–380. DOI: 10.1109/ATS.2006.260958. [Online]. Available: https://ieeexplore.ieee.org/document/4030794/citations?tabFilter=papers#citations (visited on 02/03/2025).

[27] I. Pomeranz, “Functional Design-for-Testability for Functional Test Sequences,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 12, pp. 4852–4859, Dec. 2024, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/TCAD.2024.3396654. [Online]. Available: https://ieeexplore.ieee.org/document/10518130 (visited on 02/02/2025).

[28] G. Haley, *Dft at the leading edge*. [Online]. Available: https://semiengineering.com/dft-at-the-leading-edge/ (visited on 05/17/2025).

[29] D. Harris and S. Harris, *Digital Design and Computer Architecture, Second Edition*, 2nd ed. Morgan Kaufmann, 2012, ISBN: 978-0-12-394424-5. (visited on 05/07/2025).

[30] J. M. Rabaey, *DIGITAL INTEGRATED CIRCUITS A DESIGN PERSPECTIVE* (PRENTICE HALL ELECTRONICS AND VLSI SERIES 1), en. Prentice Hall, 2003, p. 514.

[31] V. Universe, *Complete ASIC Design flow 2021 – VLSI UNIVERSE*. [Online]. Available: https://www.vlsiuniverse.com/complete-asic-design-flow/ (visited on 05/07/2025).

[32] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*, 6th ed. Morgan Kaufmann, 2021, ISBN: 978-0-12-822674-2. (visited on 05/07/2025).

[33] *Risc-v*. [Online]. Available: https://semiengineering.com/knowledge_centers/compute-architectures/instruction-set-architecture-isa/risc-v/ (visited on 06/19/2025).

[34] G. Gielen, P. De Wit, E. Maricau, *et al.*, “Emerging Yield and Reliability Challenges in Nanometer CMOS Technologies,” in *2008 Design, Automation and Test in Europe*, ISSN: 1558-1101, Mar. 2008, pp. 1322–1327. DOI: 10.1109/DATE.2008.4484862. [Online]. Available: https://ieeexplore.ieee.org/document/4484862 (visited on 04/30/2025).

[35] D. A. Neamen, *Semiconductor physics and devices: basic principles*, en, 4. ed. New York, NY: McGraw-Hill, 2012, ISBN: 978-0-07-352958-5.

[36] N. K. Jha and S. Gupta, “Testing of digital systems,”

[37] P. Peercy, “SIA updates national technology roadmap for semiconductors,” *MRS Bulletin*, vol. 22, no. 11, pp. 29–29, Nov. 1997, ISSN: 0883-7694, 1938-1425. DOI: 10.1557/S0883769400034370. [Online]. Available: http://link.springer.com/10.1557/S0883769400034370 (visited on 06/22/2025).

[38] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, “Transition fault simulation,” *IEEE Design & Test of Computers*, vol. 4, no. 2, pp. 32–38, Apr. 1987, Conference Name: IEEE Design & Test of Computers, ISSN: 1558-1918. DOI: 10.1109/MDT.1987.295104. [Online]. Available: https://ieeexplore.ieee.org/document/4069962 (visited on 12/13/2024).

[39] G. L. Smith, “Model for delay faults based upon paths.,” in *ITC*, vol. 85, Citeseer, 1985, pp. 342–349. [Online]. Available: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4bd3e334fa5d2c6a817130e3c8cee5a69a6bc3e6` (visited on 12/13/2024).

[40] S. Patil and J. Savir, “Skewed-load transition test: Part II, coverage,” in *Proceedings International Test Conference 1992*, ISSN: 1089-3539, Sep. 1992, pp. 714–. DOI: `10.1109/TEST.1992.527893`. [Online]. Available: `https://ieeexplore.ieee.org/document/527893` (visited on 01/02/2025).

[41] J. Savir and S. Patil, “On broad-side delay test,” in *Proceedings of IEEE VLSI Test Symposium*, Apr. 1994, pp. 284–290. DOI: `10.1109/VTEST.1994.292299`. [Online]. Available: `https://ieeexplore.ieee.org/document/292299` (visited on 01/02/2025).

