Institute of Computer Architecture and Computer Engineering

University of Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Bachelorarbeit

Fault Emulation of Reconfigurable Scan Networks

Denis Schwachhofer

Course of Study:	B. Sc. Informatik
Examiner:	Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Supervisor:	Zahra Paria Najafi Haghi
Commenced:	October 1, 2017
Completed:	April 3 2018

Abstract

At around their standardization by the IEEE the interest on Reconfigurable Scan Networks (RSNs) by research and industry sparked. The testing of RSNs also raises new challenges. To analyze and cope with these challenges researchers are required to perform fault simulation. And the industry incorporated RSNs into their designs and need to test them to which also requires fault simulation. But the runtime of it is significantly high due to the RSNs' structure. This thesis introduces a platform for fault emulation of RSNs and analyzes its feasibility. The speedup compared to fault simulation is presented and advantages, limitations and possible optimizations are evaluated and discussed.

Kurzfassung

Etwa zur Zeit der Standardisierung durch die IEEE wurde das Interesse von Industrie und Forschung an Rekonfigurierbaren Scan Netzwerken (RSN) entfacht. Das Testen von RSNs wirft neue Herausforderungen auf. Um diese Herausforderungen analysieren und bewältigen zu können sind Forscher auf Fehlersimulation angewiesen. Die Industrie wendet RSNs in ihren Designs an und muss diese auch testen. Daher sind auch sie auf Fehlersimulation angewiesen. Allerdings ist die Laufzeit dessen signifikant hoch aufgrund der Struktur von RSNs. Diese Arbeit stellt eine Plattform für die Fehleremulation von RSNs vor und analysiert ihre Durchführbarkeit. Der Geschwindigkeitsvorteil im Vergleich zur Fehlersimulation wird dargestellt und Vor- und Nachteile und weitere Optimierungen evaluiert und besprochen.

Contents

1 Introduction 13

2 Basics of Hardware Testing 15
2.1 Testing . . . 15
2.2 Fault Simulation . . . 17
2.3 Fault Emulation . . . 18
2.4 Linear Feedback Shift Register . . . 20
2.5 Scan Design . . . 22

3 Reconfigurable Scan Networks 25
3.1 Structure . . . 25
3.2 Scan Segment . . . 25
3.3 Operation . . . 27
3.4 Reconfiguration . . . 27

4 Related Work 31
4.1 RSNs . . . 31
4.2 Fault Emulation . . . 31

5 Implementation 33
5.1 Tools . . . 33
5.2 Structure . . . 34
5.3 Algorithm . . . 38

6 Results 41

7 Conclusion 45
7.1 Summary . . . 45
7.2 Future Work . . . 45

Bibliography 47

List of Figures

2.1 General model of a circuit (with optional ports SI, SO and TM) 17
2.2 Examples of injected stuck-at faults. 19
2.3 Two types of LFSRs 21
2.4 Level sensitive scan cell 22

3.1 Example of a RSN represented as a DAG 26
3.2 A simplified read-write scan segment without disable signals 26
3.3 A SIB with delay bit 28
3.4 Example of a RSN using SIBs 28
3.5 Example of a RSN using multiplexer 29

5.1 Tool Flow used in this thesis 35
5.2 Structure of FEPfR 36

List of Tables

5.1 Structure of pattern input 37

6.1 RSNs used in this thesis 41
6.2 Runtime and fault coverage of fault simulation and emulation 42
6.3 Speedup of fault emulation to fault simulation 43

List of Algorithms

5.1 PFR 37
5.2 TPG 38
5.3 ORA 39

List of Abbreviations

ASIC Application Specific Integrated Circuit. 13

ATPG Automatic Test Pattern Generation. 16

BIST Built-In Self-Test. 18

CUT Circuit Under Test. 16

DAG Directed, Acyclic graph. 25

DfT Design for Testability. 13

FEPfR Fault Emulation Platform for RSNs. 33

FIFO First-In First-Out. 34

FPGA Field Programmable Gate Array. 18

IC Integrated Circuit. 15

ICL Instrument Connectivity Language. 26

LFSR Linear Feedback Shift Register. 13

LUT Look Up Table. 31

MISR Multiple-Input Signature Register. 21

ORA Output Response Analyzer. 34

PFR Pattern Frame Recognizer. 34

PI Primary Input. 16

PO Primary Output. 16

PPI Pseudo Primary Input. 16

PPO Pseudo Primary Output. 16

PPSFP Parallel Pattern Single Fault Propagation. 17

RSN Reconfigurable Scan Network. 3, 13

SIB Segment Insertion Bit. 25

SISR Single-Input Signature Register. 21

SoC System-on-Chip. 13

SPPFP Single Pattern Parallel Fault Propagation. 17

STIL Standard Interface Test Language. 34

TAP Test Access Port. 25

TPG Test Pattern Generator. 34

UART Universal Asyncronous Receiver Transmitter. 33

VCD Value Change Dump. 34

WGL Waveform Generation Language. 34

1 Introduction

Two years before the standardization of IEEE 1687 in 2014 research on this novel design started to bloom. So-called Reconfigurable Scan Networks (RSNs) help to make access to embedded components as found in e.g. System-on-Chip (SoC) designs more efficient by being able to dynamically change the path. One of the big research fields on them is testing and diagnosis. The Application Specific Integrated Circuit (ASIC) industry also shortly after incorporated RSNs into their designs.

RSNs have to be tested themselves and research works on coping with the new challenges they pose to testing. For these purposes fault simulation is necessary but due to their structure runtime of it is substantially high.

The goal of this thesis is to develop a fault emulation platform for RSNs and analyze its feasibility. In the end this new platform can be used for evaluation of test strategies such as Design for Testability (DfT) modifications or strategies in pattern generation. Also a method for simulating connected instruments is proposed. The speedup is analyzed and advantages and limitations are discussed.

Chapter 2 introduces necessary basics in hardware testing for this thesis. The general motivation behind testing is presented. Furthermore fault models, fault simulation and fault emulation are explained. Finally Linear Feedback Shift Registers (LFSRs) are introduced and an overview of Scan Design is given.

In Chapter 3 RSNs are presented. The structure of scan cells is explained and elements for reconfiguring a RSN are introduced.

Chapter 4 lists related work to RSNs and fault emulation.

Chapter 5 introduces the platform developed for this thesis. It first provides the general tool flow. Then the structure of the developed platform is described and finally the algorithm behind the platform is presented.

Chapter 6 lists and explains the results and finally Chapter 7 summarizes this thesis and lists possible future work and possible optimizations.

2 Basics of Hardware Testing

This chapter presents the necessary basics of testing. It assumes that the reader has basic knowledge about digital circuits and digital components such as the basic gates (AND, OR, NOT,...) and flip-flops. Otherwise missing knowledge can be looked up in [Mic03] and [WWW06]. Also some algebraic knowledge is of advantage as finite fields play a major role for LFSRs.

For an in-depth discussion about the topics in this chapter consult [BA04] or [WWW06] and [LN94].

This chapter starts with describing testing in general and explaining necessary terminology in Section 2.1. In Section 2.2 fault simulation is introduced. Some algorithms are presented and the runtime of these is discussed. Section 2.3 familiarizes the reader with fault emulation. The basic idea is explained. Then fault injection mechanics are introduced and finally the advantages and limitations of fault emulation to fault simulation is examined. LFSRs are introduced in Section 2.4 as they are used in the implementation and finally an overview of scan design is given in Section 2.5.

2.1 Testing

With the ever increasing complexity and size of circuits according to Moore's law [Moo06], which still carries a big role in the ASIC industry, and additionally with decreasing feature size, testing becomes a more important part in the industry. Because of the small feature size chips are more susceptible to process variations in the manufacturing process, which cause defects to emerge. These defect chips must be detected as early as possible to reduce costs. Wang et al. [WWW06] mention the rule of ten. It states that for each stage of manufacturing the cost of detecting a faulty Integrated Circuit (IC) rises tenfold.

An important measure for the general testing quality is the reject rate:

$$rej = \frac{\#faulty\ parts\ passing}{\#total\ parts\ passing}$$

The lower the reject rate the better the overall test quality and the lower the costs.

Another important measure for the general manufacturing quality is the yield:

$$yield = \frac{\#faulty\ parts}{\#total\ parts}$$

A higher yield indicates a better manufacturing quality. If the yield is very low, which often happens when a new manufacturing process is used, then diagnosis can help locating the cause of the faulty parts.

Examples for defects are broken interconnects, oxide breakdowns, shorts and many more. These defects cause a faulty output of a circuit called error. Because defects cannot be directly modeled they are abstracted to so-called faults. There are several fault models which will be superficially introduced but only the stuck-at fault model plays a role in this thesis.

Bridge Faults Bridge faults describe an undesired connection between two wires. There are several types of bridge faults: 4-way (byzantine), dominated, AND-dominated, OR-dominated, AND-wired and OR-wired.

Flip-flop Transparency Fault These faults affect flip-flops. Transparent flip-flops loose their ability to act as memory and degenerate to combinatorial buffers [UKW17].

Crosstalk Crosstalk occurs when the distance between two wires running in parallel is so small that induction occurs and the current of one wire is influenced.

For more details consult [BA04].

The stuck-at fault model states that a wire in the faulty circuit is either permanently driven by a constant 0 (stuck-at 0 fault) or 1 (stuck-at 1 fault). There are $2n$ possible stuck-at faults in a circuit with n wires because each wire can either have a stuck-at 0 or stuck-at 1 fault. But a technique called fault collapsing reduces this.

Equivalence fault collapsing uses the fact that on certain gates stuck-at faults at different ports cause the same output. For example a stuck-at 0 fault on any input port of an AND gate causes the same output as a stuck-at 0 fault on the output port. For OR gates this is true for stuck-at 1 faults. On wires both ends are equivalent for both stuck-at faults. It is important to note that no stuck-at fault is equivalent to each other on a fanout. Typically collapsing ratios of about 50%–70% can be achieved but this depends fully on the structure of the circuit. Generally less faults result in less testing time as Automatic Test Pattern Generation (ATPG) only has to target the collapsed faults and also fault simulation/emulation only needs to run for the number of faults in the collapsed list.