[42] S. Vemula, “SCAN BASED DELAY TESTING,” 2005. [Online]. Available: `https://www.semanticscholar.org/paper/SCAN-BASED-DELAY-TESTING-Vemula/11215a6ba775e1c5e8f6c56174dbfd4b4891a485` (visited on 01/05/2025).

[43] B. Dervisoglu and G. Stong, “DESIGN FOR TESTABILITY USING SCANPATH TECHNIQUES FOR PATH-DELAY TEST AND MEASUREMENT,” in *1991, Proceedings. International Test Conference*, ISSN: 1089-3539, Oct. 1991, pp. 365–. DOI: `10.1109/TEST.1991.519696`. [Online]. Available: `https://ieeexplore.ieee.org/document/519696` (visited on 01/02/2025).

[44] C. J. Lin and S. Reddy, “On delay fault testing in logic circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 694–703, Sep. 1987, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: `10.1109/TCAD.1987.1270315`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/1270315` (visited on 12/13/2024).

[45] I. Pomeranz, “Weak and strong non-robust tests for functionally possible path delay faults,” *IEEE Access*, vol. 12, pp. 156 651–156 661, 2024, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: `10.1109/ACCESS.2024.3486057`. [Online]. Available: `https://ieeexplore.ieee.org/document/10734085` (visited on 11/27/2024).

[46] B. Moyer, *Error correction code (ecc)*, Accessed: 2025-03-28. [Online]. Available: `https://semiengineering.com/knowledge_centers/memory/error-correction-code-ecc/`.

[47] A. Singh, S. Chakravarty, G. Papadimitriou, and D. Gizopoulos, “Silent data errors: Sources, detection, and modeling,” in *2023 IEEE 41st VLSI Test Symposium (VTS)*, ISSN: 2375-1053, Apr. 2023, pp. 1–12. DOI: `10.1109/VTS56346.2023.10139970`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10139970?casa_token=sAuEGbks5QwAAAAA:i2Dz4eIfDFgaHzIAWrogmNAALwmbc_cB4V3rwOJ_GhQ6bIVG-GpfZQbmgDKRB_KJAvAKcT9p` (visited on 09/29/2024).

[48] J. Vanian, *Samsung Dethrones Intel As World’s Biggest Chip Maker*, en. [Online]. Available: `https://fortune.com/2017/07/27/samsung-intel-chip-semiconductor/` (visited on 04/03/2025).

[49] M. Shamsa and D. Lerner, “Defect mechanisms responsible for silent data errors,” in *2024 IEEE International Reliability Physics Symposium (IRPS)*, ISSN: 1938-1891, Apr. 2024, pp. 1–5. DOI: `10.1109/IRPS48228.2024.10529392`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10529392?casa_token=WD2j586IOSIAAAAA:h6At6wc0rj4YMaquems6JMce7kly93jIGlj5bvQhX5GGUdWDJ3whw8DjUltxFmkMTdRpWBNc` (visited on 09/29/2024).

[50] *Meeting of the test technical committee (ttc) of the ieee electronics packaging society (eps)*, Oct. 2023.

[51] A. D. Singh, “Silent data corruption from timing marginalities due to process variations,” in *2024 IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2024, pp. 1–7. DOI: `10.1109/ETS61313.2024.10567054`. [Online]. Available: `https://ieeexplore.ieee.org/document/10567054` (visited on 09/29/2024).

[52] G. Papadimitriou and D. Gizopoulos, “Silent data corruptions: Microarchitectural perspectives,” *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, Nov. 2023, Conference Name: IEEE Transactions on Computers, ISSN: 1557-9956. DOI: `10.1109/TC.2023.3285094`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10151692?casa_token=ssK3RRaViOgAAAAA:xcwkBa6KySA6Ddb-_PMJ9NOzQum5GkWybOMNAU_X-vDQ3Hs9A2wpVBXihznXBF1ZJxw1Kult` (visited on 09/29/2024).

[53] D. Gizopoulos, G. Papadimitriou, O. Chatzopoulos, N. Karystinos, H. D. Dixit, and S. Sankar, "Silent data corruptions in computing systems: Early predictions and large-scale measurements," in *2024 IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2024, pp. 1–10. DOI: 10.1109/ETS61313.2024.10567770. [Online]. Available: https://ieeexplore.ieee.org/document/10567770 (visited on 09/29/2024).

[54] N. Binkert, B. Beckmann, G. Black, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 31, 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: https://dl.acm.org/doi/10.1145/2024716.2024718 (visited on 11/21/2024).

[55] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 69–78. DOI: 10.1109/ISPASS.2016.7482075. [Online]. Available: https://ieeexplore.ieee.org/document/7482075 (visited on 05/12/2025).

[56] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Automation & Test in Europe Conference & Exhibition 2009 Design*, ISSN: 1558-1101, Apr. 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716. [Online]. Available: https://ieeexplore.ieee.org/document/5090716 (visited on 11/13/2024).

[57] G. Papadimitriou and D. Gizopoulos, "Characterizing soft error vulnerability of CPUs across compiler optimizations and microarchitectures," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2021, pp. 113–124. DOI: 10.1109/IISWC53511.2021.00021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9668302 (visited on 09/29/2024).

[58] D. Gizopoulos, G. Papadimitriou, and O. Chatzopoulos, "Estimating the failures and silent errors rates of CPUs across ISAs and microarchitectures," in *2023 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Oct. 2023, pp. 377–382. DOI: 10.1109/ITC51656.2023.00056. [Online]. Available: https://ieeexplore.ieee.org/document/10351088 (visited on 11/13/2024).

[59] O. Chatzopoulos, G. Papadimitriou, V. Karakostas, and D. Gizopoulos, "Gem5-MARVEL: microarchitecture-level Resilience Analysis of Heterogeneous SoC Architectures," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, ISSN: 2378-203X, Mar. 2024, pp. 543–559. DOI: 10.1109/HPCA57654.2024.00047. [Online]. Available: https://ieeexplore.ieee.org/document/10476486 (visited on 05/12/2025).

[60] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, ISSN: 2378-203X, Feb. 2009, pp. 105–116. DOI: 10.1109/HPCA.2009.4798242. [Online]. Available: https://ieeexplore.ieee.org/document/4798242 (visited on 11/28/2024).

[61] H. Cho, "Impact of microarchitectural differences of RISC-v processor cores on soft error effects," *IEEE Access*, vol. 6, pp. 41 302–41 313, 2018, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2858773. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8418379 (visited on 11/28/2024).

[62] B. Wibowo, A. Agrawal, and J. Tuck, "Characterizing the impact of soft errors across microarchitectural structures and implications for predictability," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2017, pp. 250–260. DOI: 10.1109/IISWC.2017.8167782. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8167782 (visited on 11/28/2024).

[63] D. P. Lerner, B. Inkley, S. H. Sahasrabudhe, E. Hansen, L. D. R. Munoz, and A. v. de Ven, "Optimization of tests for managing silicon defects in data centers," in *2022 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Sep. 2022, pp. 578–582. DOI: 10.1109/ITC50671.2022.00076. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9983919?casa_token=ZohgUDmjkG4AAAAA:124fyRjqACo6od0Oxqp6mMWA91n2tdmL7mXBUuleRv6rb5EhPZy5a9D9VQdC87Qh0a91OGkP (visited on 09/29/2024).

[64] Y. Huang, S. Guo, S. Di, G. Li, and F. Cappello, “Mitigating silent data corruptions in HPC applications across multiple program inputs,” in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, ISSN: 2167-4337, Nov. 2022, pp. 1–14. DOI: 10.1109/SC41404.2022.00022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10046091?casa_token=Qa6sjEkGDCgAAAAA:3ePOA8ZHOOX_yIhM176gQqhiMqdEJNXOalU-bcgWqvO7TFFHaoS2GaMxFDnrnxQ6MDsTjHef (visited on 09/29/2024).