To test a circuit test patterns have to be generated in the first place. This is done by using so-called ATPG programs. For details consult [BA04] or [WWW06] as this topic is out of scope of this thesis. But the quality of the generated patterns has to be determined. This process is called fault grading and returns the so-called fault coverage. The fault coverage fc is calculated as follows:

$$fc = \frac{\#detected\ faults}{\#total\ faults}$$

The higher the fault coverage the better the patterns. Optimally the target is 100% or close to that. The automotive industry for example requires a fault coverage 0f 99.999% [ISO12]. But this cannot always be achieved. It depends completely on the circuit. If there are many redundancies for example the fault coverage automatically decreases. These redundant faults can be detected but that takes extra effort. Other faults are untestable simply because there is no way to sensitize a path from the fault's location to a Primary Output (PO). Also highly sequential circuits become a problem as gates and wires deep within become increasingly harder to control and observe.

A little more terminology first:

Primary Input (PI) and Primary Output (PO) are explained in Figure 2.1

Pseudo Primary Input (PPI) and Pseudo Primary Output (PPO) also see Figure 2.1

Circuit Under Test (CUT) This is the circuit which shall be tested for faults

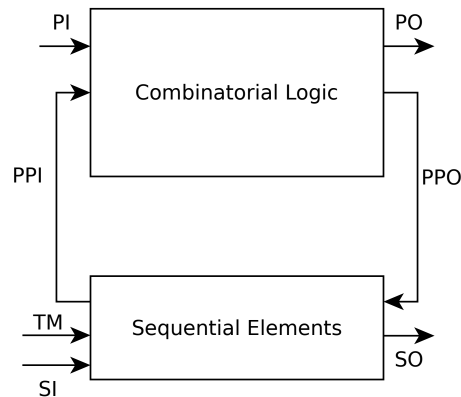


Figure 2.1: General model of a circuit (with optional ports SI, SO and TM)

2.2 Fault Simulation

Generally fault simulation is used for fault grading. Other applications are fault diagnosis and test compaction where redundant test vectors are identified and removed. In fault simulation a gate-level netlist is parsed and simulated with faults injected. In its core it uses a logic simulator extended with facilities to inject and detect faults.

2.2.1 Algorithms

The most basic algorithm is serial fault simulation where each fault is injected into the netlist one after another and simulated with the given test patterns. This is repeated until all faults have been simulated. Serial fault simulation can be sped up by using fault dropping. Fault dropping stops the simulation as soon as the output of the faulty circuit differs from the fault-free circuit and marks the fault as detected. This way the simulation does not need to apply all patterns to the CUT. Fault simulation is able to use four-valued logic. This allows to represent unknown values such as uninitialized flip-flops and to model tristates. Faults including unknown values are called potentially detected.

But serial fault simulation is very slow. Thus more efficient algorithms have been developed. One is the class of parallel algorithms. There is the Parallel Pattern Single Fault Propagation (PPSFP) algorithm where as many patterns as the processor's data width, are simulated in parallel and the Single Pattern Parallel Fault Propagation (SPPFP) algorithm where instead of patterns, faults are simulated in parallel. Note that PPSFP is only suitable for combinatorial or full-scan circuits. The meaning of full-scan is shortly explained in Section 2.5. There are also the deductive, concurrent and differential fault simulation algorithms. All algorithms mentioned here are presented in detail in [WWW06] and [BA04].

2.2.2 Runtime

Wang et al. [WWW06] indicate a runtime for serial fault simulation of $O(pn^2)$ where p denotes the number of patterns and n denotes the circuit size determined by the number of logic gates. They argue that the number of faults is proportional with the circuit size. Furthermore runtimes for parallel simulation and deductive simulation are presented which are $O(n^3)$ and $O(n^2)$ respectively. Differential fault simulation is reported to be the fastest [CY89] but still depends on the circuit size. Because of these runtime estimations fault simulation becomes less and less feasible for larger circuits. Also all algorithms, especially deductive and concurrent fault simulation, suffer from memory explosion.

2.3 Fault Emulation

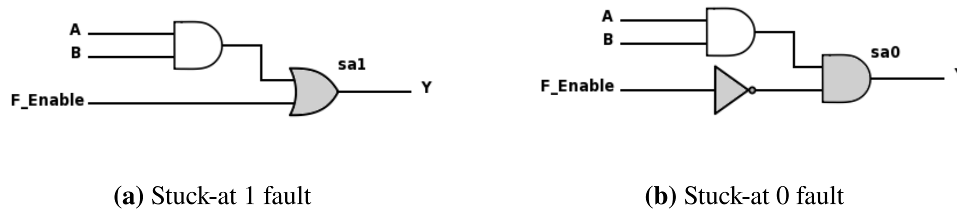
Fault emulation uses Field Programmable Gate Arrays (FPGAs) to emulate the CUT with faults injected. It is mainly used for hardening but also for fault grading [USS+17] which this thesis focuses on. Hardening describes the process of making the hardware tolerant to errors caused by external influences such as (cosmic) radiation. It is also possible to use fault emulation to select optimal Built-In Self-Test (BIST) structures [ERTU07]. In short BIST refers to the facilities that are added to design which allow it to test itself. This reduces test costs as no expensive ATE (Automatic Test Equipment) is necessary and also eases testing but on the other hand comes at the expense of additional hardware area. While circuit size plays a role in the runtime complexity of fault simulation as shown in Section 2.2.2, this is not the case for fault emulation. Instead it is limited by the size of the used FPGA.

An often used algorithm is serial fault emulation where similar to serial fault simulation each fault is injected one at a time and then all patterns are applied to the CUT. In general the runtime of fault emulation depends only on the number of patterns and the number of faults plus an eventual transmission overhead [SB98]. An alternative is to implement fault simulation algorithms in hardware [AM97].

2.3.1 Fault Injection

To inject faults, one can either modify the generated bitstream before loading it onto the FPGA or inject gates into a gate-level netlist. Optionally these gates can be equipped with an enable signal leading out of the CUT. Figure 2.2 shows how stuck-at faults with an enable signal are injected. Note that the gray gates are the injected ones. This uses the fact that applying the dominant value of each gate on one input makes the other input irrelevant to the output's value. The F_Enable lines become new PIs of the CUT. Hwang et al. [HHW98] also present a variant of injecting both stuck-at faults at once using only two NOR gates.

Injecting all faults at once or a subset at a time allows the test engineer to reduce the number of synthesis runs. Furthermore the enable signals allow to enable each fault dynamically in an arbitrary order using a decoder or a shift-register. This is also the approach used in this thesis.



(a) Stuck-at 1 fault

(b) Stuck-at 0 fault

Figure 2.2: Examples of injected stuck-at faults.

A problem with this approach though is the area blowup, as for each stuck-at fault one to two additional gates are injected into the design and there are two stuck-at faults per wire when no fault collapsing is used. This results in an area increase of 2.5–3.5 times [ERTU07] which may increase the number of synthesis runs depending on the size of the fault-free circuit and the FPGA. As will be shown in the results synthesis runs and programming the FPGA make up a big part of the time spent for fault emulation. Also the injected gates cause additional delays which reduces the maximum possible clock frequency.

Manipulating the bitstream before configuring the FPGA has the advantage that no extra space on the FPGA is needed and a synthesis run has to be executed only once. The synthesis run will actually be even faster than with injected gates, because the unmodified CUT is much smaller in comparison. But instead one needs to reverse engineer the bitstreams to find out how to manipulate these correctly. Also there is an overhead for generating gate-to-LUT mappings and manipulating the bitstream and reprogramming the FPGA multiple times [CHD95].

2.3.2 Advantages and Limitations

The biggest advantage of fault emulation to fault simulation is its runtime [ABH98]. As mentioned in Section 2.2 fault simulation generally runs in about $O(pn^2)$ while fault emulation needs $O(pf)$ where n denotes the number of logic gates in the circuit, p denotes the number of patterns and f denotes the number of faults.

But three-valued logic is harder to implement and also has a big impact on the size of the resulting circuit. It requires the duplication of every existing wire, except for maybe the clock, and exchange all basic gates with gates which can handle three-valued logic. This means that the size will be increased by a factor of 5–7 times as shown in [ERTU07]. That is twice the size two-valued logic needs. Thus the available area on the FPGA is significantly smaller.

When three-valued logic is not implemented fault emulation has to deal with unknown values. One solution is to change the design such that every register is resettable. The solution used in this thesis is to mark the occurrence of unknown values in the test patterns, so that they can be ignored. Also oscillation cannot be detected and thus can influence the final fault coverage as well.

2.4 Linear Feedback Shift Register

2.4.1 Theory

For a better understanding of LFSRs finite fields are covered first. We can express binary vectors $V = a_n a_{n-1} ... a_1 a_0$ as a polynomial $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ where each coefficient a_i either has the value 0 or 1. $P(x)$ is a member of the finite field $\mathbb{F}_2$.

A polynomial $p(x)$ is said to be *irreducible* in the finite field $\mathbb{F}_q$ iff $p(x)$ cannot be factored into non trivial polynomials. Furthermore $p(x)$ is said to be *primitive* iff it divides $1 + x^D$ with $D = q^n - 1$ but no polynomial $1 + x^i$ for any integer $i < D$. Every primitive polynomial is also irreducible. To get the reciprocal polynomial $r(x)$ of $p(x)$ let [LN94]:

$$r(x) = p(x)^{-1} = x^n p(x^{-1}) = a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n$$

Every reciprocal polynomial of a primitive polynomial is also primitive.

2.4.2 Structure and Properties

A LFSR is composed of a n stage shift-register where XOR gates are inserted at specific points. These specific points are called taps and correspond to “one terms” in a polynomial while missing XOR gates correspond to “zero terms”. For example Figure 2.3a shows a LFSR which represents the polynomial $p(x) = 1x^8 + 0x^7 + 1x^6 + 1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x + 1$. The highest term is always there and the +1 term corresponds to the feedback line from the last to the first flip-flop. The initial value of a LFSR is called a seed.

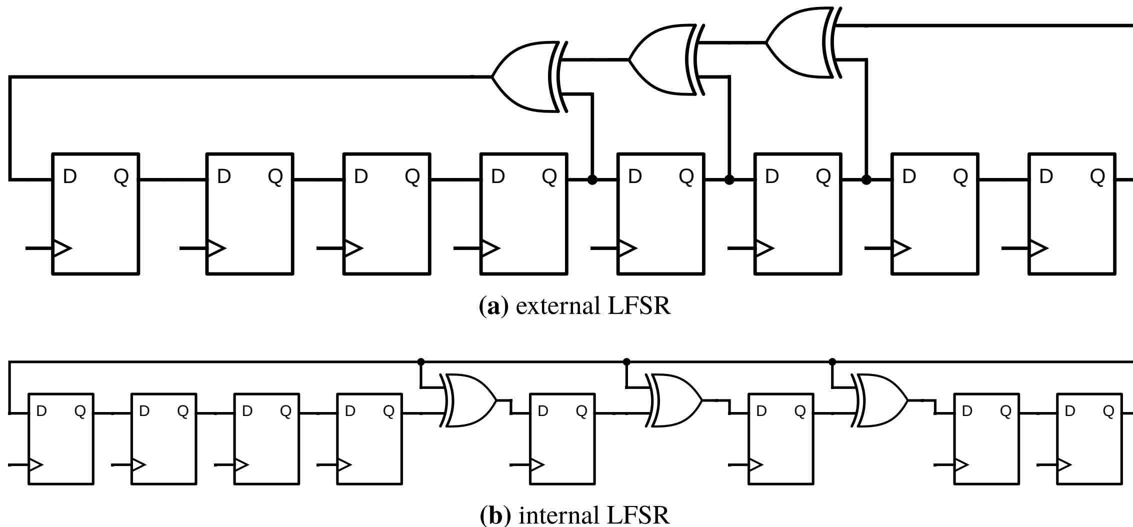
There are two types of LFSRs: internal and external.

External means that the XOR gates in the LFSR do not lie in the shift-register but instead all taps are XORed together and then fed into the first flip-flop. The flip-flops in external LFSRs are numbered in shift direction. In Figure 2.3a the direction is from left to right.

Internal LFSRs have the XOR gates in between the flip-flops. The flip-flops in internal LFSRs are numbered against the shift direction. An example for a internal LFSR is seen in Figure 2.3b. This LFSR represents the polynomial $q(x) = 1x^8 + x^4 + x^3 + x^2 + 1$ which also is the reciprocal of $p(x)$. The advantage of internal LFSRs is a better timing behavior because there is only one XOR gate between flip-flops and not all of them chained together on the longest path.

Both variants are equal except for the sequence of states they cycle through when started with the same seed.

The period of a LFSR is determined by the smallest integer t such that $p(x)$ divides $x^t = 1$. If $t = 2^n - 1$ then it is called a maximum-length-LFSR. This is given for primitive polynomials. Maximum-length-LFSRs generate all possible values modulo n except for 0, thus only $2^n - 1$ different states. The all 0 state has to be avoided because the LFSR is then stuck in it. There is an alternative implementation where instead of XOR gates XNOR gates are used. Then the all 1 state has to be avoided.



(a) external LFSR

(b) internal LFSR

Figure 2.3: Two types of LFSRs

2.4.3 Applications

LFSRs act as pseudo random number generators. That means the sequence they generate is seemingly random but actually deterministic depending on the seed. They are used in BIST for test pattern generation and signature analysis.

For test pattern generation they are structured as presented in Section 2.4.2 including an output port to feed one or multiple PIs of the CUT. LFSRs need slightly less area as binary counters. This makes them more ideal for exhaustive testing. Exhaustive testing is applying all possible combinations of the PIs which may cause a huge amount of data and is thus time consuming. There are other more directed methods. For a thorough discussion about test pattern generation see [WWW06].

In signature analysis, the output of the CUT is directed to a signature generator to compact the size of the output. This avoids saving every single output to a ROM for comparison. LFSRs are used as so-called Single-Input Signature Registers (SISRs). The output of the CUT is XORed with the content of the last flip-flop and shifted into the SISR. The resulting content of the shift register is the signature of the input stream. Basically the SISR becomes a CRC code generator. Compaction always comes with loss of information. The probability that two different input sequences produce the same signature, which is called aliasing, is:

$$P_{alias}(n) = \frac{2^{l-n} - 1}{2^l - 1}$$

l denotes the length of the input stream and n denotes the number of stages, i.e. the length of the shift register, of the SISR[WWW06]. If $l \ll n$ then the approximation $P_{alias}(n) \approx 2^{-n}$ can be used.

To compact multiple outputs from the CUT, a Multiple-Input Signature Register (MISR) can be used. A MISR has an extra XOR gate between every flip-flop of the shift register which is also connected to a PO of the CUT and depending on the polynomial to a tap.

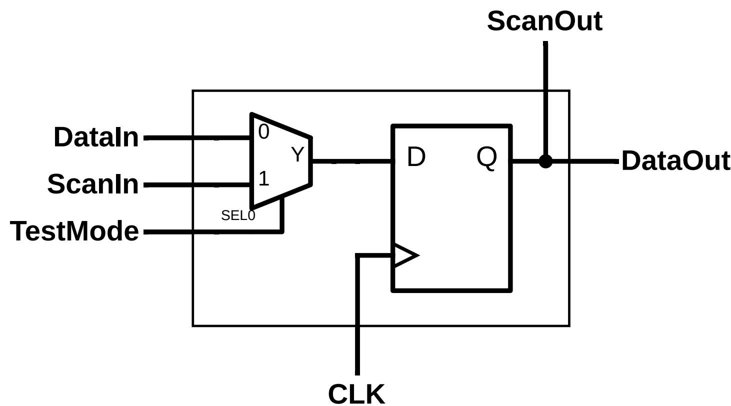


Figure 2.4: Level sensitive scan cell

2.5 Scan Design

Because ICs grow larger and become more complex, faults also become increasingly harder to test. To counter this trend several strategies and techniques have been invented. These are called DfT.

One important technique is Scan Design. The idea behind Scan is to replace all or a selected subset of all flip-flops with so-called scan cells. These scan cells are then connected in a way that they form a shift-register called a scan chain. This makes every flip-flop in the chain totally controllable and observable.

A scan cell consists of a flip-flop and a multiplexer. The multiplexer switches between the normal input D and the scan input ScanIn depending on the TestMode pin. Note that the names may differ from design to design and are chosen here for easier reference. Moreover, scan cells can be extended with a separate clock input and another reset if necessary. Figure 2.4 shows an example of a scan cell without the optional ports.

The ScanIn port of the first scan cell in the chain is connected to a new PI and the test mode pin also becomes a PI. The output port of the last scan cell is then connected to a new PO called commonly ScanOut. Figure 2.1 shows the ScanIn port (SI), the ScanOut port (SO) and the TestMode port (TM). As once can see the state of each flip-flop in the scan chain can be set independently of the combinatorial logic or other PIs.

The advantage of a scan design is, since every flip-flop is totally controllable and observable, only a combinatorial ATPG is required. The PPIs and PPOs can then be seen as real PIs and POs. Also this reduces testing time as the number of test patterns for combinatorial designs is smaller than that of sequential designs. The only difference is that the flip-flops' values have to be shifted in first. This needs exactly as many clock cycles as the length of the scan chain.

When all flip-flops are replaced by scan cells, this architecture is called a full scan design. The problem with scan cells is the additional multiplexer. This causes an area increase and also additional delay. This may also lead to the design not meeting its timing constraints.

Flip-flops on the critical path can be left unchanged. This is called partial scan design. It commonly comes with a loss of fault coverage and increase in testing time compared to full scan. Additional effort is needed to compensate the loss of controllabilty and observability and moreover, the use of sequential ATPG becomes necessary.

The increase in circuit size also affects the number of flip-flops in a design. At some point the length of a full scan chain becomes too large that the advantage of reduced testing time is lost. One solution is that multiple scan chains can be implemented in parallel, but they require additional ports as well. To avoid these ports one can use the Illinois Scan Design [HP99].

3 Reconfigurable Scan Networks

The increasing shifting time for full scan designs led to the development of a general and scalable approach called RSN. RSNs allow for arbitrary access to certain components in a design. This is especially helpful for SoC and similar architectures as on-chip components, from now on called instruments, such as sensors or DSPs can be accessed without influencing other instruments. RSNs are not only useful for testing but also for diagnosis and debugging. They were standardized by the IEEE in 2014 [IEE14].

In Section 3.1 the structure of RSNs and a scan cell in it are presented. Section 3.3 explains how a RSN is working and finally Section 3.4 introduces elements which allow a RSN to be reconfigured.

3.1 Structure

In RSNs flip-flops are still replaced with scan cells like in a scan design but instead of creating one or multiple linear chains the connection between slices of a scan chain, called scan segments, can be arbitrarily created as long as no loop is formed. RSNs can be described using a Directed, Acyclic graph (DAG) where the segments are the nodes and the connections are the edges [Sch13]. Figure 3.1 shows an example of a simple RSN. “TDR” denotes a scan segment while “SIB” denotes a Segment Insertion Bit (SIB) which is introduced later in Section 3.4.

In general a RSN consists of scan cells, combinatorial logic, multiplexers and optionally latches. There is a global clock input and three global inputs for each operation *shift*, *update* and *enable*. Additionally there may be data in- and outputs for communication with instruments.

3.2 Scan Segment

Figure 3.2 shows a simplified RSN scan segment. Note that the clock signal has been omitted. A scan segment consists of a shift register and optionally a shadow register. The shadow register is generally triggered by a negative edge of the clock. The `DataIn` and `DataOut` ports are connected to an instrument. The shift register acts additionally as a read-only access to a connected instrument while the shadow register instead provides write-only access to it.