[65] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, “Modeling Soft-Error Propagation in Programs,” in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ISSN: 2158-3927, Jun. 2018, pp. 27–38. DOI: 10.1109/DSN.2018.00016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8416468 (visited on 11/28/2024).

[66] A. Pan, J. W. Tschanz, and S. Kundu, “A low cost scheme for reducing silent data corruption in large arithmetic circuits,” in *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, ISSN: 2377-7966, Oct. 2008, pp. 343–351. DOI: 10.1109/DFT.2008.42. [Online]. Available: https://ieeexplore.ieee.org/document/4641190 (visited on 11/12/2024).

[67] M. Nicolaidis, R. Duarte, S. Manich, and J. Figueras, “Fault-secure parity prediction arithmetic operators,” *IEEE Design & Test of Computers*, vol. 14, no. 2, pp. 60–71, Apr. 1997, ISSN: 1558-1918. DOI: 10.1109/54.587743. [Online]. Available: https://ieeexplore.ieee.org/document/587743 (visited on 06/22/2025).

[68] D. Marienfeld, E. Sogomonyan, V. Ocheretnij, and M. Gossel, “New self-checking output-duplicated booth multiplier with high fault coverage for soft errors,” in *14th Asian Test Symposium (ATS'05)*, ISSN: 2377-5386, Dec. 2005, pp. 76–81. DOI: 10.1109/ATS.2005.80. [Online]. Available: https://ieeexplore.ieee.org/document/1575410 (visited on 06/22/2025).

[69] A. D. Singh, “Understanding vmin failures for improved testing of timing marginalities,” in *2022 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Sep. 2022, pp. 372–381. DOI: 10.1109/ITC50671.2022.00046. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9983900?casa_token=GwUaGS3GlDkAAAAA:MSXyCeqQuwQ4p1tmSLzYGGlXrjmP1hSuxlay4_E7z8zpqS_saYWgJtmbIum-ZHsPxR4xDIoi (visited on 09/29/2024).

[70] S. Natarajan, C. S. Oak, V. Kakollu, *et al.*, “Effectiveness of timing-aware scan tests in targeting marginal failures and silent data errors in a data center processor,” in *2024 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Nov. 2024, pp. 253–260. DOI: 10.1109/ITC51657.2024.00045. [Online]. Available: https://ieeexplore.ieee.org/document/10766694 (visited on 12/04/2024).

[71] M. Sauer, I. Polian, M. E. Imhof, *et al.*, “Variation-aware deterministic ATPG,” in *2014 19th IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2014, pp. 1–6. DOI: 10.1109/ETS.2014.6847806. [Online]. Available: https://ieeexplore.ieee.org/document/6847806 (visited on 12/05/2024).

[72] H. Jafarzadeh, F. Klemme, J. D. Reimer, *et al.*, “Robust pattern generation for small delay faults under process variations,” in *2023 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Oct. 2023, pp. 111–116. DOI: 10.1109/ITC51656.2023.00026. [Online]. Available: https://ieeexplore.ieee.org/document/10351052 (visited on 11/11/2024).

[73] H. Amrouch, G. Pahwa, A. D. Gaidhane, *et al.*, “Impact of variability on processor performance in negative capacitance FinFET technology,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 9, pp. 3127–3137, Sep. 2020, Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, ISSN: 1558-0806. DOI: 10.1109/TCSI.2020.2990672. [Online]. Available: https://ieeexplore.ieee.org/document/9090811 (visited on 01/02/2025).

[74] I. Pomeranz and S. Reddy, “On n-detection test sets and variable n-detection test sets for transition faults,” in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, ISSN: 1093-0167, Apr. 1999, pp. 173–180. DOI: 10.1109/VTEST.1999.766662. [Online]. Available: https://ieeexplore.ieee.org/document/766662 (visited on 06/22/2025).

[75] H. Jafarzadeh, F. Klemme, H. Amrouch, S. Hellebrand, and H.-J. Wunderlich, “Vmin testing under variations: Defect vs. fault coverage,” in *2024 IEEE 25th Latin American Test Symposium (LATS)*, ISSN: 2373-0862, Apr. 2024, pp. 1–6. DOI: 10.1109/LATS62223.2024.10534608. [Online]. Available: https://ieeexplore.ieee.org/document/10534608 (visited on 11/28/2024).