The `select` signal indicates if a segment is in the active scan path. If it is not active the contents in the scan segment shall not change. It can optionally be omitted and for example be replaced by local clock gating.

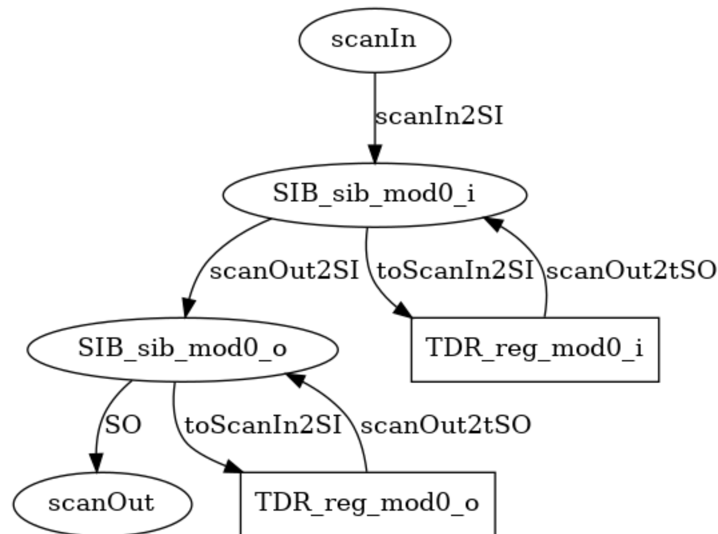


Figure 3.1: Example of a RSN represented as a DAG

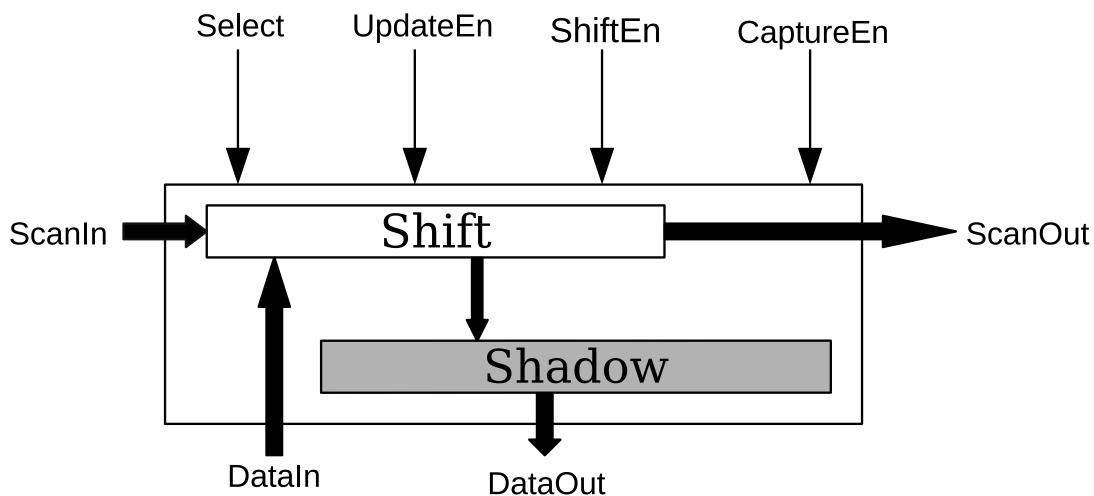


Figure 3.2: A simplified read-write scan segment without disable signals

The `select` signal if present is controlled by arbitrary combinatorial circuitry with inputs from any shadow register in any scan segment while the operation signals are connected directly to an IEEE1149.1 Test Access Port (TAP) [IEE13b]. The same applies to the update disable signal and the capture disable signal. These signals are collectively referred as *internal control signals* as they cannot be accessed directly through the TAP. IEEE 1687-2014 [IEE14] defines the interface between the TAP controller and the RSN. This allows for a highly flexible structure and a possibly high complexity. Also the TAP interface offers a standardized and already widely available access.

IEEE 1687 also introduces Instrument Connectivity Language (ICL). It is used to describe how each component inside the RSN is connected.

3.3 Operation

A scan segment supports up to three operation modes:

shift When `ShiftEn` is active data is shifted through the shift register from `ScanIn` to `ScanOut` just as in a classical scan design. This command is available on all types of segments. Also the shadow register is kept stable during this mode.

capture The capture command loads the shift register in parallel with the data laying on the `DataIn` signal. This mode is not available for write-only segments.

update The update command loads the shadow register with the content of the shift register in parallel. This mode is not available for read-only segments.

A scan segment implementation can optionally introduce additional signals to disable capture and/or update commands at any time independent of the `Select` signal. Only one mode can be active at a time. A RSN operates synchronous to the global clock. It consists of an atomic sequence of prior introduced modes: *capture*, *shift*, *update*. This is sequence is called *CSU operation*. The capture and update command must complete within a clock cycle. The update command is synchronous to the falling edge of the global clock while capture and shift are synchronized to the rising edge. The shift command can complete in an arbitrary number of cycles which usually corresponds to the length of the active scan path.

3.4 Reconfiguration

The content of each sequential element in the RSN including the primary data and control signals are called the *configuration*. A active scan path consists of multiple scan segments connected from the TAP's `scanIn` to the TAP's `scanOut` port. Each segment on the path must be selected while each segment off-path must be deselected. If these conditions are fulfilled a configuration is called valid. A sequence of CSU operations which allow to access a scan segment is called *scan access*. The process of generating the scan access is called *pattern retargeting*. To allow arbitrary hierarchies IEEE 1687-2014 proposes Segment Insertion Bits.

Segment Insertion Bit SIBs allow to dynamically add and remove segments from the active scan path. They are able to build up an hierarchy inside the RSN.

A SIB consists of a 1 bit shift register and a 1 bit shadow register. It is a write-only scan segment thus a SIB only knows the shift and update modes.

Another segment, SIB or module is connected to a SIB through the `ToScanIn`, `ToSelect` and `FromScanOut` ports. The shadow register controls a multiplexer which sets the source of the shift register to either `FromScanIn`, which either comes from a TAP, another segment, SIB or from `FromScanOut`. That means if the shadow register is deasserted the subset of all segments, which the SIB controls, is excluded from the current active path otherwise it is included. It is recommended by the IEEE to also add a delay bit to avoid race conditions. Figure 3.4 shows an example RSN.

Initially all SIBs are set to open, i.e. bypass the connected segment.

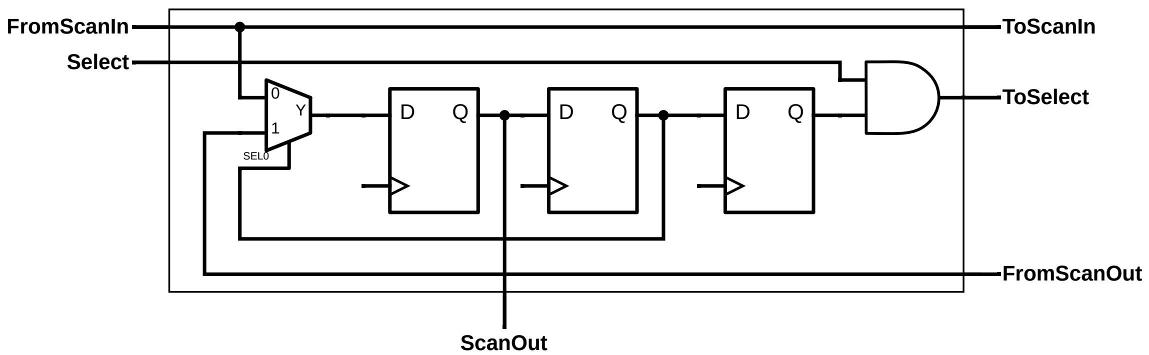


Figure 3.3: A SIB with delay bit

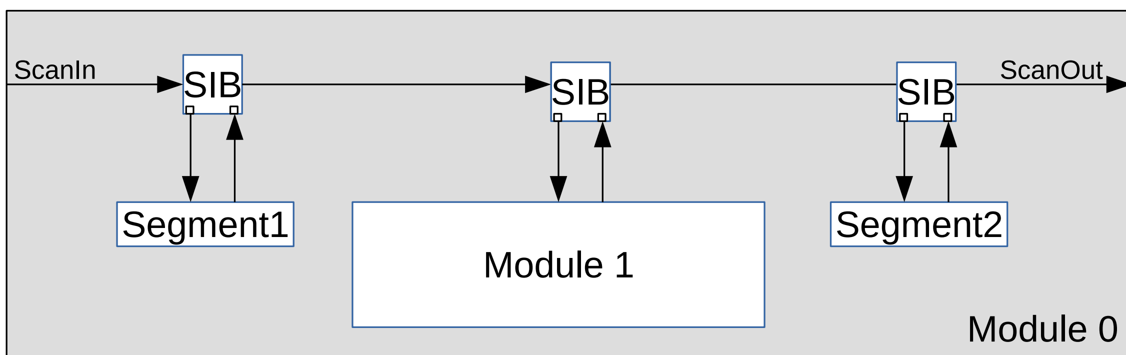


Figure 3.4: Example of a RSN using SIBs

In Figure 3.1 the SIBs can dynamically insert either segment `mod0_i` and/or `mod0_o`.

MUX An alternative method is shown in [Bar14] and uses multiplexers and configuration bits.

The idea is to separate configuration and data access. This is achieved by inserting a segment called *access control segment* which controls whether data is shifted through the configuration bits or through the data segments. Each configuration bit controls another multiplexer which in- or excludes the corresponding scan segment or module. Each module has an access control segment. The advantage is a faster data access as the configuration bits do not lie on the active data scan path. Initially all configuration bits including the access control segment are set to 0 and the content of all data scan segments is unknown. The MUX-based design can be changed so that instead of in- or excluding segments two segments are mutually exclusively accessible.

In general the scan path control logic can consist of arbitrary combinatorial logic as long as valid scan paths are created. For example in [Bar14] and [Sch13] RSNs were generated where the control logic consists of ISCAS'85 benchmark circuits.

RSNs provide unique challenges to testing and especially test pattern generation [CMR+15]. Scan chains themselves are rather easy to test by using flush tests, i.e. shift a pattern of zeroes and ones through the chain and compare the output with the input. But additionally the operation has to be tested and the shadow register. Especially the shadow register proves itself as difficult to test as its

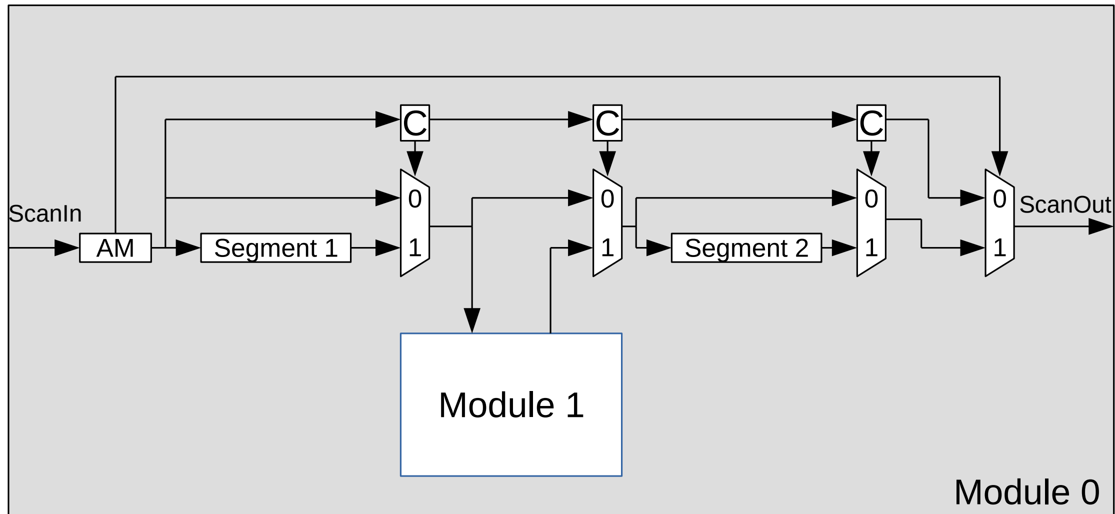


Figure 3.5: Example of a RSN using multiplexer

only controllable and observable through the shift register [UKW17]. Also scan access has to be tested if faults cause valid configurations to become invalid. Furthermore SIBs, multiplexers and the combinatorial logic for segment selection needs to be tested, too.

4 Related Work

4.1 RSNs

This thesis is based on the works of Schaal [Sch13] and Baranowski [Bar14]. [Sch13] analyses test strategies and introduces new strategies tailored to RSNs.

The doctoral thesis from Baranowski [Bar14] discussed multiple topics about RSNs on which today extensive research is still performed on. The CSU-based temporal abstraction is introduced [BKW15b]. Furthermore bounded model checking based on that abstraction is presented [BKW12]. Access time minimization and retargeting [KZL15], [IK16] is the next topic and finally a method for access port protection [BKW15a], [LA15] is introduced.

Zadegan et al. [ZLJ+14] gave a general overview of research done on RSNs to that date.

Besides verification, testability of RSNs is an important topic. Cantoro et al. [CMR+15] investigated and laid out the challenges of testing RSNs and proposed a test strategy to tackle these. Ull et al. [UKW17] proposed a minor modification of RSNs to increase the observability of the shadow register by adding a path from the shadow register to the shift register bypassing any connected instrument.

Tsertov et al. [TJD+16] presented a collection of benchmark RSNs to allow objective comparison. These were found too late to be considered in this thesis. But one of the sets is derived from the ITC'02 benchmark circuits which are used in this thesis.

4.2 Fault Emulation

Most of the research on fault emulation today focuses on hardening and fault tolerance evaluation [USS+17].

But there is still research done on fault emulation for fault grading.

Ullah et al. [USS+17] presented an approach using Xilinx primitives and the MicroBlaze™ to rewrite Look Up Tables (LUTs) to inject faults on runtime without either regenerating a bitstream or manipulating and loading it repeatedly onto the FPGA. This is called Runtime Reconfiguration. A LUT-mapper is used to preserve the design's structure while converting the netlist to LUT-level netlist usable by synthesis tools for place-and-routing.

Lavanyashree and Jamuna [LJ17] showed the feasibility of using multiple copies of the circuit to process multiple faults in parallel. They use a gate-level injection approach. A demultiplexer is used to enable either a stuck-at 0, stuck-at 1 or a bit flip fault.

Ellervee et al. [ERTU07] implemented a fault emulation platform which resembles the platform developed in this thesis. Instead of transmitting externally generated test pattern a LFSR is used. A shift register is used to enable each fault and fault dropping with a GOLD CUT was implemented.

5 Implementation

This chapter presents the implementation called FEPfR (Fault Emulation Platform for RSNs) developed for this thesis.

FEPfR runs on a Xilinx Virtex®-7 FPGA VC707 evaluation board equipped with a XC7VX485T FPGA. The XC7VX485T has 485,760 Logic Cells which corresponds to 303,600 LUTs and 607,200 flip-flops [Xil18]. A UART interface is used for communication between FEPfR and the workstation. The UART is an IP from opencores.org [Cab16]. It is using a baud rate of 500000 bps. An eight bit word is transmitted with no parity and one stop bit. No flow control is used.

The design of FEPfR aims to be taking as little area as possible while not limiting the maximum achievable clock frequency. It only needs 247 LUTs (0.08%) and 259 registers (0.04%) when synthesized with a dummy circuit which ANDs all the inputs. The smaller the area needed for FEPfR itself the bigger the CUTs can be. Additionally FEPfR is portable i.e. does not depend on certain Xilinx primitives or similar. It is written in SystemVerilog 2012 [IEE13a].

Section 5.1 introduces the tools in this work and the general tool flow. Section 5.2 presents the structure and each element of FEPfR. Finally Section 5.3 explains the algorithm used in FEPfR.

5.1 Tools

This subsection presents all used tools and details on in-house developed ones. Vivado 2016.4 is used for the final synthesis and bitstream generation while the RSNs were synthesized before fault injection using Synopsys Design Compiler® M-2016.12 with a small subset of the lsi10K library. The subset contains only 2-input basic gates and a D-flip-flop without a set or clear input. For synthesis with vivado a behavioral description of the lsi10K subset was written in Verilog. This behavioral description is also used by TLib. The RSNs were generated with gICL, a tool written by Schaal [Sch13]. For pattern generation the eda1687 tool written in [Bar14] was used. Figure 5.1 shows how all tools were working together. The remaining part of this subsection presents the in-house tools used.

gICL

gICL is a Ruby script which generates ICL files, as described in Section 3.1, and Verilog files. The input is a SOC file downloaded from the official ITC'02 Test Benchmarks page[1]. The format of a SOC file can be looked up in [Eri02].

[1] http://www.itc02socbenchm.pratt.duke.edu/

Slight changes were necessary to replace shadow registers with LFSRs. The reason is given in Section 5.2. A new class called `ICL_LFSR` was written which represents a scan segment with a LFSR as shadow register and another class `ICL_LFSR_Module` was also added. `ICL_LFSR_Module` embeds `ICL_LFSR` and provides an interface, i.e. connections, generics, to the user. The polynomial for the LFSR is chosen according to the length of the scan segment and also a random nonzero initial seed is generated.

eda1687

eda1687 [Bar14] is a tool tailored specifically to generate patterns for RSNs. It internally uses a SAT-solver. The patterns are stored in the Standard Interface Test Language (STIL) format. eda1687 reads ICL files and generates SAT instances which are then solved to generate patterns. It can generate several types of pattern sets. The pattern sets used in this thesis are the RST+M sets. RST means a test for the reset functionality is applied and M indicates a merged test where multiple segments are tested simultaneously. For more details on the sets see [UKW17].

TLib

TLib is a collection of tools for logic simulation (lsim), fault simulation (fsim), pattern generation (plist), fault list generation (flist) and circuit manipulation. Circuit manipulation includes small corrections such as removal of unused logic, renaming, removal of tristates (correct) but also gate-level fault injection (finject). The patterns generated by TLib are stored in a subset of the Waveform Generation Language (WGL) format which is easy to parse. TLib can also read in and write a subset of STIL. The netlists must be given as Verilog files. Furthermore TLib can generate fault lists in either an internal format or in a format parseable by TetraMAX.

Small validation tools were developed prior to this thesis to run regression tests on the logic and fault simulation. These tools parse the patterns, generate a Verilog testbench, use the generated Value Change Dump (VCD) file and the WGL file and converts both into a simpler format called "dump". For this thesis the logic simulation tool was used to convert the STIL file to WGL and determine the positions of possible unknown values. Small changes have been made to the WGL/VCD-conversion (conv2dump) tool used in validation such that the output dump file can be used by FEPfR. conv2dump also sets the corresponding frame bits as described in Section 5.3.

5.2 Structure

Figure 5.2 shows the structure of FEPfR. The Pattern Frame Recognizer (PFR) is a state machine in the top level of FEPfR which filters the incoming data. Its operation is illustrated in Section 5.3. The received patterns are stored in a FIFO. The FIFOs were modified such that their read pointer can be reset separately. The Test Pattern Generator (TPG) reads the patterns from the FIFO and applies them to the CUT until the FIFO is empty. All the separate `read_rst` signals in Figure 5.2 are the same but have different meaning for each connected component. They are separated to avoid overlapping edges. Furthermore the TPG controls the clock of the CUT. The CUT's clock is running at the same frequency as FEPfR's clock.