[76] H. Jafarzadeh, F. Klemme, H. Amrouch, S. Hellebrand, and H.-J. Wunderlich, “Time and space optimized storage-based BIST under multiple voltages and variations,” in *2024 IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2024, pp. 1–6. DOI: 10.1109/ETS61313.2024.10567295. [Online]. Available: https://ieeexplore.ieee.org/document/10567295 (visited on 11/29/2024).

[77] H. Jafarzadeh, F. Klemme, J. D. Reimer, H. Amrouch, S. Hellebrand, and H.-J. Wunderlich, “Minimizing PVT-variability by exploiting the zero temperature coefficient (ZTC) for robust delay fault testing,” in *2024 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Nov. 2024, pp. 26–30. DOI: 10.1109/ITC51657.2024.00013. [Online]. Available: https://ieeexplore.ieee.org/document/10766727 (visited on 12/04/2024).

[78] L. Chen and S. Dey, “Software-based self-testing methodology for processor cores,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369–380, Mar. 2001, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/43.913755. [Online]. Available: https://ieeexplore.ieee.org/document/913755 (visited on 01/05/2025).

[79] R. Cantoro, F. Garau, P. Girard, *et al.*, “Effective techniques for automatically improving the transition delay fault coverage of self-test libraries,” in *2022 IEEE European Test Symposium (ETS)*, ISSN: 1558-1780, May 2022, pp. 1–2. DOI: 10.1109/ETS54262.2022.9810392. [Online]. Available: https://ieeexplore.ieee.org/document/9810392 (visited on 01/14/2025).

[80] M. Bartolomucci, N. I. Deligiannis, R. Cantoro, and M. S. Reorda, “Fault grading techniques for evaluating software-based self-test with respect to small delay defects,” in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, ISSN: 1942-9401, Jul. 2024, pp. 1–6. DOI: 10.1109/IOLTS60994.2024.10616077. [Online]. Available: https://ieeexplore.ieee.org/document/10616077 (visited on 11/28/2024).

[81] J. E. Rodriguez Condia, F. A. Da Silva, A. Ç. Bağbaga, *et al.*, “Using STLs for effective in-field test of GPUs,” *IEEE Design & Test*, vol. 40, no. 2, pp. 109–117, Apr. 2023, ISSN: 2168-2356, 2168-2364. DOI: 10.1109/MDAT.2022.3188573. [Online]. Available: https://ieeexplore.ieee.org/document/9815288/ (visited on 01/08/2025).

[82] J.-D. Guerrero-Balaguera, J. E. R. Condia, and M. S. Reorda, “On the functional test of special function units in GPUs,” in *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, ISSN: 2473-2117, Apr. 2021, pp. 81–86. DOI: 10.1109/DDECS52668.2021.9417025. [Online]. Available: https://ieeexplore.ieee.org/document/9417025 (visited on 01/10/2025).

[83] B. Fuller, *Are you ready for ai?* [Online]. Available: https://semiengineering.com/are-you-ready-for-ai/ (visited on 06/19/2025).

[84] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, “Microprocessor software-based self-testing,” *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, May 2010, Conference Name: IEEE Design & Test of Computers, ISSN: 1558-1918. DOI: 10.1109/MDT.2010.5. [Online]. Available: https://ieeexplore.ieee.org/document/5396292 (visited on 12/12/2024).

[85] P. Wohl and J. Waicukauski, “Test generation for ultra-large circuits using ATPG constraints and test-pattern templates,” in *Proceedings International Test Conference 1996. Test and Design Validity*, ISSN: 1089-3539, Oct. 1996, pp. 13–20. DOI: 10.1109/TEST.1996.556938. [Online]. Available: https://ieeexplore.ieee.org/document/556938 (visited on 01/15/2025).

[86] *Minimips*, Accessed: 2025-05-02. [Online]. Available: https://opencores.org/projects/minimips.