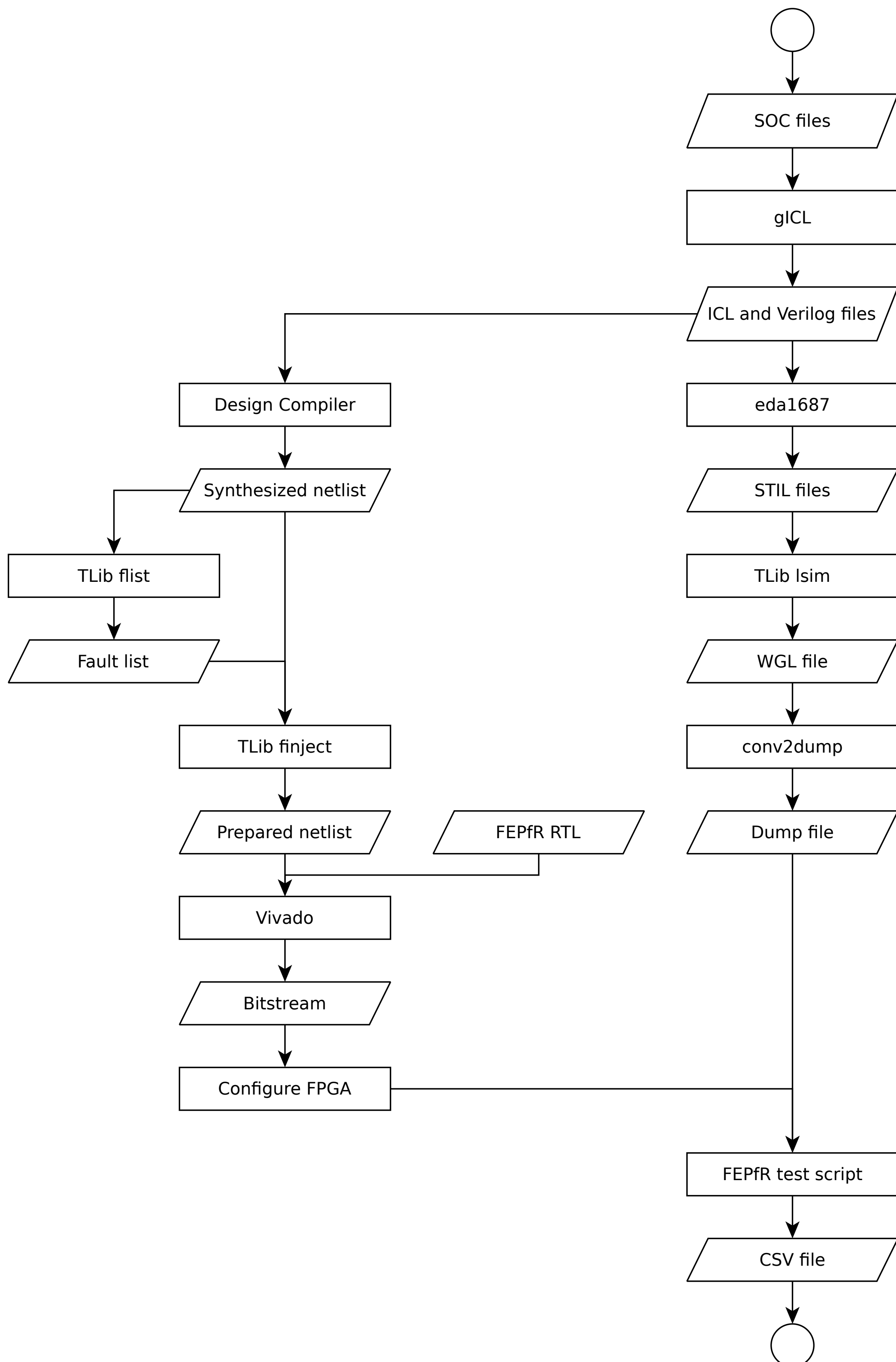


Figure 5.1: Tool Flow used in this thesis

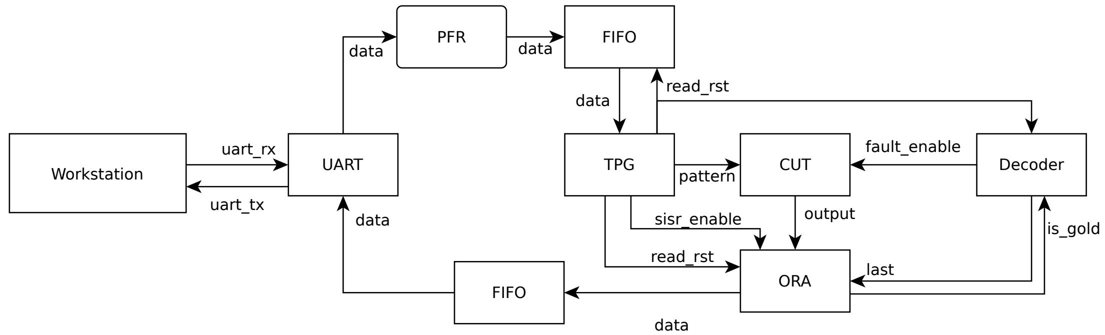


Figure 5.2: Structure of FEPfR

The output of the CUT feeds the 32-stage SISR inside the Output Response Analyzer (ORA) directly. 32 stages were chosen to minimize the probability of aliasing as shown in Section 2.4.3. The SISR implements the polynomial $p(x) = x^{32} + x^7 + x^6 + x^2 + 1$ which is a primitive polynomial. The decoder is a binary-to-one-hot decoder with an internal counter. This counter increments when `read_rst` is asserted and `is_gold` is deasserted. `is_gold` is necessary to avoid that the decoder enables the first fault while the gold signature is generated. The `last` signal is asserted by the decoder as soon as the biggest value is reached.

The depth of the FIFO connected to the TPG is fixed and controlled by a generic. The generic is chosen independent of the number of patterns to allow repeated test with arbitrary numbers of patterns as long as currently set depth is not exceeded. The depth of the FIFO connected to the ORA depends on the number of faults:

$$depth(\text{FIFO_ORA}) = pow\left(2, \left\lceil \frac{\text{Number of faults}}{8} \right\rceil\right)$$

where $pow(b, e) = b^e$. In this thesis the number of faults always surpasses the number of patterns.

There also are further changes to the CUT than just injecting faults. To minimize occurrences of unknown values, the shadow registers were replaced by LFSRs. These LFSRs aren't necessarily maximum-length-LFSRs but each single one has a random seed. Also the shadow register's DataOut signal is connected to the shift register's DataIn line. They simulate instruments connected to the segment. This method was chosen because the required area overhead is the smallest except for constant values, but a real instrument would rarely just output a constant value. An alternative are cellular automata. Real instruments are larger than two or four XOR gates. This proves to be a problem as the area overhead they incur will add up over the number of segments in the CUT. Moreover, real instruments have external inputs which have to be driven. This would either be achieved by extending the pattern with additional values for these instruments or by generating random values. To generate random values LFSRs or cellular automata can be used. But this approach boils down to using pseudo-random number generators to set the shadow register's content which is already achieved by replacing them with LFSRs except there is also the additional area overhead for the instruments themselves. Therefor, replacing shadow registers with LFSRs or cellular automata proves to be the best approach.

Furthermore the method from [UKW17] is additionally used to increase fault coverage. This additionally reduces the number of unknown values combined with LFSRs.

Bit	7	6	5	4	3	2	1	0
Meaning	reset	select	captureEn	shiftEn	updateEn	scanIn	sisr_enable	frame

Table 5.1: Structure of pattern input

The structure of the patterns can be seen in Table 5.1. Each field represents one bit. Note that `sisr_enable` is not connected to the CUT but instead directly to the SISR in the ORA. `sisr_enable` is deasserted when simulation found an unknown value otherwise asserted. This avoids unknown values tampering with the signature. Also the frame bit is ignored after the PFR.

The ORA does not use an internal counter to count the detected faults. But this can be easily changed to reduce the transmission size to the workstation.

To avoid the transmission of patterns they can be synthesized as a ROM. This would significantly speed up a run but on the other hand the flexibility to simply transmit a new pattern set for fault grading would be greatly limited as each change in patterns would lead to a new synthesis run. Therefor, the current structure makes FEPfR suitable for refining test patterns, guiding ATPG and works independently of the used fault model.

Also fault dropping as for example in [ERTU07] has not been implemented because of the overhead area required for a fault-free CUT.

Algorithm 5.1 PFR

```
 1: procedure PFR(inputs)
 2:     for all i ∈ inputs do
 3:         if i[0] = 1 then
 4:             FIFO_TPG.PUSH(i)
 5:             INPUTS.REMOVE(i)
 6:             break
 7:         end if
 8:     end for
 9:     for all i ∈ inputs do
10:         FIFO_TPG.PUSH(i)
11:         if i[0] = 1 then
12:             FIFO_TPG.PUSH(i)
13:             ASSERT(startTPG)
14:             break
15:         end if
16:     end for
17: end procedure
```

Algorithm 5.2 TPG

```
1: procedure TPG
2:     Await(startTPG)
3:     Enable(CUTClock)
4:     loop
5:         while FIFO_TPG is not empty do
6:             pattern ← FIFO_TPG.pop
7:             ApplyToCUT(pattern)
8:         end while
9:         Assert(ReadRst)
10:    end loop
11: end procedure
```

5.3 Algorithm

The incoming data is checked if the frame bit is set to one as described in Algorithm 5.1. `inputs` in mentioned algorithm represents all data transmitted through UART. Each pattern only occupies seven bits so the frame bit is used to mark the first and the last pattern. When the first pattern has been detected by a set frame bit, every subsequent byte is treated as a test pattern and pushed onto the FIFO until a pattern has the frame bit set again. The detection of the last pattern also asserts the `start` signal. This method removes any transmission overhead and is very easy to realize. Also it allows to send an arbitrary number of patterns as long as the FIFO connected to the TPG does not become full without the last pattern having a set frame bit.

The TPG is not generating patterns itself but instead applies them to the CUT, as can be seen in Algorithm 5.2, and is the main control unit of FEPfR. As soon as `start` is asserted, the TPG starts to apply patterns to the CUT and the SISR inside ORA starts at the same time to generate the golden signature (fault-free CUT). When the TPG starts, it also enables the clock connected to the CUT. This is done to avoid having the RSN in a different state when generating the golden signature.

After the first round the decoder receives a signal to increment and thus the first fault is enabled. Then the procedure repeats for each fault until the decoder asserts the `last` signal.

When the TPG asserts `ReadRst` the ORA compares the signature currently generated by the SISR with the gold signature. A "1" is shifted into the internal data field if they differ and a "0" otherwise. For every eight comparisons, the ORA pushes the results into the FIFO. If `last` has been asserted and the ORA has results left which are not pushed onto the FIFO yet, they will then be pushed onto it. This is done directly after assertion (see Algorithm 5.3). The UART immediately sends the data back to the workstation. The workstation measures the time needed from sending the patterns until all the expected data is received and also analyses the data for the number of detected faults and prints the fault coverage and the runtime.

The UART has a signal for frame errors, i.e. the stop bit was off. If that signal is asserted while patterns are transmitted FEPfR immediately stops and starts sending "0xFF" to the workstation. Also a repeating "10" pattern is sent when the FIFO connected to the TPG is full but the last pattern hasn't been detected.

Algorithm 5.3 ORA

```
1:  procedure ORA
2:      data ← 0
3:      counter ← 0
4:      Await(ReadRst)
5:      goldSignature ← signature
6:      repeat
7:          Await(ReadRst)
8:          result ← goldSignature ≠ signature
9:          data ← data << 1 | result                // << is left shift, | is bitwise or
10:         counter ← counter + 1
11:         if counter = 8 then
12:             FIFO_ORA.push(data)
13:             counter ← 0
14:         end if
15:     until last asserted
16:     if counter ≠ 0 then
17:         FIFO_ORA.push(data)
18:     end if
19: end procedure
```

6 Results

For collecting results two classes of test RSNs were used: SIB-based (_sib suffix) and MUX-based (_mux suffix). Table 6.1 presents an overview of the RSNs used in this thesis. The column "Gates" indicates the number of gates the RSN consists of after synthesis with Design Compiler. "Gates (+Inj)" presents the number of gates after fault injection. The number of gates after fault injection is about 4.4 times bigger than before. The column "Levels" shows the depth of the hierarchy in each RSN. "Faults" lists the number of stuck-at faults.

Vivado synthesis ran on a Server equipped with two Intel® Xeon® CPU E5-2667 CPUs and 500GB RAM running with CentOS 7. At most 3 synthesis runs were active in parallel. The fault simulation also ran on the same server.

For fault simulation Synopsys TetraMAX® M-2016.12 was used. The same patterns were used for fault simulation and fault emulation. Table 6.2a presents the result of the fault simulation and Table 6.2b the results of the fault emulation. The column "Synthesis" indicates the runtime of the synthesis run.

The target clock frequency for synthesis is 50 MHz. MUX-based RSNs seem to have very long combinatorial paths with injected faults. Thus synthesis runtime increases because Vivado tries to meet the timing. Moreover, additional optimizations were run. The clock frequency for h953_mux, g1023_mux and d281_mux had to be reduced to 32 MHz to meet timing.

The area of the FPGA didn't become the limiting factor as the biggest synthesized RSN (d695_sib) only takes up $\approx$ 25% of the FPGA's area. d695_mux is missing because Vivado kept using up all memory when synthesizing and finally crashed. That is also the reason bigger RSNs from the ITC'02 set are not listed.

Name	Gates	Gates (+Inj)	Levels	SIBs/Muxes	Scan Segments	Patterns	Faults
u226_mux	15770	69899	2	59	99	8095	38100
u226_sib	16109	71387	2	50	90	7685	38502
d281_mux	40041	176849	2	67	117	19760	96408
x1331_mux	41275	182052	3	38	63	49349	99074
d281_sib	42399	188490	2	59	109	19739	100942
x1331_sib	42753	189511	3	32	56	20583	101934
g1023_sib	57824	256258	2	80	145	27425	137922
g1023_mux	59686	265781	2	94	159	28655	142028
h953_sib	60084	266296	2	55	101	28576	143284
h953_mux	62375	277788	2	63	109	28968	148048
d695_sib	91109	403811	2	168	325	42808	217266

Table 6.1: RSNs used in this thesis

Name	Fault Coverage	Time
d281_mux	98.30%	06:13:20
d281_sib	98.52%	06:47:40
g1023_mux	98.10%	41:02:12
g1023_sib	98.32%	39:37:49
h953_mux	96.67%	45:41:07
h953_sib	96.77%	44:24:06
u226_mux	97.49%	00:18:50
u226_sib	97.06%	00:25:49
x1331_mux	99.16%	07:34:14
x1331_sib	96.64%	06:29:41
Average	97.70%	19:51:29

(a) Fault Simulation

Name	Fault Coverage	Time[s]	Synthesis
d281_mux	98.03%	60.011	01:07:13
d281_sib	95.49%	40.361	01:07:34
g1023_mux	99.35%	127.901	03:37:27
g1023_sib	96.60%	76.401	02:36:48
h953_mux	99.50%	134.769	11:43:52
h953_sib	96.73%	82.610	04:46:23
u226_mux	99.23%	6.368	00:41:13
u226_sib	96.24%	6.108	00:29:58
x1331_mux	99.18%	98.983	01:29:15
x1331_sib	95.56%	42.487	00:30:35
Average	97.59%	67.600	02:26:34

(b) Fault Emulation

Table 6.2: Runtime and fault coverage of fault simulation and emulation

d695_sib is missing in the results because a bug prevents FEPfR to transmit all of the results. Only a small subset was received after one hour. The cause of this bug wasn't found.

The runtime of fault emulation can be estimated using the following formula:

$$T(p, f, cf) = p \cdot f \cdot \frac{1}{cf} + O_{trans}(f, p)$$

where p is the number of patterns, f is the number of faults, cf indicates the clock frequency and $O_{trans}(f, p)$ is the overhead caused by transmitting the patterns and receiving the result. In this thesis $O_{trans}(p)$ equals to:

$$o_{trans}(f, p) = \frac{10\left(p + \left\lceil \frac{f}{8} \right\rceil\right)}{500000}\,\mathrm{s}$$

where the factor 10 comes from the number of bits that are actually transmitted through UART (1 start bit, 8 data bits and 1 stop bit), $\left\lceil \frac{f}{8} \right\rceil$ is the size of the result in bytes and 500000 is the baudrate. For example the estimated runtime for u226_mux is:

$$T_{u226_mux}(8095, 38100, 50\,\mathrm{MHz}) = 6.168\,39\,\mathrm{s} + \underbrace{0.257\,15\,\mathrm{s}}_{O_{trans}(8095,38100)} = 6.425\,54\,\mathrm{s}$$

This value is 0.90% off of the experimentally determined value. The reason that the experimental value is smaller is that this model assumes the results are sent after emulation is done but the implementation starts sending every 8 available results as explained in Section 5.3.

In general the transmission overhead in average is around 1.77%. Compared to the total time this overhead is small enough that no faster interface is justified but may be desirable with increasing pattern and fault numbers.

There is a noticeable difference in fault coverage. Normally fault coverage of 2-valued emulation is expected to be lower than 3-valued simulation. Also not every output value of the CUT is considered in the signature. But when taking a look at all MUX-based circuits except d281_mux they are significantly higher than in the simulation. There hasn't been found any explanation for this phenomenon.

Name	FE (+Synth)[s]	FS[s]	Speedup (+Synth)	FE[s]	Speedup
d281_mux	4093.01	22,400	5.47	60.01	373.27
d281_sib	4094.36	24,460	5.97	40.36	606.03
g1023_mux	13 174.90	147,732	11.21	127.90	1155.05
g1023_sib	9484.40	142,669	15.04	76.40	1867.37
h953_mux	42 366.77	164,467	3.88	134.77	1220.37
h953_sib	17 265.61	159,846	9.26	82.61	1934.95
u226_mux	2479.37	1,130	0.46	6.37	177.44
u226_sib	1804.11	1,549	0.86	6.11	253.59
x1331_mux	5453.98	27,254	5.00	98.98	275.34
x1331_sib	1877.49	23,381	12.45	42.49	550.31
Average	10 209.40	71,488.8	6.96	67.60	841.37

Table 6.3: Speedup of fault emulation to fault simulation

Table 6.3 lists the runtimes of fault emulation once in- and once excluding synthesis and the runtimes of fault simulation in seconds. “FE” stands here for Fault Emulation, “FS” for Fault Simulation and “Synth” for synthesis. The column “Speedup (+Synth)” is the ratio of fault simulation to fault emulation and synthesis time. This scenario describes repeated emulation where the design is changed and thus has to be synthesized anew each time. In general the speedup factor including synthesis is between around 0.46 and 15. The average speedup factor is around 6.96. For the two smallest RSNs fault simulation proves to be faster. But the bigger the RSN gets the higher the speedup becomes. To improve the speedup the synthesis run can be executed with runtime optimized primitives (called strategies in Vivado).

The column “Speedup” is the ratio of fault simulation to only fault emulation. This describes the scenario where subsequent runs on the same synthesized design are executed. The speedup is in the range from low hundreds to almost 2000. In average the speedup factor is around 841.37. The same effect as in the synthesis scenario can be made that with increasing size the speedup becomes better but the increase is significantly higher. These speedup factors are also around 4 times higher than those determined by Ellervee et al. [ERTU07].

7 Conclusion

7.1 Summary

The fault emulation platform developed in this thesis called FEPfR has been proven to be significantly faster than fault simulation using commercial grade tools. The speedup factors achieved are between 0.4 and 15 when also considering synthesis and 177–1935 when only fault emulation itself is considered. This is a considerable factor even compared to factors achieved so far.

As such fault emulation has been proven to be clearly more efficient and so can reduce research time significantly. Furthermore, the industry benefits from using fault emulation when grading patterns for RSN designs with a shorter time-to-market.

7.2 Future Work

Future work includes researching the cause for the unreasonable memory consumption of the synthesis. Also 3-valued logic should be implemented. The differences in fault coverage mainly stem from the fact that simulation uses 3-valued logic and fault emulation only 2-valued logic. But 3-valued logic needs additional area which combined with fault dropping may cause limitations. Also synthesis time will increase. FEPfR should then be evaluated with different fault models such as FFTR and bridge faults.