[87] A. Riefert, L. Ciganda, M. Sauer, P. Bernardi, M. S. Reorda, and B. Becker, "An effective approach to automatic functional processor test generation for small-delay faults," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.140. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6800341 (visited on 01/15/2025).

[88] C.-Y. Chen and J.-L. Huang, "Reinforcement-learning-based test program generation for software-based self-test," in *2019 IEEE 28th Asian Test Symposium (ATS)*, ISSN: 2377-5386, Dec. 2019, pp. 73–735. DOI: 10.1109/ATS47505.2019.00013. [Online]. Available: https://ieeexplore.ieee.org/document/8949401 (visited on 01/05/2025).

[89] T. Angel, *Antares MIPS32*, https://github.com/AngelTerrones/Antares, [Online], 2015.

[90] M. Gautschi, P. D. Schiavone, A. Traber, *et al.*, *Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices*, Feb. 2017. DOI: 10.1109/TVLSI.2017.2654506. [Online]. Available: https://ieeexplore.ieee.org/document/7864441 (visited on 05/16/2025).

[91] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: An ultra-low-power pulpissimo soc in 22nm fdx," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145.

[92] *Pulp-platform/pulpissimo*, original-date: 2018-02-09T10:24:02Z, May 2025. [Online]. Available: https://github.com/pulp-platform/pulpissimo (visited on 05/16/2025).

[93] *Cv32e40p*, Accessed: 2025-05-16. [Online]. Available: https://github.com/pulp-platform/cv32e40p.

[94] Siemens EDA, *Tessent™ shell reference manual*, Software Version 2021.3, Document Revision 24, https://eda.sw.siemens.com/, Siemens EDA, 2021.

[95] "IEEE Approved Draft Standard Test Interface Language (STIL) for Digital Test Vector Data," *IEEE P1450/D2, October 2023*, pp. 1–141, Dec. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10273846 (visited on 05/16/2025).

[96] M. Tuna, M. Benabdenbi, and Laboratoire, "Software based self-test of register files in risc processor cores using march algorithms," Mar. 2006.

[97] *Cv32e40p docs control and status registers*. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40s-user-manual/en/latest/control_status_registers.html (visited on 06/17/2025).

A
Status and Control Registers

Table A.1 shows the control and status registers of the cv32e40p [90] core. For each register its size is given, and a description. Besides that it is indicated if the register is writable in functional mode. If this is not the case the core would need to enter debug mode to access the register.

Table A.1: Control Status Register Overview of the CV32E40P [97]

Register name	Size (index)	Description	Writable
Debug Control and Status	[31:0]	Debug registers, track activity of certain control sequences.	No
Debug PC	[31:1]	Contains the virtual address of the next instruction in debug mode.	No
Debug Scratch register 0	[31:0]	General-purpose debug scratch register.	No
Debug Scratch register 1	[31:0]	General-purpose debug scratch register.	No
Trigger Data register 1	[2]	Execute bit, enables matching on instruction address.	No
Trigger Data register 2	[31:0]	Stores the instruction address to match against for a breakpoint trigger.	No
Machine Cause	[5:0]	Shows if an exception is an interrupt and its corresponding code.	Yes
Machine Counter-Inhibit register	[3:2], [0]	Controls which performance counters are active.	Yes
Machine Exception PC	[31:1]	Machine exception program counter.	Yes
Machine Performance Monitoring Counter 0	[63:0]	Hardware performance monitor counter 0.	Yes
Machine Performance Monitoring Counter 2	[63:0]	Hardware performance monitor counter 2.	Yes
Machine Performance Monitoring Counter 3	[63:0]	Hardware performance monitor counter 3.	Yes
Machine Performance Monitoring Event Selector 3	[15:0]	Event selector for Machine Performance Monitoring Counter 3	Yes
Machine Interrupt Enable register	[31:16], [11], [7], [3]	Machine interrupt enable bits.	Yes
Machine Scratch	[31:0]	OS or Firmware can use this to store temporary data across trap handling.	Yes
Machine Status	[7], [3]	Machine interrupt enable and prior enable bits.	Yes
Machine Trap-Vector Base Address	[31:8], [0]	Base address of trap vector, and trap vector mode.	Yes

B

Bugged Design Space Exploration Results

Figure B.1 shows the results of the CATPG run with additional observable scan cells, instead of additional observe points. In theory, the two mentioned options add the same functionality to the flip-flops, and therefore should give similar FC results. However, this is not the case as can be seen in Figure B.1. This is probably caused by some internal mechanism of Tessent.

The most feasible explanation is that this is caused by Tessent internally treating the observable-only scan cells as controllable when trying to find suitable patterns. Resulting in ATPG aborts because the values being loaded into these scan cells are not actually applied to the corresponding flip-flops. Evidence for this is seen in the fault analysis report, by an increase in ATPG aborts caused by an increase of uncontrolled faults. This in combination with the fact that for these simulations a maximum scan chain length of one is used, has likely confused Tessent and caused bugged the results.

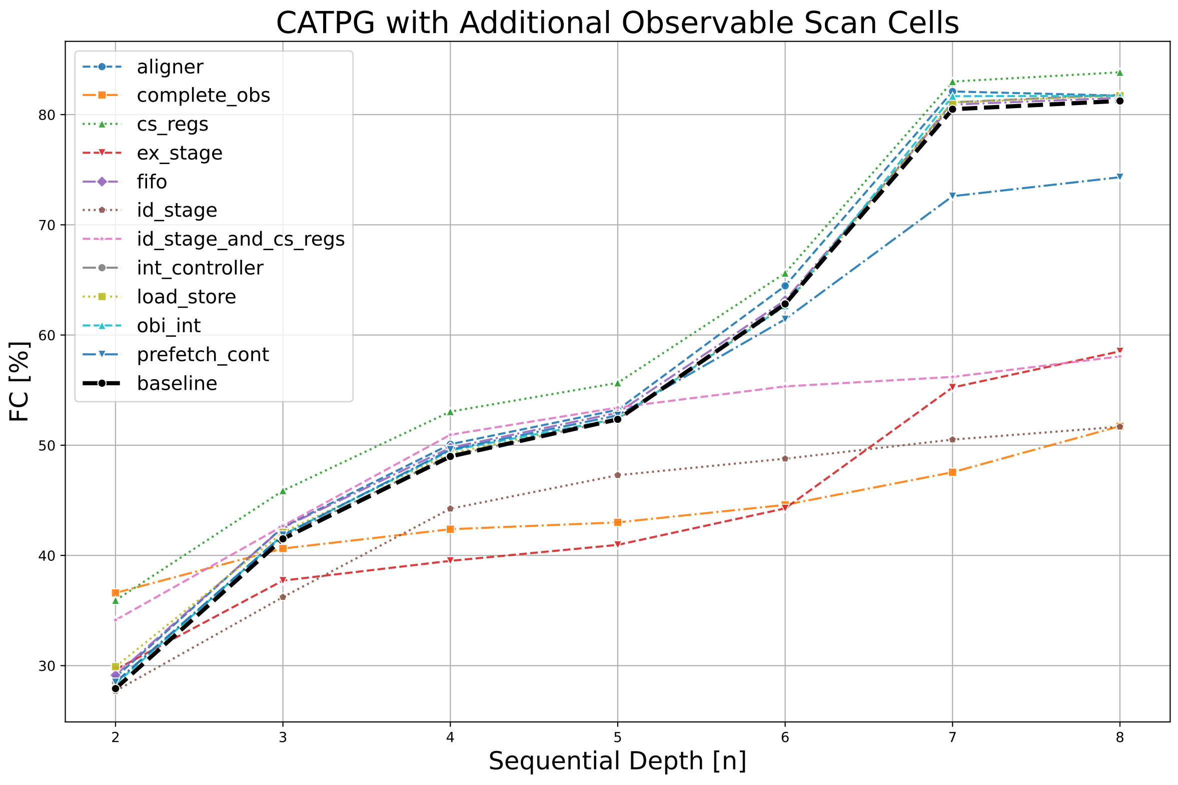


Figure B.1: TDF Results with Additional Observable Scan Cells