One idea to implement fault dropping in FEPfR is to replace the `sisr_enable` bit in the pattern with the actual golden value. This does not loose neither memory nor time because the SISR is then replaced by a simple comparator and each pattern already is 8 bit in size. That way no additional area on the FPGA is used. But logic simulation has to be executed which itself will need a lot of time depending of the size of the RSN and the number of patterns. Especially in combination with 3-valued logic one has to evaluate if area or time is more necessary.

Implementing and evaluating different fault injection mechanism w.r.t. to time and area would be interesting. The time to inject a fault will play a major role in this evaluation. Also these mechanism should be evaluated for their compatibility to different fault models. Fault injection as done in this thesis for example works completely independent of the used fault model as long as the fault can be modeled using basic gates. A multiplexer then will switch between fault-free and faulty. In fact every possible model can be injected at once. But this will create very long combinatorial paths and increase the area needed by even more. Furthermore a comparison of the method implemented in this thesis against the version used in Dunbar and Nepal [DN11] with respect to area and maximum achievable clock frequency is of interest.

Bibliography

[ABH98] J. Abke, E. Böhl, C. Henno. "Emulation based real time testing of automotive applications". In: *4th IEEE International On-Line Testing workshop*. 1998, pp. 28–31 (cit. on p. 19).

[AM97] M. Abramovici, P. Menon. "Fault simulation on reconfigurable hardware". In: *The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (1997). DOI: 10.1109/fpga.1997.624618 (cit. on p. 18).

[BA04] M. L. Bushnell, V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits (Frontiers in Electronic Testing)*. English. Springer, 2004. ISBN: 978-0-7923-7991-1. DOI: 10.1007/b117406 (cit. on pp. 15–17).

[Bar14] R. Baranowski. *Reconfigurable scan networks : formal verification, access optimization, and protection*. en. 2014. DOI: 10.18419/opus-3246 (cit. on pp. 28, 31, 33, 34).

[BKW12] R. Baranowski, M. A. Kochte, H.-J. Wunderlich. "Modeling, verification and pattern generation for reconfigurable scan networks". In: *2012 IEEE International Test Conference*. IEEE, Nov. 2012. DOI: 10.1109/test.2012.6401555 (cit. on p. 31).

[BKW15a] R. Baranowski, M. A. Kochte, H.-J. Wunderlich. "Fine-Grained Access Management in Reconfigurable Scan Networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.6 (June 2015), pp. 937–946. DOI: 10.1109/tcad.2015.2391266 (cit. on p. 31).

[BKW15b] R. Baranowski, M. A. Kochte, H.-J. Wunderlich. "Reconfigurable Scan Networks". In: *ACM Transactions on Design Automation of Electronic Systems* 20.2 (Mar. 2015), pp. 1–27. DOI: 10.1145/2699863 (cit. on p. 31).

[Cab16] J. Cabal. *Simple UART for FPGA*. 2016. URL: https://opencores.org/project,simple_uart_for_fpga (visited on 03/04/2018) (cit. on p. 33).

[CHD95] K.-T. Cheng, S.-Y. Huang, W.-J. Dai. *1995 Ieee/Acm International Conference on Computer-Aided Design: Digest of Technical Papers : November 5-9, 1995 San Jose, California (IEEE ... ON COMPUTER-AIDED DESIGN//PROCEEDINGS)*. IEEE Computer Society, 1995. ISBN: 0-8186-7213-7 (cit. on p. 19).

[CMR+15] R. Cantoro, M. Montazeri, M. S. Reorda, F. G. Zadegan, E. Larsson. "On the testability of IEEE 1687 networks". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. IEEE, 2015. DOI: 10.1109/ATS.2015.7447934 (cit. on pp. 28, 31).

[CY89] W.-T. Cheng, M.-L. Yu. "Differential fault simulation - a fast method using minimal memory". In: (1989). DOI: 10.1145/74382.74453 (cit. on p. 18).

[DN11] C. Dunbar, K. Nepal. "Using Platform FPGAs for Fault Emulation and Test-set Generation to Detect Stuck-at Faults". In: *Journal of Computers* 6.11 (Nov. 2011). DOI: 10.4304/jcp.6.11.2335-2344 (cit. on p. 45).

[Eri02] K. C. Erik Jan Marinissen Vikram Iyengar. *ITC'02 SOC Test Benchmarks. Format*. June 22, 2002. URL: http://www.itc02socbenchm.pratt.duke.edu/format.html#format (visited on 03/11/2018) (cit. on p. 33).

[ERTU07] P. Ellervee, J. Raik, K. Tammemäe, R.-J. Ubar. "FPGA-based fault emulation of synchronous sequential circuits". In: *IET Computers & Digital Techniques* 1.2 (2007), p. 70. DOI: 10.1049/iet-cdt:20050065 (cit. on pp. 18, 19, 32, 37, 43).

[HHW98] S.-A. Hwang, J.-H. Hong, C.-W. Wu. "Sequential circuit fault simulation using logic emulation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.8 (1998), pp. 724–736. DOI: 10.1109/43.712103 (cit. on p. 18).

[HP99] I. Hamzaoglu, J. Patel. "Reducing test application time for full scan embedded cores". In: *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. IEEE Comput. Soc, 1999. DOI: 10.1109/ftcs.1999.781060 (cit. on p. 23).

[IEE13a] IEEE. *IEEE 1800-2012: IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE, 2013. ISBN: 978-0-7381-8111-0 (cit. on p. 33).

[IEE13b] IEEE. *IEEE Standard for Test Access Port and Boundary-Scan Architecture*. 2013. DOI: 10.1109/ieeestd.2013.6515989 (cit. on p. 26).

[IEE14] IEEE. *IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device*. 2014. DOI: 10.1109/ieeestd.2014.6974961 (cit. on pp. 25, 26).

[IK16] A. Ibrahim, H. G. Kerkhoff. "Analysis and design of an on-chip retargeting engine for IEEE 1687 networks". In: *2016 21th IEEE European Test Symposium (ETS)*. IEEE, May 2016. DOI: 10.1109/ets.2016.7519301 (cit. on p. 31).

[ISO12] ISO. "ISO 26262-5:2012 Road vehicles - Functional safety - Part 5: Product development: hardware level". In: (2012) (cit. on p. 16).

[KZL15] R. Krenz-Baath, F. G. Zadegan, E. Larsson. "Access time minimization in IEEE 1687 networks". In: *2015 IEEE International Test Conference (ITC)*. IEEE, Oct. 2015. DOI: 10.1109/test.2015.7342408 (cit. on p. 31).

[LA15] H. Liu, V. D. Agrawal. "Securing IEEE 1687-2014 Standard Instrumentation Access by LFSR Key". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. IEEE, Nov. 2015. DOI: 10.1109/ats.2015.23 (cit. on p. 31).

[LJ17] B. Lavanyashree, S. Jamuna. "Design of fault injection technique for VLSI digital circuits". In: *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. IEEE, May 2017. DOI: 10.1109/rteict.2017.8256869 (cit. on p. 31).

[LN94] R. Lidl, H. Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, 1994. ISBN: 978-0-5214-6094-1. DOI: 10.1017/cbo9781139172769 (cit. on pp. 15, 20).

[Mic03] A. Miczo. *Digital Logic Testing and Simulation*. Wiley-Interscience, 2003, pp. 233–281. ISBN: 978-0-4714-3995-0. DOI: 10.1002/0471457787.ch5 (cit. on p. 15).

[Moo06] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. DOI: 10.1109/n-ssc.2006.4785860 (cit. on p. 15).

[SB98] R. Sedaghat-Maman, E. Barke. "Real time fault injection using logic emulators". In: *Proceedings of 1998 Asia and South Pacific Design Automation Conference*. IEEE, 1998. DOI: 10.1109/aspdac.1998.669529 (cit. on p. 18).

[Sch13] M. Schaal. *Test rekonfigurierbarer Scan-Netzwerke*. de. 2013. DOI: 10.18419/opus-3194 (cit. on pp. 25, 28, 31, 33).

[TJD+16] A. Tsertov, A. Jutman, S. Devadze, M. S. Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, R. Krenz-Baath. "A suite of IEEE 1687 benchmark networks". In: *2016 IEEE International Test Conference (ITC)*. IEEE, Nov. 2016. DOI: 10.1109/test.2016.7805840 (cit. on p. 31).

[UKW17] D. Ull, M. Kochte, H.-J. Wunderlich. "Structure-Oriented Test of Reconfigurable Scan Networks". In: *2017 IEEE 26th Asian Test Symposium (ATS)*. IEEE, Nov. 2017. DOI: 10.1109/ats.2017.34 (cit. on pp. 16, 29, 31, 34, 36).

[USS+17] A. Ullah, E. Sanchez, L. Sterpone, L. Cardona, C. Ferrer. "An FPGA-based dynamically reconfigurable platform for emulation of permanent faults in ASICs". In: *Microelectronics Reliability* 75 (2017), pp. 110–120. DOI: 10.1016/j.microrel.2017.06.032 (cit. on pp. 18, 31).

[WWW06] L.-T. Wang, C.-W. Wu, X. Wen. *VLSI Test Principles and Architectures: Design for Testability (The Morgan Kaufmann Series In Systems On Silicon)*. Morgan Kaufmann, 2006. ISBN: 978-0-1237-0597-6. DOI: 10.1016/B978-0-12-370597-6.X5000-8 (cit. on pp. 15–18, 21).

[Xil18] Xilinx. *7 Series FPGAs Data Sheet. Overview*. Xilinx. Feb. 27, 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (visited on 03/02/2018) (cit. on p. 33).

[ZLJ+14] F. G. Zadegan, E. Larsson, A. Jutman, S. Devadze, R. Krenz-Baath. "Design, Verification, and Application of IEEE 1687". In: *2014 IEEE 23rd Asian Test Symposium*. IEEE, Nov. 2014. DOI: 10.1109/ats.2014.28 (cit. on p. 31).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature