Master's Thesis

Emulation-based Sub-module Activity Analysis to Accelerate Fault Injection Campaigns

Daniel Böhmer, BSc

Institute for Technical Informatics
Graz University of Technology
Austria

Assessor:	Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
Advisor:	Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
	Dipl.-Ing. Dr. techn. Johannes Grinschgl

Graz, April 2013

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

Fault injection is a hardware and software verification method in research and industry. It is used to detect flaws and weaknesses in the hardware description of complex digital designs. It gives the verification engineer the possibility to examine the system-under-test in the occurrence of faults. The emulation of digital hardware designs upon FPGAs has become a standard methodology to test the digital design extensively before going into production. And the emulation of the hardware can further be used to inject faults directly into the design. Such a process is then known as fault emulation.

Usually the verification engineer has to decide how a fault can be injected, but modern system-on-chip designs have gotten very complex and now they provide a big spectrum of fault injection targets. Not all locations and timings can be covered in fault injection campaigns.

This thesis studies and develops a novel method to gain pre-injection knowledge. The activation and the deactivation of selected modules shall give information about possible fault injection targets. The activity of those modules is observed during a normal execution with an expected behavior, also known as a golden-run. Subsequent fault injections afterwards can build up on the knowledge about the activity of those modules. To observe the activity of hardware modules, those modules are extended by sensors. A measurement unit is placed into the system-under-test and the sensors on the modules inform the measurement unit about the activity where it is processed and recorded. The fault injection is performed with an additional fault injection framework. It exists parallel to the activity measurement unit within the digital design. An external controller logic decides afterwards which fault injection will be executed upon the design.

The activity and fault injection experiments were performed on a general purpose processor. A processor usually consists of many different and independent modules. The in VHDL freely available LEON3 processor was selected for the experimentation.

Kurzfassung

Fehlerinjektion ist eine Verifikationsmethode in Forschung und Industrie für digitales Hardware-Design. Damit können Fehler und Schwächen in einem Sicherheitskonzept nachgewiesen werden. Verifikationsingenieure können das System unter der direkten Einwirkung von Fehlern beobachten und evaluieren. Die Emulierung von Hardware-Designs auf einem FPGA ist eine Standard-Anwendung, um das Design während der Entwicklung ausgiebig zu testen. Eine solche Hardware-Emulierung kann nun auch verwendet werden, um absichtlich Fehler während des Funktionsbetriebes in die digitalen Schaltkreise einzubringen. Eine solche Anwendung bezeichnet man dann auch als Fehler-Emulation.

Normalerweise muss der Verifikationsingenieur selbst entscheiden, wo und wie er einen Fehler einbringen soll. Aber komplexe System-on-Chip-Designs (SoC-Designs) bieten ein zu breites Spektrum an Möglichkeiten. Eine vollständige Abdeckung kann manuell nicht erreicht werden.

Diese Arbeit untersucht eine neuartige Möglichkeit, Vorwissen über das System zu erlangen um damit Fehler-Injektionen gezielt einsetzen zu können. Die Aktivierung und Deaktivierung ausgewählter Module soll Informationen über mögliche Fehlerinjektions-Ziele geben. Die Aktivität dieser Module wird während einer normalen Ausführung mit bekanntem Verhalten (Golden-Run) beobachtet. Anschließende Fehlerinjektionen werden mit dem Wissen über die Aktivität ausgewählt und gesteuert. Um diese Aktivität beobachten zu können, werden die Module mit Sensoren erweitert. Weiters wird das System mit einer zentralen Aktivitäts-Messungs-Einheit ausgestattet. Die Sensoren informieren diese Einheit über die aktuelle Aktivität ihrer Module. Die Fehlerinjektionen werden mit einem zusätzlichen Framework eingebracht. Dieses Framework existiert parallel zur Aktivitäts-Messung im digitalen Hardware-Design. Ein externer Controller entscheidet anschließend, welche Strategie zur Fehlerinjektion auf dem Design angewandt wird.

Die Experimente zur Aktivitätsanalyse und zur Einbringung der Fehler wurden an einem *General Purpose Processor* durchgeführt. Ein solcher Prozessor enthält generell ein großes Spektrum an verschiedenen unabhängigen Modulen und eignet sich daher besonders für das Forschungsgebiet der Fehlerinjektionen. In dieser Diplomarbeit wurde der in VHDL frei verfügbare LEON3 Prozessor verwendet.

Danksagung

Ich möchte mich zuallererst bei meinen Eltern bedanken, die mir das Studium der Telematik durch ihren finanziellen und emotionalen Beistand ermöglicht haben.

Ein weiterer großer Dank geht an meine Freunde und Mitstudenten, im Speziellen an die Mitglieder der Basisgruppe Telematik, deren Gemeinschaft mich durch mein ganzes Studium begleitet hat und mich so vieles lehrte, was nicht durch Lehrveranstaltungen abgedeckt werden konnte.

Ich möchte ich mich bei meinem Betreuer Dipl.-Ing. Dr. Johannes Grinschgl dafür bedanken, dass er es mir ermöglichte in diesem Gebiet zu arbeiten und dass er mich mit all seinem Wissen und seiner Geduld während der Diplomarbeit über alle Maße unterstützte. Weiters möchte ich meinen Dank an Dipl.-Ing. Manuel Menghin richten. Er unterstützte mich bei der Korrektur und beim Abschluß meiner Diplomarbeit.

Ein herzliches Dankeschön gilt besonders Ass.-Prof. Dr. Steger. Im besonderen für die Möglichkeit am Institut für Technische Informatik als studentischer Forschungsassistent im Gebiet des Hardware/Software Co-Designs tätig zu sein. Aus dieser Tätigkeit hat sich ein großer Teil dieser Diplomarbeit ergeben.

Acknowledgements

Firstly, I would like to thank my parents, for the possibility to study telematics and for their support and assistance through all the years.

Special thanks to all my friends and fellow students, especially the members of "Basisgruppe Telematik". The community and friendship accompanied me through the entire studies. Also they thought me all those things that lectures can not teach.

I want to thank my advisor Dipl.-Ing. Dr. Johannes Grinschgl. I am grateful for the opportunity to work in this field of research and for all the support and patience he gave to me. Further I would like to thank Dipl.-Ing. Manuel Menghin for supporting me at the end of my thesis.

A very big thanks goes to Ass-Prof. Dr. Steger. Especially for giving me the possibility of working at the Insitute for Technical Informatics as a research assistant in the field of hardware/software codesign. A big part of my thesis arised from this work.

Contents

1 Introduction **11**
1.1 Context 11
1.2 Thesis approach 12
1.3 Thesis organization 13

2 Fundamentals **14**
2.1 Faults 14
2.1.1 Terminology 14
2.1.2 Types of faults 14
2.1.3 The source of faults 15
2.2 Fault injection 15
2.2.1 Physical fault injection 16
2.2.2 Simulated fault injection 16
2.2.3 Emulated fault injection 16

3 Related work **19**
3.1 Optimize fault injection campaigns 19
3.1.1 Using pre-injection analysis 19
3.1.2 Optimize the fault space 20
3.1.3 Detect dependencies 21
3.1.4 Optimizing emulated fault injections 21
3.2 Power emulation 22
3.2.1 Power emulation using access counters 22
3.2.2 Power emulation using sensors on components 22

4 Concept **24**
4.1 General 24
4.2 The LEON3 processor 24
4.3 The Modular Fault Injector (MFI) 25
4.4 The FPGA and development environment 27
4.5 The concept of the activity analysis 28
4.6 Fault injection optimization 30

5 Implementation **31**
5.1 Architecture of the activity measurement unit 31
5.1.1 Activity Sensors 31
5.1.2 The activity measurement 32
5.1.3 Usage on the FPGA 33
5.1.4 Configuration Registers 34
5.1.5 Activity Memory Entry 34

5.2 Modules 35
5.2.1 Multiplier - smul32 35
5.2.2 Divider - sdiv32 38
5.2.3 Data cache - dcache 40
5.2.4 Instruction cache - icache 48
5.2.5 Register file - regfile 48
5.2.6 Gaisler General Purpose Input/Output - GRGPIO 49
5.2.7 AES coprocessor 50
5.3 Fault injection unit 52
5.4 Automatic module analysis 53
5.4.1 Simulation of the modules 53
5.4.2 Preparation of the simulation results 54
5.4.3 The density of the samples 56
5.4.4 Automatic generation of an activity sensor in VHDL 58
5.5 Overview 58

6 Experiments **60**
6.1 The software on the LEON3 processor 60
6.2 The emulation observance software on the PowerPC processor 61
6.2.1 Activity analysis 61
6.2.2 Fault injection control 62
6.2.3 Experiment environment 63
6.3 Fault injections 64
6.3.1 Saboteurs 64
6.3.2 Fault injection with permanent faults 65
6.3.3 Transient faults (randomly and activity based) 67
6.3.4 Results 70
6.3.5 Further observations 70

7 Conclusion **72**
7.1 Summary 72
7.2 Future work 73
7.2.1 Further experiments 73
7.2.2 Automatic detection of relevant signals for the sensors 73
7.2.3 Automatic insertion of sensors 73

8 Appendix **74**
8.1 Compilation of the software running on the LEON3 processor 74
8.2 Synthetization of the LEON3 with the Activity Unit and Fault Injection Controller for the FPGA 74
8.3 Execution of the Activity Analysis with the PowerPC 75
8.4 Simulation with ModelSim for the sampling 76
8.5 Configurations 77
8.5.1 HDL Modules on the AMBA Bus 77

List of Figures

3.1 Pre-injection analysis on resource usage for path based injections, as described by Tsai et al. [31]. 20
3.2 Possible fault injections on a memory location following the principle of Barbosa et al. [9]. 21
3.3 Power emulation concept of Genser et al. [16]. 23

4.1 Main components of the LEON3 processor [4]. 25
4.2 Golden Run flow as proposed by [18]. 27
4.3 An activity sensor of a module. 28
4.4 The module activity send by the activity sensors. 29

5.1 Implementation principle of a sensor. 31
5.2 Modules of the LEON3 processor are extended with sensors and connected to the activity measurement unit. 32
5.3 The LEON3 processor extended with activity sensors and the activity measurement unit that communicates with the PowerPC of the FPGA. 33
5.4 The configuration registers of the activity data collection module. 34
5.5 The arrangement of the modules within the module mask. 34
5.6 A single entry in the memory of the Activity Measurement Unit. 34
5.7 Block diagram of the mul32 module. 36
5.8 Signal waveforms of the multiplier. 36
5.9 Result of the VCD analysis of several multiplier initiations. 37
5.10 Block diagram of the div32 module. 39
5.11 Signal waveforms of the divider. 39
5.12 Result of the VCD analysis of several divider initiations. 40
5.13 Block diagram of the data cache. 42
5.14 Signal waveforms of the data cache on a cache-miss. 43
5.15 High/Low Density for a data cache-miss with only 1 load instruction. 43
5.16 Signal waveforms of the data cache on a cache-miss with two sequential loading instructions. 44
5.17 High/Low Density for a data cache-miss with 2 load instructions. 44
5.18 Signal waveforms of the data cache on a cache-hit. 45
5.19 High/Low Density for a data cache-hit with only 1 load instruction. 45
5.20 High/Low Density for a data cache-hit with 2 load instructions. 46
5.21 Signal waveforms of the data cache with a write instruction. 46
5.22 High/Low Density for a cache write instruction. 47
5.23 Block diagram of the register file module. 48
5.24 Block diagram of the AES module. 51
5.25 Modular fault injector with trigger configuration and saboteurs (S) in multiplier, data cache and instruction cache. 52
5.26 GUI for the MATLAB script to analyse the VCD data. 54

5.27 The distance of sample 1 of a recorded read procedure of the data cache. 56
5.28 The density matrices for the measurement of a read procedure. 57
5.29 The final densities for the measurement of a read procedure. 57
5.30 Data path of the automatic module analysis technique. 59

6.1 Trigger adjustment of the fault injection unit. 62
6.2 Statistic of the fault classifications for randomly injected stuck-at-zero faults. . . 67
6.3 Statistic of the fault classifications for randomly injected stuck-at-one faults. . . 68
6.4 Statistic of the fault classifications for stuck-at-zero fault injections based on activity timing. 69
6.5 Statistic of the fault classifications for stuck-at-one fault injections based on activity timing. 69

List of Tables

2.1 Saboteur operation types [18]. 18

4.1 Example of an activity record for 4 different resources. 29

5.1 A single sample of read procedure with cache-miss of the data cache. 55

6.1 The signals that were equipped with saboteurs. 65
6.2 The outcomes of permanent faults on each signal. 66
6.3 Results of the experiments with transient fault injections. 70

1 Introduction

1.1 Context

Today most semiconductor-based technology has reached a very high level of integration. The density of logic gates within a chip is increasing with every decade. As integrated circuits (IC) get smaller, single gates get more prone to all kind of faults. The reason can be found in the reduced voltage and the reduced internal capacities that increase the sensitivity to noise and radiations [22]. But the small gate level size also allows the integration of further functionality. Today, many integrated circuits are system on chips (SoC). They combine various components of a computer on a single die. Multi-core processors, memories, interfaces, signal processing functions and peripherals on a single chip raise the complexity of the design. And with the increasing complexity the fault threat increases as well.

Faults can be either of natural source or originate from attacks. Especially attack faults are a challenging field in research, since attackers have gotten very creative in bypassing implemented fault recovery mechanisms. Faults that arise from natural sources are mostly dependent on the environment close to the circuit, like electro-magnetic waves or simply alpha-particles or neurons [29], [12].

When a SoC device is designed and implemented, engineers have to make sure that the future device is immune to natural faults and secure to fault attacks. Implemented fault recovery mechanisms have to be tested carefully. Developers also must understand how the device reacts in occurrence of a fault. The more complex hardware devices get, the harder the dependence of different modules can be understood. A fault that may occur in one module may propagate through several other modules until it raises an error or hopefully recovers.

Fault Injection is one possibility to understand how the system reacts under the occurrence of faults. They are induced on purpose by the developers and the effect of the fault is observed in a controlled environment. Fault injection can be done in many different ways. The device can be injected with a fault in its physical form, where it already exists as a prototype. A different method is the simulation of the hardware description. Hardware simulation is used as a functional verification method in early design stages. It is a simple and flexible approach to debug the design. Beside the functional verification, the simulated injection of faults can lead to the detection of design flaws and test specific fault detection and recovery mechanisms. A further important method is the fault emulation. Emulation is the mapping of a hardware design - the system-under-test - into a specific hardware platform [6]. The platform imitates the behavior of the hardware design. That means that the platform can act like the real physical device itself. This is done with the usage of an FPGA (Field Programmable Gate Array).

Those three methods differentiate heavily in their advantages and disadvantages [34]. Using the simulation and emulation method, the big advantage is that a prototype does not have to exist yet. A simulation is very flexible. All signals can be observed and manipulated very easily. It is easy to debug the hardware functionality. However, simulation approaches usually need excessive CPU times, especially for complex systems. This depends mostly on the level of abstraction. While the functional simulation can be fast, a gate-level simulation can be very time consuming for fault injection purposes. The emulation approach overcomes this issue since the hardware is imitated in real-time. However, FPGAs do lack of limited debugging capabilities. Compared to the simulation, the current states of hardware components and their signals can not be observed and manipulated as easy as in simulation approaches. Further techniques have to applied to use fault injection in emulated environments.

The security and dependability verification of system-on-chips is expensive and very time consuming, despite the used techniques and approaches. For example, complex SoC designs have many possible targets for fault injections. It is the responsibility of the verification engineer to select security critical modules and signals for testing. The fault space that covers timing, locality and type of possible faults is usually very big and not even remotely comprehensible. A primary approach is the injection of faults on specific signals and timings. A further approach is to automatically inject faults on a random selection. Both approaches are very exhausting if fault coverage must be achieved. In the first case, the test developer has to decide manually where and when an injection has to take place. In the second a huge amount of faults is going to be induced and their effect has to be analyzed. So research and industry is constantly investigating new methods for security and dependability evaluations.

1.2 Thesis approach

This thesis studies a novel optimization method of a fault emulation technique upon FPGAs. The idea is to raise the number of fault injections that have a higher probability of bringing the system into an unintended behavior. Such faults are denoted as activated faults. On the opposite, the number of fault injection campaigns which do not have any effect to the system should be reduced to a minimum.

The primary approach is to observe and record the activity of different independent modules of a complex System-on-chip design during a normal execution. Such an execution is called a golden run. In subsequent identical executions, faults can be injected automatically into the system using the recorded activity information. Since this information contains the location and timing of active hardware modules, the number of useless faults can be avoided in the first place.

To observe the activity of the components, they are extended with specifically adjusted sensors. The sensors observe the input and output signals, as well as selected registers of those modules. To build up the sensors, every module has to be studied in detail. Two different approaches to create the sensor logic are carried out. The first is the manual inquiry of the module, by examining the hardware description. The second is done on an automatic basis. The complete hardware design is simulated once, whereby the module in question is activated several times. The simulation results are then analyzed by a MATLAB script, whereby the activation is filtered out of the complete signal traces. Similar signal progressions are detected and exhibit the important signals for an activity of the module. The sensor can then be build upon the obtained simulation results.

It is important that the sensors differentiate between an intentional and an unintentional activation. An intentional activation happens, if the output generated by the module is further used. An unintentional activation would be the activation of a module whose input signals are changing, but the output is never considered to be used further on. So, the sensors have to contain some digital logic to decide autonomously when an activation is taking place.

Further, those sensors provide the runtime information to a central component, the activity measurement unit. It consists of a controller, memory and a communication interface. The controller is responsible for the selection of the sensors, the activity recording period and the storage. The communication interface is needed for the configuration and to read the recorded activity information after the golden run. An external controller can download the activity information and decide which strategy for a fault injection shall be used. The golden run, the fault injection experiments and the evaluation of the experiment is done automatically. The fault injection framework from Grinschgl et al. [18] builds the basis for the experimentation.

The approach of detecting the activity is closely related to power emulation approaches. Power emulation is a technique to estimate the current power profile of a system-under-test (SUT). The SUT is loaded into an FPGA and the current needed power can be calculated with power emulation techniques. Of course, simulation approaches or physical measurements can be far more accurate. But emulation techniques were developed to overcome the same issues that exist in this field as in the domain of fault injections. Therefore, this technique can be used as basis for a novel fault emulation method.

1.3 Thesis organization

Next, Chapter 2 introduces the reader into the fundamentals of faults and fault injections. Chapter 3 describes different approaches in science and research to optimize fault injections. It also shows a technique to estimate the power consumption of digital design. Those approaches and techniques inspired the design and implementation of a novel fault injection optimization method. The concept of this new method is presented in Chapter 4. The implementation upon this concept is shown further in Chapter 5. Fault injection experiments are presented in Chapter 6. In Chapter 7, the conclusion to the implementation and the experiments is given. A comparison to the related work of other research is drawn. Also a hint for further work is given. The appendix in Chapter 8 shows the configuration of the software and gives instructions about the usage of the implemented work.

2 Fundamentals

In this chapter the fundamentals of faults and fault injections are described. The different types of faults and their corresponding terminology are explained. Their intentional insertion into a system is called fault injection. In research fault injection is a technique to understand the behaviour of the system under the occurrence of faults.

2.1 Faults

2.1.1 Terminology

When it comes to faults, it is important that the terms are well defined. Avižienis et al. [7] differentiated in his work about the fundamental concepts of dependability between **fault**, **fault tolerance**, **failure** and **error**.

A failure is depicted as an event that occurs when a system fails to deliver the correct service. It is said that the system fails because it does not comply with its specification anymore. An error is the specific part of the systems state that may raise a failure. In fact when an error influences the service of the device, a failure occurs. A fault, which is considered as the cause of an error can be active or dormant [7]. An active fault produces an error, while a dormant fault doesn't. The capability of a system to provide a correct result (the expected service) in the presence of faults is called fault tolerance.

2.1.2 Types of faults

Faults that usually occur within integrated circuits can be differentiated between permanent and transitional faults [34].

- Permanent faults (hard errors) are faults that originate from destructed digital parts. For example, a gate that is not switching anymore is either providing a constant or an undefined signal. Through the influenced structure of the chip, the fault remains even after a reset. So the behavior of the system may not be consistent anymore. An example for such a permanent fault is a single event latch-up (SEL), which is a self-sustained short circuit [8]. It can be caused by short time overvoltage or electrostatic discharge. This may destroy several parts of a component, due to high currents flows through the short circuit.
- Transient faults are faults that only occur during a time period. The fault can either disappear after a few clock cycles, for example because the signal state changes, or it will be removed with a reset. However, the typical classification of this type is that the system can return to its normal, consistent behavior after the fault vanishes. Such a soft error often appears as a single event upset (SEU). This is a bit flip in a register or memory cell. A multiple event upset (MEU) is the occurrence of multiple SEUs at the same time [8]. This fault type is especially important for buses on SoCs.

  Transient faults can arise due to cosmic particles or cosmic rays that interact with electronics, especially in satellite or aircraft electronics at high altitudes. But due to the increasing density of logic gates, the lower supply voltage and the higher clock frequency the number of randomly injected faults is also raising at sea level [27].

2.1.3 The source of faults

The correct function of SoCs depends on many parameters. Bar-El et al. [8] gives an well-arranged overview of causes that provoke faults like SEUs. In the first place, the system developers have to make sure that the device is not exposed to this kind of faults:

- Fluctuation in the supply voltage are well studied and understood. As an overvoltage may destroy the circuits, a too low voltage may provoke the CMOS gates to fail.
- Clock cycle variations may result in asynchronous signal and data propagation.
- The temperature of the device and its environment can have an heavy impact on the function of the device. The manufacturers provide temperature thresholds. As long as the device operates within these thresholds the function of the circuits is guaranteed.
- Electromagnetic waves or radiation can cause spurious currents within the system and provoke a bit flip [34].

Additionally to the previous fault sources, an attacker may use further techniques and possibilities to insert faults to devices:

- With a laser a high amount of photons can be directed onto the circuitry. The photoelectric effect drives a current that itself can drive a gate and inject a SEU.
- X-rays and ion-beams have the same influence, but it is not even required to remove the body of the chip.

2.2 Fault injection

Fault injection is the intended temporal insertion of faults into a hardware design.

Carreira et al. [11] refers to three major benefits of using fault injections:

- To understand the effects of real faults
- To gain feedback for a system correction
- To forecast the expected behaviour of the system

To describe it simply, the motivation is to see how the system behaves when a fault occurs on circuit level. It must be observed if the built-in fault recovery mechanism detects and repairs the fault, or if the system reaches an insecure state.

As mentioned previously in the introduction, there are 3 main approaches to introduce faults deliberately to a system [12]:

- Physical or hardware implemented fault injection
- Fault injections using hardware simulation
- Fault injections using hardware emulation

In research all approaches are examined, but while the physical approach exists since a long time, the emulation approach can be considered as a fairly new domain of fault injections. All approaches do have different advantages and disadvantages and they are needed in different domains, to test a system in terms of security and dependability.

2.2.1 Physical fault injection

Physical fault injection, also called hardware implemented fault injection is done by stressing the hardware [30]. The device itself in its physical representation is attacked. Which means, a prototype is needed for fault injections. Flaws and bugs that are found at this time result in high redevelopment costs. For most of the attacks specific knowledge about the device is needed and for a test developer it is hard to cover the large area of possible internal faults. However, a physical access is very easy for an attacker to a big range of systems, like smart-cards, mobile-phones or multimedia devices. The injection of transient faults by manipulating the supply voltage is very common [29]. Faults can further be injected using electromagnetic interference or pin-level injection. Karlsson et al. [21] states that especially the pin-level injection is the most used technique for physical fault injection. There, the pin of a circuit is directly connected to ground or a voltage source. So research in this field is very relevant in terms of security.

2.2.2 Simulated fault injection

A simulation of the hardware description language is the first choice when it comes to test a design. Simulators are very effective, especially in early design stages when a prototype is still not available yet [12]. For fault injections, simulators offer the most flexible approach since all signals can be supervised and manipulated easily. Some simulators like ModelSim [25] allow the alteration of selected signals to different values. The HDL code does not have to be modified. Rohani et al. [30] divides all simulated fault injection techniques into two groups:

- Those which can use simulator commands, like the previous example.
- Those which need code modifications. The HDL code is extended with mutants and saboteurs. They are described in the next section.

The advantage of simulators is that the fault propagation can be analysed easily. This means that the impact that a fault causes, can be traced until it raises an error or it gets detected and resolved. Depending on the simulation environment faults can be simulated at all levels, e.g. at gate-level (GL), at register-transfer-level (RTL) or at system-level (SL).

A big drawback is that a simulation needs a big amount of CPU time on complex systems, especially for low level simulations. When it comes to simulate a big amount of fault injections, e.g. at the register-transfer-level, the simulation approach reaches its limits very fast. However, research is trying to reduce the needed CPU time by using checkpoints or fault collapsing [30]. Fault list collapsing is an optimization method, in which many similar faults with the same outcome are replaced with a single fault injection. Using checkpoints means that the simulator saves the state of the SUT right before a fault injection, and restarts the simulation exactly from that checkpoint.

2.2.3 Emulated fault injection

Hardware emulation allows the execution of hardware circuits during the development phase. The hardware description is synthesized and loaded into an FPGA. The execution, especially for very complex systems, is much faster than a simulation since it executes in real-time. But it has the disadvantage of a reduced observability and controllability. It is harder to observe and manipulate certain signals directly. Grinschgl et al. [19] states that the advantage that comes with the speed-up compensates the disadvantages. A huge amount of attacks can not be handled with simulations, whereby this can be done with the emulation on FPGAs. There are three main approaches to use fault injection on emulated platforms:

- Partial reconfiguration
- Mutant based fault injection
- Saboteurs

While this thesis concentrate on the usage of saboteurs as the fault injection approach of choice, the others constitute possible alternatives. They can be used similarly with the implemented idea.

Partial reconfiguration allows the change of certain parts of the hardware circuits that are emulated on an FPGA during runtime. While a specific part is altered, the other parts of the system can still operate. This technique has to be supported by the development FPGA and is therefore limited to the chosen platform [18]. There are several methods available to inject faults using partial reconfiguration. One possibility is to modify the bitstream that is loaded to the FPGA. Another approach is, to change the configuration memory of the FPGA [20]. Currently, partial reconfiguration is slower than modern approaches with saboteurs. The reconfiguration processes suffers from long delays, but research deals with the problem of accelerating partial reconfiguration [20].

Mutants are modules in the system that got altered by the test developer. The general idea is that they behave like the original module during normal operation. But when a fault shall be injected for validation then the mutant behaves faulty. For example, it provides an output that is wrong in one bit. This technique is similar to mutation testing in software engineering, where code sequences are altered, e.g. the exchange of an addition operation with a subtraction. This technique has the huge disadvantage that the test developer has to bring a big effort to understand the hardware description. This means that the automatic generation of mutants is also very complex task [30].

Saboteurs are modules that are placed on signals lines. During normal operation of the system-under-test saboteurs are fully transparent. When they are activated, they influence the signal and inject a fault. Table 2.1 gives an overview of possible modes of how a saboteur can disturb a signal.

The consequence of using saboteurs is that the hardware description language has to be modified for the security testing. After changing the hardware description of a system, the synthetisation has to undergo the full process again. And this is a very time consuming process. So it is advisable, to place saboteurs on all security and dependability relevant signals. The saboteurs can then be activated individually through a controller from the outside. So multiple synthetisation processes are avoided.

Complex systems can have a big spectrum of security and dependability relevant signals. This can be a bit-flip on a single signal, but also multiple bit-flips on a bus. For security concerns, multiple fault injections on different signals at the same time have to be considered as well.

Saboteur mode	Fault type	Description
Stuck-at-zero	Permanent	Signal value is set to `0`
Stuck-at-one	Permanent	Signal value is set to `1`
Indetermination	Permanent	Undefined signal state until reload
Bridging fault	Permanent	No output propagation until reload
Negation of input	Permanent	Undefined signal state until reload
Bit-flip	Transient	Output inverts input for one cycle (SEU)
Artificial delay	Transient	Input to output propagation delay

Table 2.1: Saboteur operation types [18].

So the placement of a high number of saboteurs is an exhaustive and time consuming process, when done manually. In the work of Grinschgl et al. [17] a tool is introduced that automatically places a high amount of saboteurs directly into VHDL code [1] . So a maximum test coverage can be obtained by manipulating all security relevant signals. The placement of saboteurs to every signal of a critical module in the design is suggested in this work.

If the amount of saboteurs is very high, the emulation effort raises as well. To keep the cost of the past-injection analysis low, research has to focus several optimization methods. They are discussed in the next chapter.

While saboteurs can imitate nearly all kind of functional faults due to their modes, they have also the disadvantage that they cannot model faults below the gate level [13]. Mutants can do that, because there the fault response is generated internally. So Copppens et al. [13] states that any fault model is possible with mutants.

3 Related work

In this chapter related scientific works are presented. The first part of this chapter, Section 3.1 covers research papers that deal with different approaches to optimized fault injection campaigns. The presented papers show optimal fault injection timings and locations. The paper presented in Section 3.1.4 shows how the impact of emulated fault injections can be supervised.

In Section 3.2 two power emulation concepts are outlined. Power emulation is a concept of estimating the power consumption of digital circuits during the design phase.

3.1 Optimize fault injection campaigns

As many systems are very complex, so are the fault injection validation techniques. As mentioned previously, fault injections in simulations are usually very time consuming. This is why security and dependability verification methods for complex systems are also done with emulation platforms based upon FPGAs. Nevertheless, the analysis of the effect of a single fault injection can still be an exhausting procedure that has to be done by the test developer. In this chapter several research methods are shown. They outline the task of reducing the number of fault injections. The target is to reduce the overhead that comes with the impact analysis of a fault injection.

3.1.1 Using pre-injection analysis

Tsai et al. [31] state in their work

> *Faults should be injected in locations with a high probability of being activated. Conversely, faults must not be placed where activation is very unlikely.*

The motivation to this statement can be found in the fact that a fault that is never activated it simply useless. It should be avoided since it only costs injection time. The injection of faults that are never activated is very likely if the injection targets are chosen randomly. So a smart election of possible targets has to be done.

Therefore, Tsai et al. [31] introduces two basic methodologies to increase the efficiency of fault injections on a micro controller:

- **Stress based injections:** The run-time workload activity at system level is monitored and recorded. This means that the usage of different parts of the system is supervised and fault injections are done based upon these values. For example, if a threshold of the usage per second is exceeded a fault is injected in exactly this part. This is based upon the proven fact that stressed parts of the system are more prone to faults than non-stressed parts are [31].

- **Path based injections:** This is a method that uses the analysis of the program flow and its resources at the application level. Based on this knowledge fault injection targets can be selected more specifically. This includes also the dependence of the program input, as stated by Tsai et al. [31]. But a more generalized term would be that the path based injection is heavily dependent on the used environmental constraints, like input and the beginning state. To guarantee full coverage, the tester has to make sure that every possible path is taken somehow.

As can be seen, both methods build up on the analysis of resource activities. Possible targets are chosen upon this information. Figure 3.1 gives an overview of how this information can be depicted. The resources usage is shown in a sequential manner after the analysis.

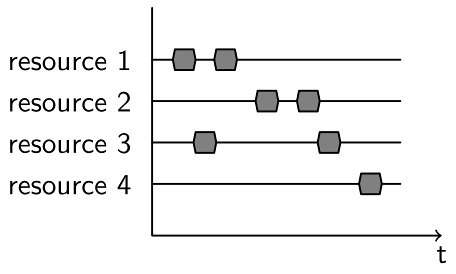


Figure 3.1: Pre-injection analysis on resource usage for path based injections, as described by Tsai et al. [31].

In the experiments of Tsai et al. [31], they analysed the workload by the number of accesses to memory and the time the simulated CPU is active. For path based injection they analysed the assembly language of the executed code on the simulated CPU. They injected faults directly with the simulation environment, e.g. by changes to the system state, like the contents of the registers. Using random injection they had only 1 fault activation on 4 injected faults. But they received a fault injection rate of 100% with their path-based injection method, as they used the knowledge which register is going to be used exactly. Of course, fault activation means that the fault did only propagate through the system, either to an error or to the activation of a fault recovery mechanism. The stress based injection method was very dependent on the used target. While fault injections on the memory had little impact it was very successful at the input/output (I/O) interface to the file system.

3.1.2 Optimize the fault space

The fault space is the quantity of all kind of faults at every possible time at every possible location. Compared to the previous techniques not only the injection targets are taken into account, also their function and timing. Barbosa et al. [9] provided the following motivation as basis for another optimization of fault injection campaigns:

Faults should only be placed in a resource immediately before the resource is read by an instruction.

This statement is very logical, since it makes no sense to inject faults into registers or memory locations that are either not used or overwritten later. By parsing the assembly code of the program that is executed on the microcontroller, the activated resources and their functional usage were detected. Figure 3.2 shows an example of memory accesses and possible fault injection timings.

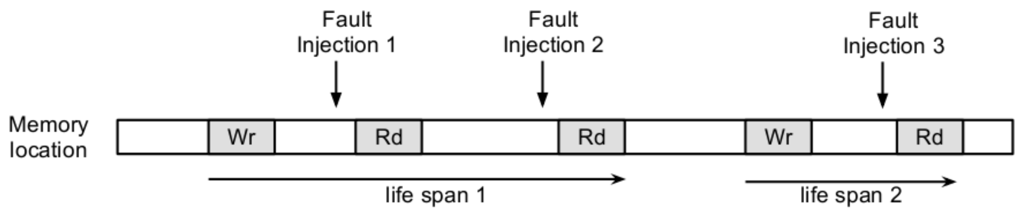


Figure 3.2: Possible fault injections on a memory location following the principle of Barbosa et al. [9].

The assembly code was obtained by disassembling the high-level language code. During the golden run, the program counters and the values of the general purpose registers where traced. So knowledge of used resources including the time, when they are going to be used, was gained. For the fault injections, they targeted the processor registers and some data memory locations of the executed program with single bit flips. Using this technique the fault effectiveness raised one order of magnitude, from approximately 4% to more than 40% for fault injections in the registers and from about 2% to 20% in the memory.

3.1.3 Detect dependencies

Munkby et al. [26] developed a method to detect certain patterns of fault injections. They have created a life span analysis of the operands that are used during execution. They are widening the idea of Barbosa et al. [9] to adapt it to a controller, which is constantly calculating an output out of an input. So as a first optimization of their approach, they target faults only during the lifetime of an operand. For example, the life span of a data variable begins with its initialization and ends with the last read instruction. Further, they created a program dependence graph out of the assembly code. The nodes of the graph represent instructions and the edges the control or data dependencies. This graph visualizes if an instruction uses a variable that is defined by another instruction as a data dependency. Further, a control dependency exists if an instruction is guarded by another, like an if-then-else construct would do.

By visualizing the detected and the undetected injected faults in the graph, they were able to detect similar patterns. Those patterns can be used, to keep the number of needed faults smaller by predicting the outcome of other possible injections. Further the patterns can be used to adapt the fault detection mechanisms in their hardware, to harden the fault tolerance of the device.

3.1.4 Optimizing emulated fault injections

All previously described methods use fault injections on simulated circuits. Simulation approaches have the advantage that the outcome of the fault injection is easier to analyse since the propagation of the fault can be easier followed. A simulation can be compared with white box testing, where internal states and signals can be observed easily. Because the emulated fault injection does not provide exactly this advantage, the impact of an injected fault can not be observed that easily. Research is done to overcome this issue. Grinschgl et al. [19] implemented a module that checks the outcome of an emulated fault injection automatically based on a golden run. Such a golden run, also called reference run, is an execution of the system-under-test without interference of any faults. By checking memory accesses, output signals and the program counter, dangerous alterations of the system-state can be detected at runtime. The fast differentiation between flow, memory and input/output manipulation allows the easier classification of critical attacks.

3.2 Power emulation

There are several different approaches to gain information about the energy consumption of a SoC. Embedded devices need to make sure that the consumption does not exceed the power that can be provided. Especially when the device is powered by battery or by magnetic radio frequency (RF) fields. The requirement for developers is the same as in the security and dependability domain. The information about the future power consumption of a new developed system must be available as soon as possible. Emulation approaches, based upon an FPGA, were developed to avoid the disadvantages of simulation or physical methods. A Physical measurement is the most accurate, but also the most expensive method. When the designed circuitry is already available in its physical form, any changes are probably only going to implemented in the next device generation. Gate-level simulations are very accurate, but unfortunately very slow. This may be a big problem if the circuitry is going to be a very complex one, like a system-on-chip usually is. And especially for software designer the energy consumption of a SoC is interesting. A big potential for energy savings can be found in the application layer [16]. But low-level hardware simulations are often not available to the SoC software designer. They usually use system-level approaches.

3.2.1 Power emulation using access counters

Bhattacharjee et al. [10] developed a power emulation method that is based upon component accesses. They modified the VHDL code of a LEON3 processor to extend the design with their functionality. When selected components start one of their functions, called an event, then a pulse is triggered to a corresponding counter. For example, if the data cache is accessed and produces a cache-hit, then the counter for a cache-hit is incremented. Using this counter value the consumed power for a component can be computed using the following formula

$$P = P_{idle} + \sum_i (E_i n_i f)/cycles$$

E_i represents the energy that is needed for the event i, e.g. a memory access or a cache-hit. n_i is the corresponding count that states how often this event happened. The energy per event was gained by gate-level simulations with micro-benchmarks. A functional simulation with *ModelSim* generated a "Value Change Dump" (VCD) file that contains all signal changes during runtime. This file was given as input to *Synopsys PrimeTime PX* which estimates the power consumption on a gate-level simulation basis.

3.2.2 Power emulation using sensors on components

Genser et al. [16] developed an emulation approach to estimate the power consumption of a SoC. The design of the SoC was augmented with an power estimation unit. Selected components were equipped with sensors, containing a power model that is specific for that model. Register values and input and output signals of an component were routed to the sensor. Figure 3.3 pictures this concept.

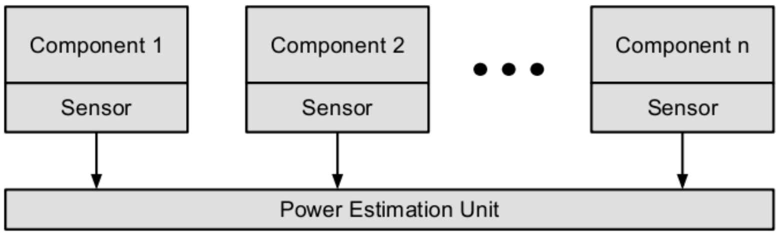


Figure 3.3: Power emulation concept of Genser et al. [16].

Each sensor has a power model stored. It covers enough states so that an accurate overall power estimation can be achieved. The power model was pre-calculated with the value of the signal and register values using a linear regression method.

$$P = \sum_i c_i x_i + \epsilon$$

The values x_i are the current states that consist of the n routed signals from the component. The corresponding coefficients c_i and the deviation ϵ are obtained through a preceding power measurement process. This was done by executing benchmarks on the SoC within a gate-level simulation. The obtained power model calculates the current consumed power y based on the current state.

4 Concept

4.1 General

Goal of this thesis is an optimization of verification processes using emulated fault injections.

The digital design of choice for this thesis is a general purpose processor. Simply because a processor consists usually of big spectrum of different modules. A stable and verified behavior is crucial for its function and a single fault in one module can have a big influence in the overall function. Especially in harsh environments faults can propagate through the entire system, provoking inconceivable system failures. Another central point is the fact that system-on-chip designs have become an ubiquitous part in modern technology. Future trends to smart devices lead to the development of powerful chips containing a big spectrum of different components, interfaces and peripherals. A general purpose processor like the LEON3 from Aeroflex Gaisler [2] is freely available under the GNU GPL open source license. It is designed with VHDL [1] and can be used and modified freely in research and education. The characteristics of this processor is described later in Section 4.2.

The design of the LEON3 processor is extended with functionality that records the activity of selected modules and components. The extension consists of sensors that are implanted into the modules. They detect when a module is getting active and inactive. These sensors sends this information to a measurement unit that records this data, when a given observation period is set. With the records stored in a memory an appropriate fault injection strategy can be chosen and applied. Chapter 5 depicts the implementation of this part.

The fault injection system, called Modular Fault Injector (MFI) is taken from [18]. It comes with a fault injection controller that enables the storage of different fault patterns and the activation of an arbitrary amount of saboteurs. In Section 4.3 the MFI is presented. The hardware description of the modified LEON3 processor is further extended with the modular fault injector and some saboteurs. The saboteurs are placed on all relevant control signals of the modules and components whose activity is supervised.

The modified design of the LEON3 processor is synthesized and loaded upon a development board with Virtex 5 FPGA. The used FPGA is presented in Section 4.4.

4.2 The LEON3 processor

The LEON3 is a 32-bit multi-core processor that implements the SPARCv8 architecture. It was developed by the European Space Agency (ESA) and is now maintained by Gaisler Research. The processor design is a synthesizable VHDL model that is configurable through VHDL generics. The amount of CPU cores can vary between 1 and 4 and nearly all components can be removed, added or adapted by the developer. Components and peripherals are connected over an AMBA AHB and APB bus. Figure 4.1 shows the principle architecture with the main components of the LEON3 processor.

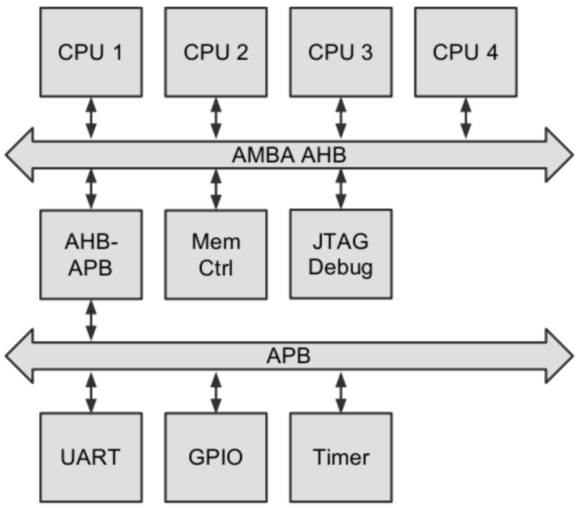


Figure 4.1: Main components of the LEON3 processor [4].

A single CPU core consists of a 7-stage integer unit and a 3-port register file. In the default configuration it includes the parts shown in the next listing. Each of those can be removed to minimize the number of gates:

- A hardware multiplier and a hardware divider module. If they are available, `sdiv` and `smul` instructions can be used in the assembly code. Otherwise the compiler has to implement the calculation within the binary code. With these modules multiplications and divisions can be finished in a few clock cycles.
- A data cache and an instruction cache enables the accelerated access to previously cached data and instructions.
- A floating point unit (FPU) can accelerate operations on floating point numbers.
- A memory management unit (MMU) enables the usage of virtual memory addresses.

During this thesis the FPU and the MMU are not used. So they are not synthesized into the design. For commercial purpose there exists also a fault-tolerant processor that is mainly used for space applications. Unfortunately, it is only available with a commercial license.

4.3 The Modular Fault Injector (MFI)

The modular fault injector that is used for all fault injections during the experimentation phase was taken from Grinschgl et al. [18]. It consists of a fault injection controller, triggers and an arbitrary amount of saboteurs. The saboteurs can be placed on all kind of signals within the LEON3 processor. They support all modes that were described previously in this document in Section 2.2.3 in Table 2.1.

The controller supports multi-bit fault injections. This means that more than a single signal can be disturbed at the runtime of the system-under-test. The disturbance of a complete or even bigger part of a bus can emulate a physical fault attack more precisely than a single event upset could do. In exchange a SEU emulates a fault that can happen on natural basis, like a bit flip induced by alpha particles.

The fault injection controller provides the following functionality:

- It can store multiple different fault patterns within a memory. A single fault pattern consists of the information which saboteurs shall be activated when a trigger is fired.
- It can activate and deactivate triggers. The triggers are responsible for activating the saboteurs at a given time or at a given register value. Time is specified as the number of clock cycles that passes since the activation of the time trigger.
- A general purpose input/output (GPIO) interface is provided as the communication interface for the configuration.
- A saboteur interface that can control an arbitrary amount of saboteurs.
- Saboteurs can be activated for one clock cycle, more than one clock cycles or permanently.

In this configuration the GPIO interface is used to connect the fault injection controller to the PowerPC processor of the development board. The PowerPC can be programmed and controlled by the developer.

There are two different triggers available within the MFI. But an arbitrary amount of triggers can be used:

- A multi-bit trigger that compares a register value with a preset value. If they are the same the trigger sends a hit signal. It is used to supervise the program counter of the LEON3 processor. It triggers the saboteurs when a specific instruction is executed.
- A counter trigger that sends a hit signal to the saboteurs when a previous set number of clock cycles passed by.

The fault injection flow will be done based on a golden run, as proposed in the work of Grinschgl et al. [18]. Figure 4.2 pictures the basic flow of the fault injection strategy. A golden run is a complete run of the system-under-test without any influence of the fault injection framework. But the golden run is supervised by the framework and the result, which is the output and the program flow, is stored. Next, a fault pattern is selected and activated. After a reset of the system-under-test a run with a fault injection is started and the result of this run can then be compared with the golden run. Since such an interfered run can directly lead into an infinite loop, a timeout is needed. After the check of the result the next fault injection pattern is selected. This is done until all patterns are probed.

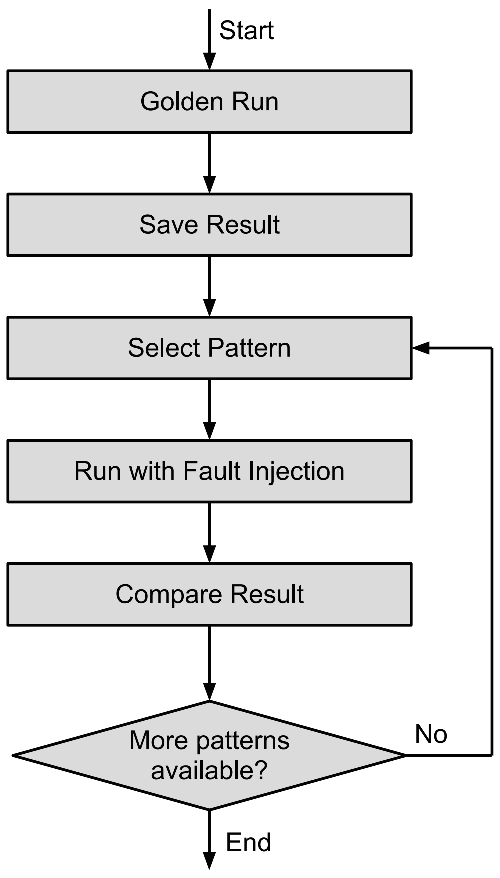


Figure 4.2: Golden Run flow as proposed by [18].

4.4 The FPGA and development environment

Maxfield [24] describes an FPGA (Field Programable Gate Array) as an IC (Integrated Circuit) that consists of configurable logic blocks and configurable interconnections between those logic blocks. The configuration is the programing of the FPGA. It can be loaded with any synthesizable digital hardware description. Usually, FPGAs are used when the functionality of a digital circuit may still be changed later on. Because modern FPGAs can be reprogramed easily. But also because an FPGA is much cheaper than an ASIC (application specific integrated circuit). An ASIC is an IC whose implementation is *"frozen in silicon"* [24]. And the manufacturing of ASICs is far too expensive for little amounts of units. So FPGAs provide a cheap possibility of using system-on-chip designs in special fields.

A very important approach is to test and verify complex integrated circuits during the development. The simulation of big designs is not only a very CPU-time consuming process. A simulation is also an insular environment. Using FPGAs, the design can be used in combination with their future components and peripherals and the function of the design can be verified very soon during the development phase.

Lately, FPGAs are also becoming very cheap and newer applications and usages arise in the industry. The market of reconfigurable computing was split into the following segments by Maxfield [24]:

- **ASIC**: FPGAs are used to protype ASICs, but nowadays when the usage of FPGAs are cheaper than a small amount of ASICs they can also fully replace them.
- **Digital signal processing (DSP)**: FPGAs can be used easily for massive parallel algorithms.
- **Physical layer communications**: Based on the possible high frequencies of the emulated circuits, low level and high level protocols can be implemented in a single device.
- **Embedded microcontrollers**: FPGAs are big enough to contain the functionality of a complete microprocessor and they provide usually enough input and output ports to emulate them.
- **Reconfigurable computing:** Modern FPGAs posses the ability for partial reconfiguration. The hardware description can change during the runtime. This enables many new algorithmic possibilities.

For the experiments and verification of the concept a **Xilinx ML 507** emulation platform [32] builds the basis. The board comes with a Virtex5 FPGA (Field Programmable Gate Array) that is big enough to load the full equipped LEON3 and gives some room for adjustments and enhancements. Additional to the FPGA the platform comes with a PowerPC processor that can be used to control and supervise the emulated hardware. Further the development board comes with all general interfaces like DVI, USB, PS/2, Audio, Ethernet, an LCD and much more.

4.5 The concept of the activity analysis

Similar to the power emulation techniques presented in Section 3.2 an activity measurement unit shall record the timings where a single module is getting active and when it is getting inactive. Modules of the LEON3 processor are extended with sensors that provide the information, when a resource or a module is activated. A sensor observes the input and output signals, optionally also register values (internal states) of the module to detect if a module is active or not. Figure 4.3 shows that principle.

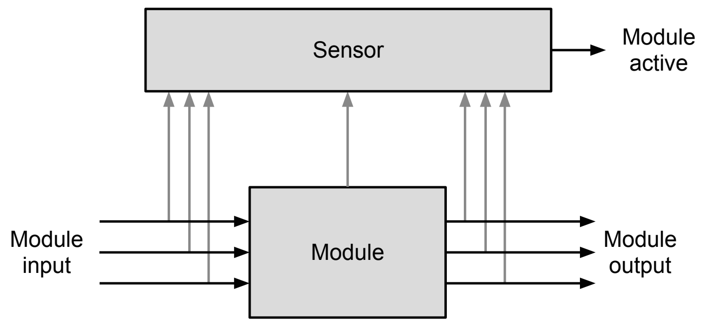


Figure 4.3: An activity sensor of a module.

The input and output ports of the module must be chosen carefully after some investigation of every module. The idea is that the sensor only shows activity when the output of the module is also used after the activity. It maximizes the influence of any fault injection during the recorded activity period. This marks also the big difference to the concept of power emulation. It has to differentiate between a supposed and a not intended activity. The LEON3 processor does not use any clock gating techniques. This means the components of this processor are actually always active. Power emulation has to respect that the modules are consuming energy even when a module is not directly used. Here the system has to make sure, that any component can take inputs and produce output even when the module is not needed. But if this output is not used during the runtime of the system, it was no intended activity and must therefore be ignored. The sensor must contain combinational logic to make this decision. Figure 4.4 shows the signals that are send from the sensors to the activity measurement unit. When the signal is high, then a module is active on purpose.

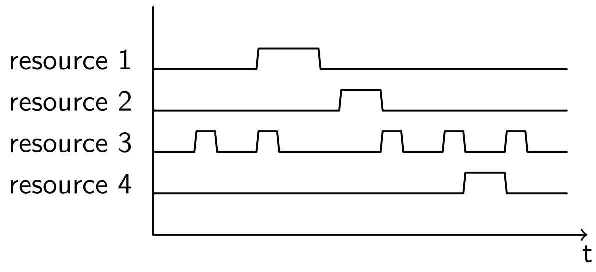


Figure 4.4: The module activity send by the activity sensors.

The activity measurement unit stores the activity states for further analysis. The activity is recorded during a golden run. In a successive identical run a fault can be injected at the time and the location that is achieved through the analysis of the activity. Table 4.1 shows the exemplary activity data record as it would belong to the activity stream shown in Figure 4.4.

time	res1	res2	res3	res4
t_1	0	0	0	0
t_2	0	0	1	0
t_3	0	0	0	0
t_4	1	0	1	0
t_5	1	0	0	0
t_6	0	0	0	0
t_7	0	1	0	0
t_8	0	0	0	0
t_9	0	0	0	0

Table 4.1: Example of an activity record for 4 different resources.

The entry t_9 in Table 4.1 is the final entry that is stored at the end of the recording period. It can be used to gain the information about the duration.

4.6 Fault injection optimization

When it comes to inject a fault into a several parameters have to be chosen at first:

- **The type of fault:** As shown in Table 2.1 in Section 2.2.3 about the fundamentals of fault injections, there are several different fault types that can be injected with saboteurs. For example, a stuck-at-one fault has a completely different influence to the system than a stuck-at-zero fault.
- **The time the fault will be injected:** Especially on SoCs the timing of a fault is an essential parameter. Supposed that a signal is only used in 5% of the time, the other 95% may not have any influence. But to provide nearly full testing coverage many faults have to be injected on the chosen signal.
- **The time duration the fault lasts:** The longer an injected fault lasts, the higher is the probability of getting activated.
- **The signal that is influenced:** SoC consists of a high amount of signals. The more signals are tested, the longer the overall testing time takes. Either the number of signals that should be tested with fault injection methods is kept low by selecting them manually or a pre-injection method tells which signals may be interesting.

Time, time duration and location of the fault can be selected previously with the knowledge about the activity.

5 Implementation

5.1 Architecture of the activity measurement unit

This section describes the architecture and implemention of the activity measurement unit. It is implemented in VHDL. The activity measurement unit consists of sensors and those are extensions to the existing hardware modules of the LEON3 processor. Further the implementation of the activity measurement unit and the usage on the FPGA is described.

5.1.1 Activity Sensors

The basic design of a sensor was shown previously in Figure 4.3. The observed input and output signals of the modules are combined using combinational logic. The activity information is stored within the sequential logic of the sensor and driven to the output of the sensor. Figure 5.1 shows this general principle, which is also know as a two process entity.

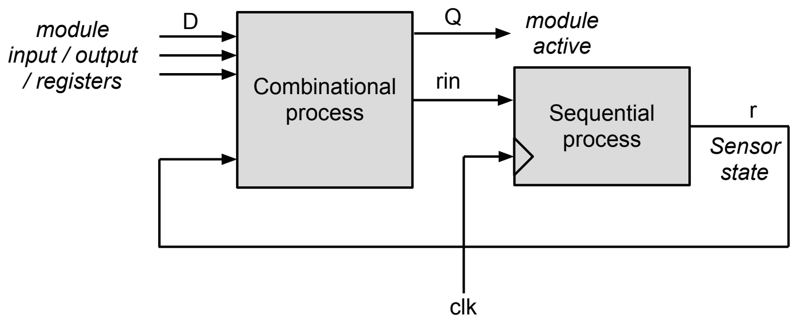


Figure 5.1: Implementation principle of a sensor.

The design principle conforms not only with the VHDL design of the LEON3 processor, it also provides an easy possibility for further extensions and adaptations [15]. By using this design principle, the risk of reaching the critical path is also reduced.

The combinational process takes the module signals D that are observed as input and creates the register states

$$rin = f_r(D, r)$$

and

$$Q = f_q(D, r)$$

whereby rin states the non-registered value and r the register value that is held after an raising clock edge. The output Q is driven within the combinational logic out of the register r. The activation signal will always be set 1 clock cycle too late due to the clocked register. The analysis process of the activity data has to make sure that this value is corrected later on.

A descriptive example of an implemented sensor in VHDL is later shown in Listing 5.3 in Chapter 5.4.

5.1.2 The activity measurement

Every interesting and abstractable module of the LEON3 processor is further extended with such a sensor. The activity signals are directed to the **activity measurement unit**. Next, in Figure 5.2 the concept of the measurement unit with the modules and their sensors is shown. The blue parts are from the LEON3 processor, while the green parts illustrate the extensions. The grey paths show the wiring to the outside controller to program the function and the register values.

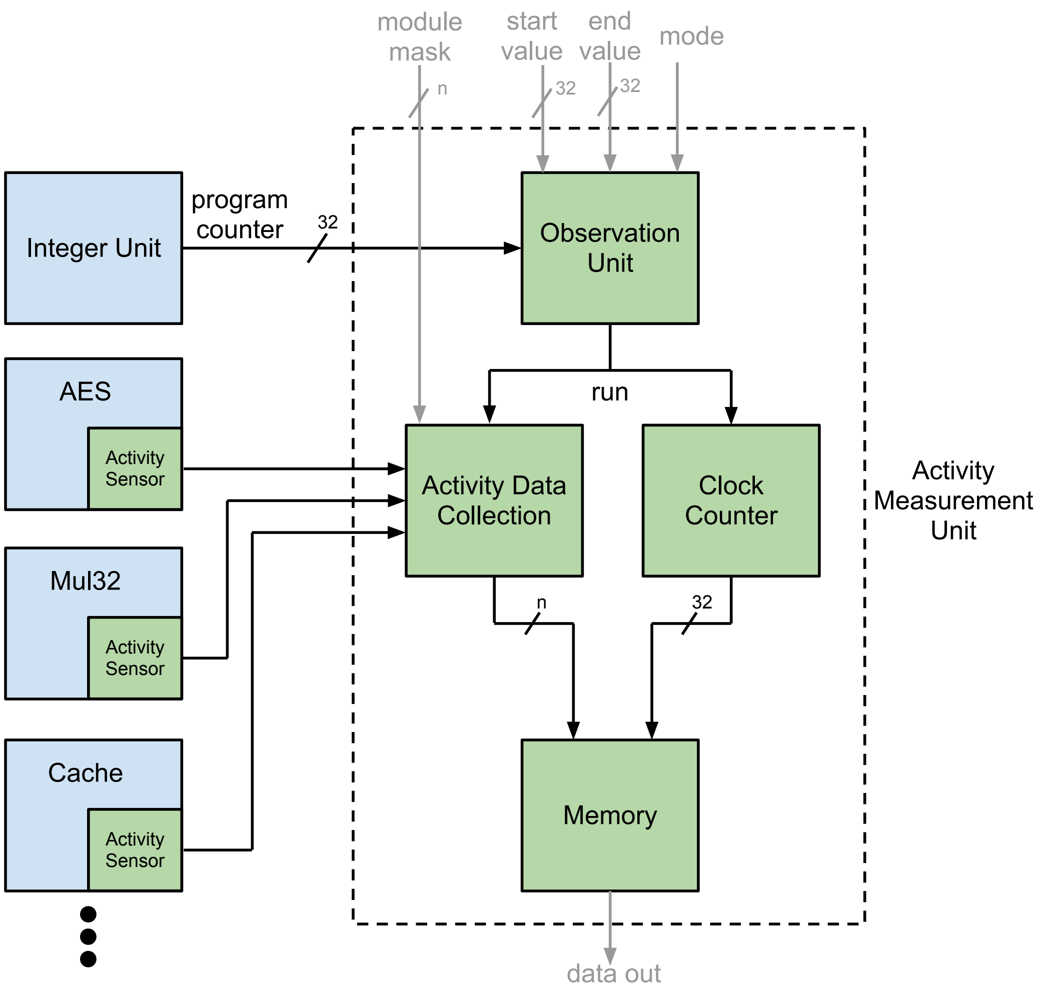


Figure 5.2: Modules of the LEON3 processor are extended with sensors and connected to the activity measurement unit.

The activity unit consists of the following parts:

- **The Observation Unit** keeps track of the program counter of the integer unit. When the program counter of the LEON3 processor reaches the predefined start value the observation unit sets an output signal to high until the end value is reached. The runtime between the start value and the end value is called *observation period*. The start value and the end value can either be programmed by the software that runs on the LEON3 over the AMBA AHB bus or from the outside using a GPIO interface. The GPIO interface is used by the PowerPC processor of the development board. Alternatively, instead of an end value as program counter, the observation period can be stopped after a certain amount of clock cycles. This functionality enables the observation of specific program function and periods.

- **The Activity Data Collection Unit** collects all signals from the activity sensors. When the observation unit informs the data collection unit about an ongoing observation period, the activity data is directed to the memory. A module mask, which can be set from the outside at runtime (AMBA AHB bus or GPIO interface), can deactivate or activate certain sensors.
- **The Memory Unit** receives the data from the activity data collection and stores it for further analysis together with the current clock cycle as time stamp. Furthermore, the memory only stores a new data entry when the vector from the activity data collection changes. This means, that only the changes in the activity stream of the modules are stored, as it was shown previously in Table 4.1.

5.1.3 Usage on the FPGA

The FPGA used for development and experimentation comes with a PowerPC processor to control and observe the hardware that is emulated.

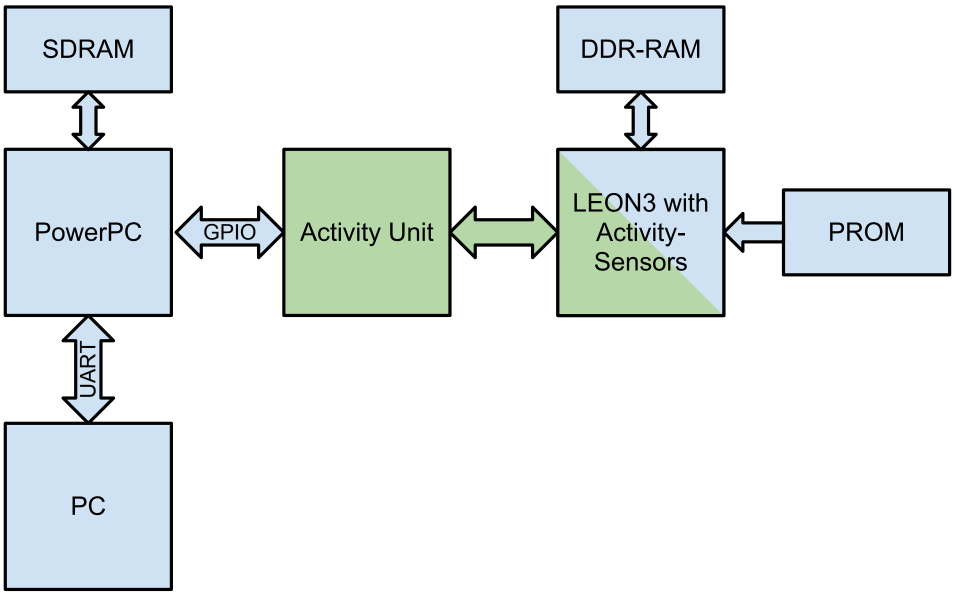


Figure 5.3: The LEON3 processor extended with activity sensors and the activity measurement unit that communicates with the PowerPC of the FPGA.

To configure the start/end value and to read out the memory of the activity measurement unit a GPIO interface connects the PowerPC with the activity unit. The interface enables flexibility and interchangeability. So the hardware design does not depend on the currently used FPGA development board. Any controller can access and implement the GPIO interface. The PowerPC processor itself is connected over an UART interface with the development PC and can be controlled and programmed from there.

When the LEON3 processor is reset, it runs the program that is stored within its PROM memory. The processor executes the instructions and activates and deactivates certain hardware modules. When an observation period is active, the sensors start to notice the activity. It directs the information to the activity unit where it it will be stored in the memory for further analysis.

After the run the PowerPC can read out the memory, filter and process the data and finally start to decide which fault injection strategy is used. This is described later in Section 5.3.

5.1.4 Configuration Registers

As mentioned previously, the registers of the *Activity Measurement Unit* can be set

- by the software that runs on the LEON3 processor out of the PROM using the AMBA interface
- by the supervising PowerPC on the FPGA over a GPIO interface

Figure 5.4 illustrates the configuration registers of the observation unit

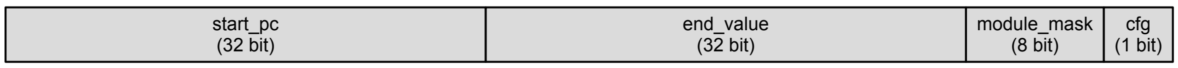


Figure 5.4: The configuration registers of the activity data collection module.

The register `start_pc` is a 32 bit value that is compared with the current program counter of the integer unit. When it is reached, the activity data of the sensors that are set within the `module_mask` is recorded until the `end_value` is reached. This is either the final program counter or a number of clock cycles. If the configuration bit `cfg` is set to `0` the end value is used as program counter. When the bit is set to `1` it is used as clock cycle count.

The module mask is further shown in Figure 5.5. It is an 8 bit register. One bit for each module that is supervised. If a bit is set to `1` the corresponding module is observed and recorded.

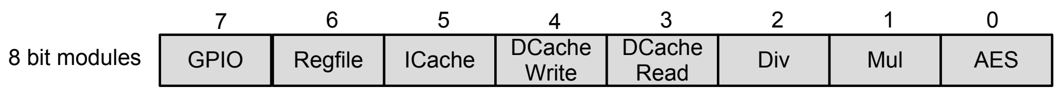


Figure 5.5: The arrangement of the modules within the module mask.

An example that shows how the registers can be set within the native LEON3 software or with the PowerPC is shown in the appendix, Section 8.5.

5.1.5 Activity Memory Entry

The activity data that is collected from the sensors, is put together and stored with a time stamp in a memory that is build up as a FIFO (First In, First Out). An entry in the FIFO is a 40 bit value that consists of a 32 bit clock cycle entry and an 8 bit module entry that shows the active and inactive modules.

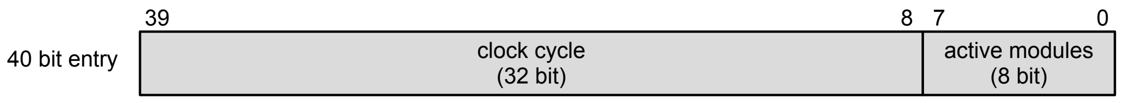


Figure 5.6: A single entry in the memory of the Activity Measurement Unit.

The module entry has the same structure as the module mask, shown in Figure 5.5.

5.2 Modules

For the activity analysis the following modules of the LEON3 processor were investigated and analysed in terms of their intended activity.

- The multiplier of the processor: `mul32`
- The divider of the processor: `div32`
- The data cache of the integer unit: `dcache`
- The instruction cache: `icache`
- The register file: `regfile`
- The general purpose input/output: `grgpio`
- An AES coprocessor: `aes`

Except for the AES coprocessor all investigated modules are original parts of the LEON3 processor, build by Gaisler Research. The AES coprocessor was taken from the Open Cores platform [28] and was built into the processor design. It was connected to the existing AMBA AHB of the LEON3 processor by creating a new bus interface.

In this chapter, all mentioned modules are introduced by their function and configuration. The input and output signals were investigated in several ways. First, a simple simulation of the processor design was taken with ModelSim [25]. The processor design executes a binary which activates multiple times the module in question. By analysing the assembly code of the binary, several program counters were marked as possible starting points. At the times where the program counter of the execution stage of the integer unit reached such a value, the input and output signals of the module were observed manually as waveforms. With the waveforms and the IP manual of Gaisler Research [14] a first impression was made. For some modules this manual analysis sufficed to detect its behaviour. Enough information could be gained about the input and output signal that may indicate the indented activity.

For more complex modules, like the data cache a developed MATLAB script helped to detect those activity periods. This script is described in Section 5.4.

For each module first possible fault injection targets were collected and described in this chapter as well.

5.2.1 Multiplier - smul32

The multiplier is a module that takes two signed or unsigned 32 bit integer values from the integer unit as inputs and produces a 64 bit result as output. The time needed for the multiplication depends on its configuration. While the fastest configuration takes exactly 2 cycles for a 32x32 bit multiplication, the 16x16 bit configuration takes 5 cycles. The difference is, that the first configuration needs approximately 15000 gates while the latter would take about 6500 gates [2]. In this thesis the amount of gates is not the primary factor, but for simplicity the simulated and the emulated processor was equipped with the slowest configuration.

5.2.1.1 Function

The multiplier is activated by the integer unit when a `smul` instruction is executed. For example, the following C code contains a multiplication:

```
volatile int a = 10;
volatile int b = 2;
volatile int result1;

result1 = a*b;
```

This code is then translated and disassembled into the following assembler instructions:

```
4000118c:       c2 03 a0 74     ld  [ %sp + 0x74 ], %g1
40001190:       da 03 a0 70     ld  [ %sp + 0x70 ], %o5
40001194:       82 58 40 0d     smul  %g1, %o5, %g1
40001198:       c2 23 a0 68     st  %g1, [ %sp + 0x68 ]
```

The relevant program counter is `0x40001194`, where an intended activity of the multiplier is started.

5.2.1.2 Signalflow

The block diagram of Figure 5.7 shows the multiplier module with its input and output ports. It can be seen that the input signal `muli.start` and the output signal `mulo.ready` are very descriptive about the functionality.

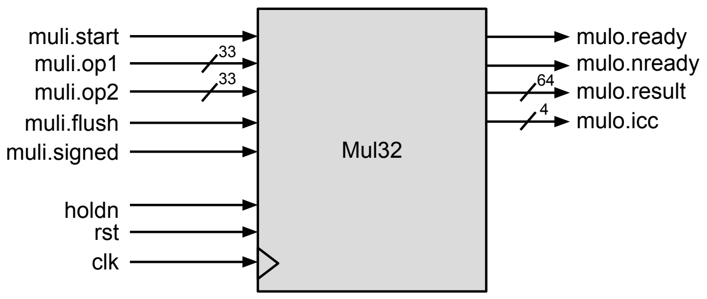


Figure 5.7: Block diagram of the mul32 module.

Next, in Figure 5.8, it is shown how the signals are set by the integer unit when it comes to use the multiplier. As mentioned previously, the starting point of investigation was selected at the position where the `smul` instruction is executed.

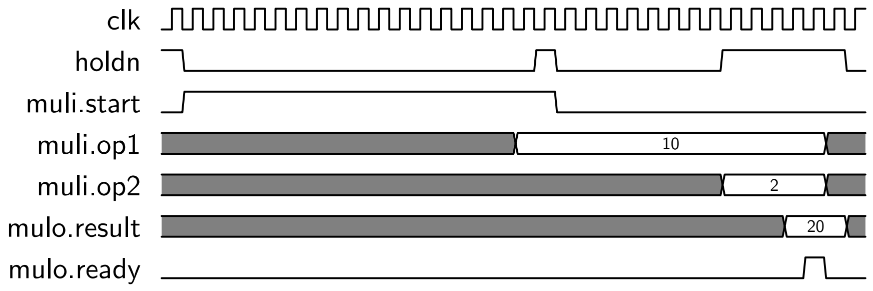


Figure 5.8: Signal waveforms of the multiplier.

It can be seen that the integer unit activates the start signal even when the input data still isn't loaded yet. The signal `holdn` is pulled to low during this time. When the first operand is provided by the memory at `muli.op1` the input signal `holdn` goes to high. The same procedure happens again for the second operand `muli.op2`. The result is calculated and when the output signal `mulo.ready` is set to high the result is provided.

5.2.1.3 Analysis of multiple samples

Since a single simulation record, like the one previously shown in Figure 5.8, cannot really guarantee that all activations follow this certain scheme, a verification method was implemented based on multiple activations of the module. The MATLAB script, more specifically described in Section 5.4, automatically reviews a ModelSim simulation result with several activations. It is capable to detect the relevant signals that are needed for an implementation of an activity sensor.

The next figure shows the result of the simulation analysis. The multiplier was started several times and the script returned the important signals of all supervised signals.

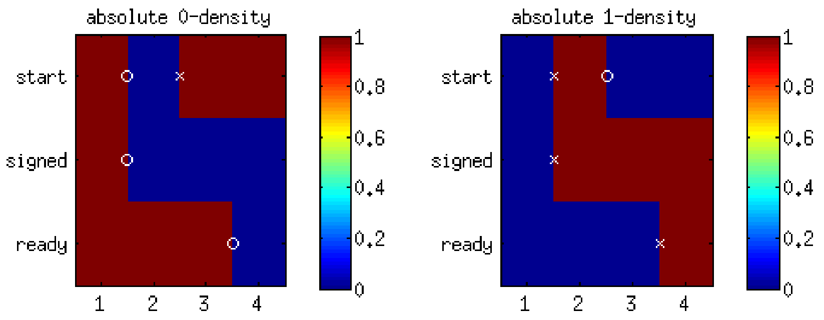


Figure 5.9: Result of the VCD analysis of several multiplier initiations.

First, it can be seen in Figure 5.9 that `start` and `ready` are included. Both rectangles show colored matrices. On the y-axis (rows) the different signals are shown. The x-axis (columns) illustrate the progressing time. The first matrix (left) shows the density of 0-values within all samples. Red entries show a high density, which represents a high appearance of 0 values at this position in time. The second matrix (right) shows the same for 1 values. Since all observed samples show the same behavior there are no densities between the minimum appearance amount and the maximum appearance amount. Therefore, it is easy to declare that `start` and `signed` indicate the start of the activity, while `ready` marks the end. This proves that all waveforms look like the one shown previously in Figure 5.8.

5.2.1.4 Implementation of the activity sensor

With the information gained previously an activity sensor can be implemented very easily.

It can be seen that every multiplication is initiated with the input signal `muli.start` and terminated with the output signal `mulo.ready`. The implementation of the sensor is now a register. It is set to `1` with the start signal and reset to `0` with the ready signal.

As an alternative, the activity in terms of a power emulation can be considered by observing the input signal `holdn`. But for the terms of a fault emulation, it is well-arranged if the activity is represented as a cohesive signal. That means it is simply made up as a single rising and successive falling signal per activation.

5.2.1.5 Possible fault injections

Fault Injections can be done on several locations inside and outside of the multiplier. For example, with the usage of saboteurs:

- The output signal lines `mulo.result` can be disturbed.
- The input signal `muli.start` can be suppressed, provoking the function of the multiplier and confusing the integer unit.
- The output signal `mulo.ready` can be suppressed. This may provoke the same fault as suppressing the `start`-signal.
- Internal registers of the multiplier can be injected with a fault.
- Disruption of the `holdn`-line: If this global signal is set to low, the multiplier is halted. Therefore, the result is provided later than expected.

As stated before, the loading process of the input operators needs more cycles, if the operators have to be loaded from the memory instead from the cache. So it is favored to place the fault injections close to the end of the falling edge of the activity signal.

5.2.2 Divider - sdiv32

The divider performs signed and unsigned divisions of a 64-bit value with a 32-bit value. The GRLIB manual [14] states that the operation needs at least 36 clock cycles. Like the multiplier the output provides a ready signal when the operation is complete.

5.2.2.1 Function

The divider is started by the integer unit when a dividing instruction is executed. The following C-Code shows an example of a division:

```
volatile int a = 10;
volatile int b = 2;
volatile int result1;
result1 = a/b;
```

The C-Code is compiled and disassembled into the following descriptive code:

```
400011ec:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
400011f0:       9b 38 60 1f     sra  %g1, 0x1f, %o5
400011f4:       81 83 60 00     mov  %o5, %y
400011f8:       da 07 bf f0     ld  [ %fp + -16 ], %o5
400011fc:       01 00 00 00     nop
40001200:       01 00 00 00     nop
40001204:       82 78 40 0d     sdiv  %g1, %o5, %g1
40001208:       c2 27 bf ec     st  %g1, [ %fp + -20 ]
```

The interesting assembly line that activates the divider is at the program counter `0x40001204` with the `sdiv` instruction. Here, the values of the registers `%05` and `%g1` are taken. The result is the stored in the register `%g1`.

5.2.2.2 Signalflow

The block diagram in Figure 5.10 shows the divider with its input and output ports. Like the multiplier the divider has very descriptive port names.

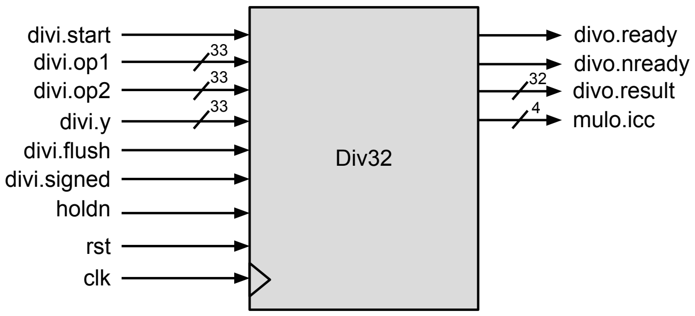


Figure 5.10: Block diagram of the div32 module.

The simulation of a single activation of the divider produces the following signal flow, shown in Figure 5.11.

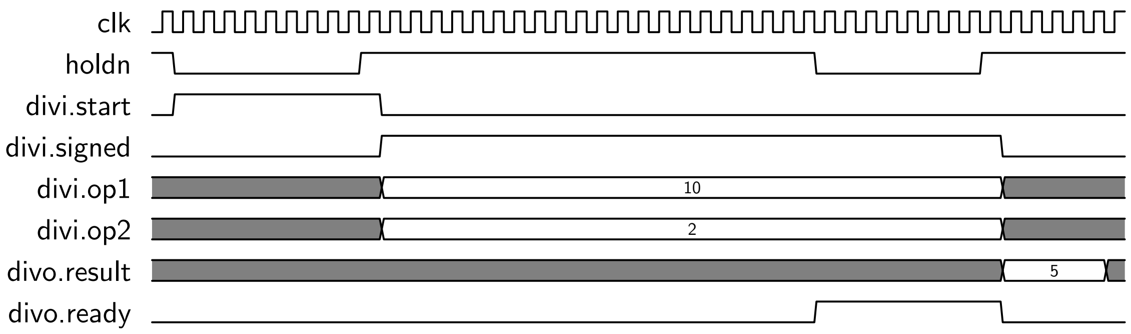


Figure 5.11: Signal waveforms of the divider.

The waveform of the divider is very similar to the one of the multiplier. The signal `holdn` is able to stop the divider when it is set to low. When it is high, the divider is running. The signal is used by the integer unit to keep the module waiting until the cache or other contributive modules of the processor are ready.

Like the multiplier, the divider has meaningful signal names, namely `divi.start` and `divo.ready`.

5.2.2.3 Analysis of multiple samples

Even though the input and output ports do have very descriptive names, the simulation was executed with several initiations of the divider. The automatic analysis of the simulation results is shown in Figure 5.12.

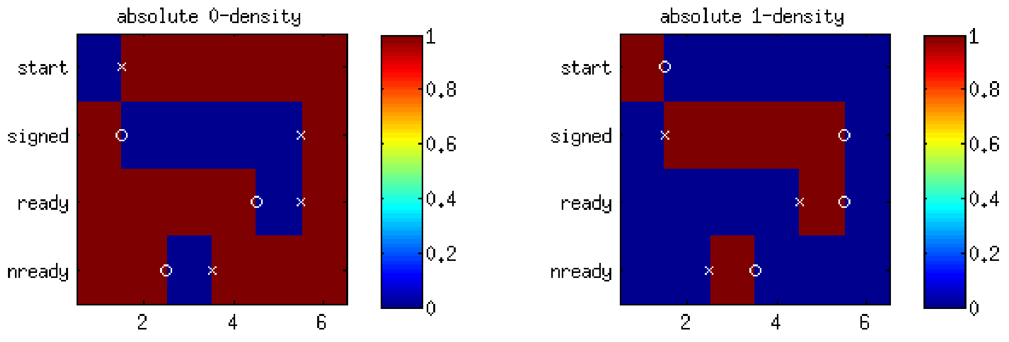


Figure 5.12: Result of the VCD analysis of several divider initiations.

Again all recorded samples look the same because there are only densities at the maximum (red) and the minimum (blue) available. So the waveform in Figure 5.11 are exemplary.

5.2.2.4 Implementation of the activity sensor

Like the multiplier, the divider can be equipped with a sensor using the `start` and the `ready` signals. If the signal `start` is set, the sensor reports activity until the signal `ready` follows.

5.2.2.5 Possible fault injections

The divider can be injected with a fault, for example at the following locations:

- The signal `divo.result` can be modified using saboteurs, when the result shall be provided.
- The output signal `divo.ready` can be suppressed. That may force the integer unit into an unknown state if it is never getting informed about the existing result.
- The input signals `divi.start` may be blocked.
- The internal registers of the divider can be modified, using a mutant.

5.2.3 Data cache - dcache

The data cache is a small memory within the CPU. It stores copies of data from the main memory (RAM) to enable a faster access. When the integer unit reads a data entry out of the memory or writes a value to the memory, a copy of that data is directly saved within the cache. When the same memory location is read again later, the cache can provide the data faster than the main memory. Using a cache a noticeable speed-up is gained for the price of additional chip area.

The data cache of the LEON3 processor is due to the big amount of input/output ports and to its corresponding naming convention, much harder to understand than all the other modules. The data cache also provides a bigger spectrum of functionality that has to be considered when implementing an activity sensor.

5.2.3.1 Function

There are 3 different kinds of intended activity:

- A reading procedure on a not-cached memory location, known as a cache-miss.
- A reading procedure on a previously cached memory location, called a cache-hit.
- A store (write) procedure. When a value is stored within the main memory, it is also automatically written to the cache.

So the data cache is considered as active when the processor executes a load (`ld`) or store (`st`) instruction. The program counter of the integer unit, which is leading to this instruction in the assembly code, has to reach the memory stage. The overlapping of sequential instructions makes it hard to distinguish between upcoming and finished instructions. The principle is called instruction pipelining. The execution stage and the memory stage of the integer unit overlap, when two loading instructions occur consecutively.

As an example, the following piece of C-code statements activate the data cache:

```
int x=5; //global variables, already assigned with values
int y=10;

void access()
{
  volatile int i = x+y; //LD LD ADD ST
}

void access2()
{
  volatile int i = x++; // LD INC ST
}
```

After compiling the C-code into the processor specific machine language, it can be disassembled. This creates an assembly program. It can be analysed and understood more easily, of how the data cache is actually used. Next, the statement block of the function `access()` is shown:

```
400011c4: da 03 40 00  ld  [ %o5 ], %o5
400011c8: c2 00 40 00  ld  [ %g1 ], %g1
400011cc: 82 03 40 01  add  %o5, %g1, %g1
400011d0: c2 27 bf f4  st  %g1, [ %fp + -12 ]
```

The function `access2()` would be compiled and disassembled into the following code block:

```
400011fc: c2 03 40 00  ld  [ %o5 ], %g1
40001200: 98 10 00 01  mov  %g1, %o4
40001204: 82 00 60 01  inc  %g1
40001208: c2 23 40 00  st  %g1, [ %o5 ]
```

It can be seen later on the waveforms, that the difference between a single load instruction and two sequential load instructions has to be respected for building an activity sensor on the data cache module.

The block diagram of the data cache is shown in Figure 5.13. It outlines the fact, that the data cache has a big amount of input and output ports. Their naming convention makes it hard to understand the functionality without digging deep into the VHDL code.

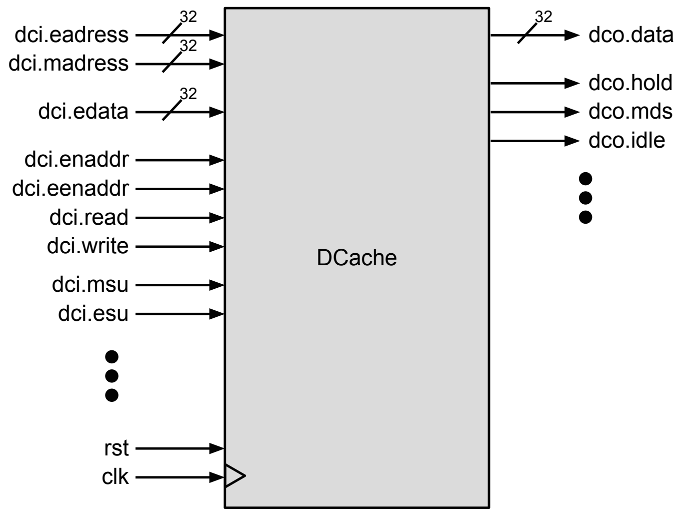
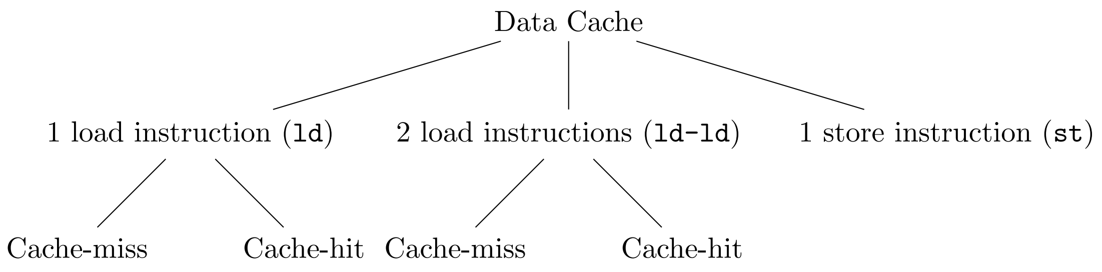


Figure 5.13: Block diagram of the data cache.

5.2.3.2 Analysis of multiple samples

Considered the complexity of this module, the MATLAB script shows its advantage by filtering the signals that do not contain any valuable information. All signals of the data cache were recorded during the simulation of the processor executing a program with several loading and store procedures. But still, different cases have to be considered.



Cache-misses were generated by modifying the processor design. The signal `flush` was set to high manually to clear the data cache. Therefore, the next memory access led to a cache-miss and the value had to be loaded from the memory again.

For the cache-hits the processor design was not influenced. The MATLAB script can ignore the first access to a memory location which is most likely a cache-miss. Successive accesses triggered a cache-hit.

For easier understanding of all those cases the signal flows and their analysis is discussed together in the following chapters.

5.2.3.3 Read procedure with cache-miss

When the processor requests the content of a memory location that is currently not cached, then the data cache has to redirect the loading procedure to the memory.

The first waveform shows the active function `access2()` where a single load instruction is called. It starts when the program counter of the execution stage reaches the load instruction.

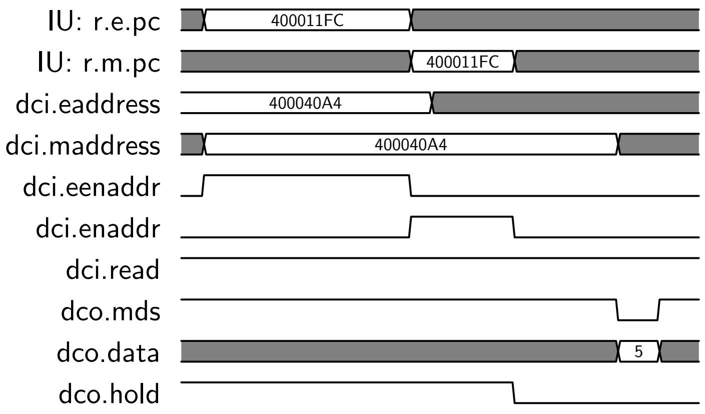


Figure 5.14: Signal waveforms of the data cache on a cache-miss.

The signals `r.e.pc` and `r.m.pc` are the program counters of the execution stage and the memory stage respectively.

Figure 5.15 shows also the densities of the signals with a single load instruction obtained by the MATLAB script.

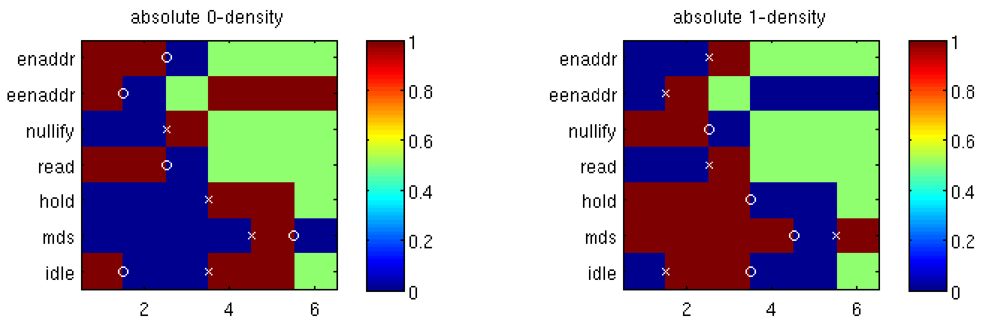


Figure 5.15: High/Low Density for a data cache-miss with only 1 load instruction.

It can be seen, that the signals `eenaddr`, `enaddr` and `idle` give a good indication about the start of the activity. They are all going from logical 0 to 1. The green parts in the densities are signals that were 50% of the activity time high and 50% low. Before any further assumptions can be made, the next function has to be analyzed.

The next waveform in Figure 5.16 belongs to the function `access()` where two sequential loading instructions are executed by the integer unit. These two instructions overlap due to the pipelining of the integer unit.

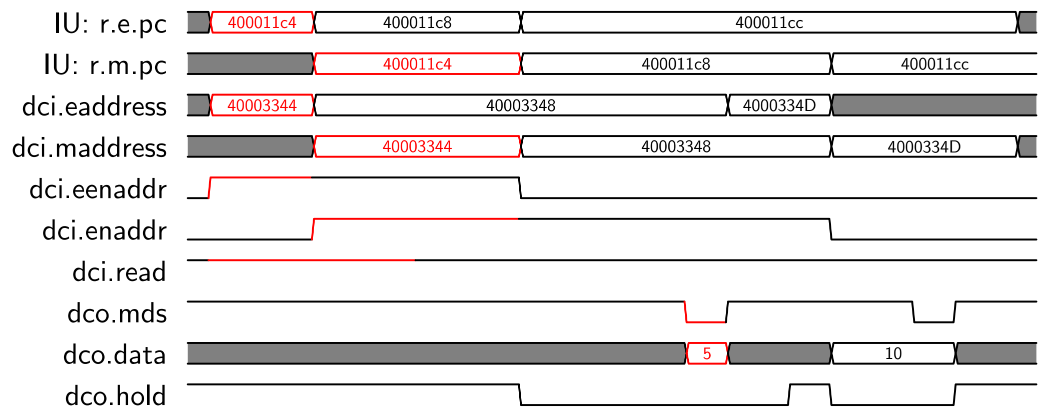


Figure 5.16: Signal waveforms of the data cache on a cache-miss with two sequential loading instructions.

The signals that belong to the first loading procedure are drawn red. When the program counter reaches the value `0x400011c4` the data cache is instructed to load the value that is stored on the memory location `0x40003344`. When `eenaddr` goes high, the value of `eaddress` is read while the function of the signal `enaddr` belongs to `maddress`. Since the value of the memory location is not stored in the cache, the instruction is redirected to the memory by the cache. When the data is available, it is shown by the signal `mds` which is set to low.

The densities of a cache-miss with 2 sequential load instructions in Figure 5.17 shows some interesting and declarative behavior. Especially the signal `mds` that goes to low two times. Also the fact that `enaddr` stays at high during the process gives information about an ongoing activity.

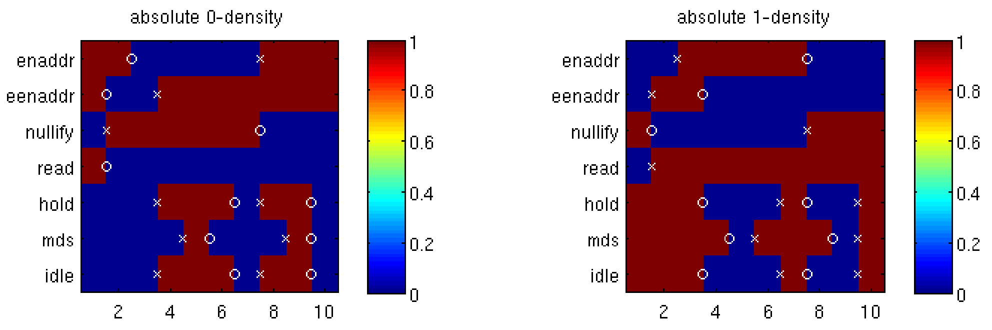


Figure 5.17: High/Low Density for a data cache-miss with 2 load instructions.

So, by analysing the waveforms and densities of a cache-miss the following statements can be made in the first place:

- The reading process is initiated when `eenaddr` and `read` is set to high.
- When the result of a cache-miss is ready, `mds` is set to low, whereby in the next cycle `enaddr` is also set to high.

These waveforms show the problem of creating simple predictions about the intended activity for the data cache. The first observed problems can be summarized to the following:

- The read signal is not a good predictor, since it can stay high even when the results were provided and the data cache is already unused again.

- The pipelining of the integer unit can overlap sequential instructions. So when one result is provided, another result is still upcoming.
- Reading procedures do need different amounts of time. This is due to the memory bus access on a cache-miss.

For more information, the waveforms of cache-hits have to be analysed.

5.2.3.4 Read procedure with a cache-hit

Next, a cache-hit is studied. A cache-hit occurs, when the requested memory value is already held in the data cache. The cache can provide the result faster than the main memory. Therefore, the activity takes less time.

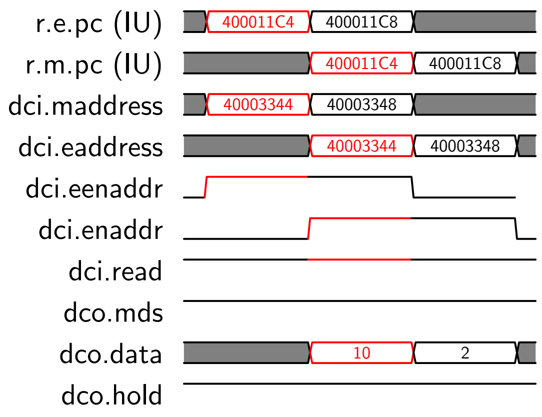


Figure 5.18: Signal waveforms of the data cache on a cache-hit.

The waveforms that belong to the first load instruction are drawn red. A big difference to the cache-miss is that the output signal `mds` is not showing that the result is ready. In fact, the result occurs immediately at the output. But the difference to the cache-miss is shown by the output signal `hold`. It stays high.

The result of the MATLAB analysis in Figure 5.19 shows that the `hold` signal is missing. It was removed automatically since it was a constant value. The signals `eenaddr` and `enaddr` show the same behaviour.

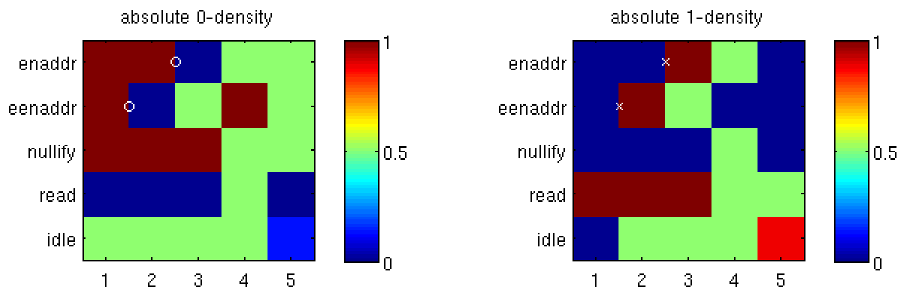


Figure 5.19: High/Low Density for a data cache-hit with only 1 load instruction.

Two sequential load instructions with cache-hits show the same behaviour as presented in Figure 5.20. The usual signals raise as indicated. The signal `read` did stay constant at high and is therefor not shown in the result.

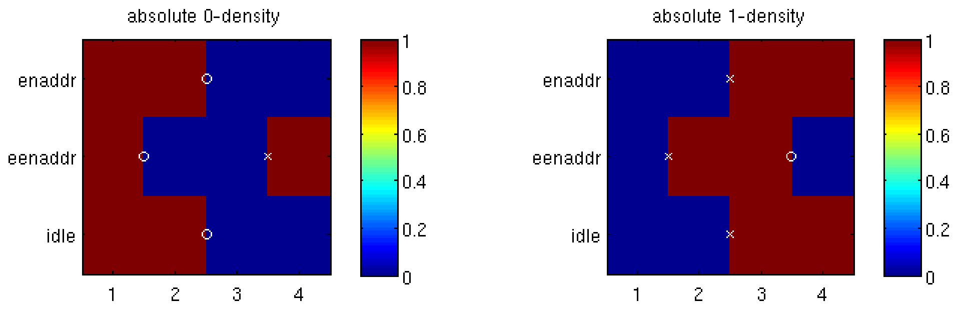


Figure 5.20: High/Low Density for a data cache-hit with 2 load instructions.

The densities reveal that there is not a big difference to the cache-miss. But the missing signals `read` and `hold` outline the importance of these two signals. They are used to differentiate between a cache-miss and a cache-hit.

5.2.3.5 Write procedure

Next, the waveform of a write procedure is shown. The write procedure is initiated when the program counter reaches the value `0x400011D0` as in the example at the begin of this chapter.

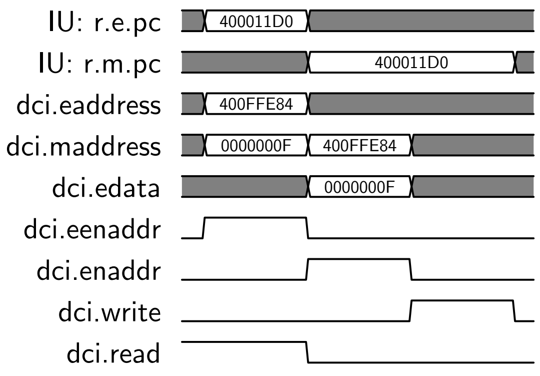


Figure 5.21: Signal waveforms of the data cache with a write instruction.

In this example the value `0xF` (dec. 15) at `dci.data`, which is the result of the addition of 5 and 10, is stored at the memory location `0x400FFE84`.

A writing procedure consists of 3 steps:

1. `dci.eenaddr` is set to high.

2. `dci.enaddr` is set to high, and dci.eenddr goes low again.

3. `dci.write` is set to high, and dci.enaddr goes low again.

The densities in Figure 5.22 reveal that the read signal always goes to low, while the write signal goes to high. Since a writing procedure is done in constant time the write signal suffice as the deciding sensor signal.

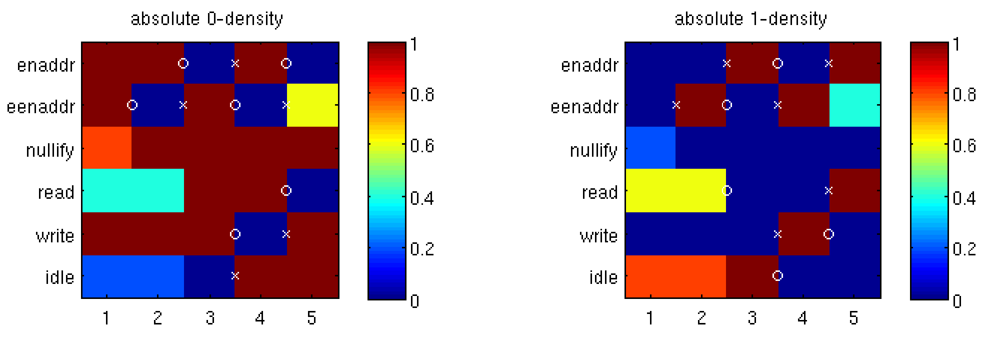


Figure 5.22: High/Low Density for a cache write instruction.

5.2.3.6 Implementation of the activity sensor

Finally, with the previously gained information about the function of the data cache, a sensor can be built . It was implemented in a way so that it can be distinguished between a read and a write procedure.

A write process is detected, if the signal `enaddr` is high but at the same time, the signal `read` is set to low. The sensor deactivates the information of the activity when the output signal `write` is set to high once.

A read process is detected if the input signals `enaddr` and `read` are set to high. If the signal `eenaddr` is also high, then it can be seen that another reading process will be initiated in the next step. Exactly this happens when 2 loading instructions occur sequentially. A binary register flag in the sensor stores this information. If the `hold` signal goes to low after a read instruction was detected, than the process is classified as a cache-miss, otherwise it is a cache-hit. The data of a cash-miss is delivered when the output signal `mds` is set to low. A cache-hit ends when `read` goes to low. If the flag for another load instruction is set, it will be reset and the activity ends not until the same termination sequence as before was observed.

5.2.3.7 Possible fault injections

Since the data cache is a very complex module, there exists nearly an infinite amount of fault vectors that can be injected. The experiments will concentrate on the following ideas.

During a read instruction the input lines `dci.eaddress` and `dci.maddress` can be corrupted, forcing the data cache to either load a wrong memory address or handle the fault. Normally, the data that is provided over `dci.eaddress` will be the same on `dci.maddress` later on. Depending on the pipeline stage of the program counter one of the two data buses are read. The outcome of this fault injection is nearly unpredictable. There is also the possibility of suppressing the output lines of the data cache, forcing the integer unit to show a different behaviour.

During a write instruction the input signal `dci.edata` can be influenced by a saboteur, provoking a wrong data value to be stored in the memory. This is a common fault injection, but the success rate of a fault propagation by using the activity information can be much higher. Also the write signal `dci.write` could be suppressed.

5.2.4 Instruction cache - icache

To speed up an instruction fetch of the processor, the LEON3 also owns an instruction cache. Previously fetched instructions from the memory are stored in the cache. If the same instruction needs to be used a little bit later on - for example within a loop - the fetch processes faster. In contrast to the data cache, the instruction cache does not need to provide a write access. Data can change during the runtime, the instructions can not. Stored instructions are only replaced with new fetched ones.

5.2.4.1 Implementation of the activity sensor

An instruction fetch comes with every change of the program counter at the fetch stage. The integer unit uses the instruction cache with every fetch, independent if the needed instruction is held within the cache or not. If the searched value is not held within the cache, the fetch is handed over to the memory within a clock cycle.

So the activity sensor is implemented only showing cache activity if a cache-hit is happening.

For this purpose, it is enough to monitor the register signal `r.hit` of the instruction cache. Since a cache hit does not take any longer than one clock cycle to process, a stop signal is not necessary.

5.2.4.2 Possible fault injections

Primary ideas of a fault injections within the domain of the instruction cache are:

- The fetched instruction can be sabotaged. So the instruction cache provides either a wrong or invalid instruction.
- The instruction address that is provided from the integer unit to the instruction cache can be modified. Forcing therefor the instruction cache to provide a wrong instruction.

5.2.5 Register file - regfile

The register file consists of an array of processor registers.

The LEON3 register is a 3-port register with two read ports and one write port. It provides parallel read-access to two registers at the same clock cycle. The requested data is delivered instantly after the rising clock edge [14]. Figure 5.23 shows the block diagram of the register file with its input and output ports.

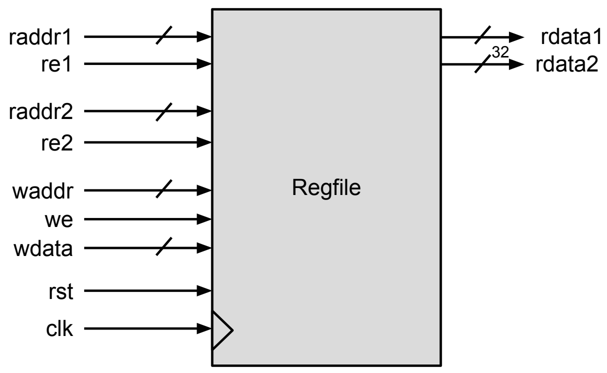


Figure 5.23: Block diagram of the register file module.

The following example shall illustrate the function of the regfile. The assembly code

```
400011a4: c2 07 bf f4  ld  [ %fp + -12 ], %g1
```

instructs the CPU to load a data value out of the memory (directed to the data cache) and to store it in the global register `%g1`, which is managed in the register file. This means at the write-back stage of the integer unit the register file will be instructed to write a value.

The code

```
400011bc: 82 78 40 0d  sdiv  %g1, %o5, %g1
```

has two concurrent read access during the decode stage and later on, after the division again a write access.

5.2.5.1 Implementation of the activity sensor

To detect any started activity within the register file it is sufficient to monitor the read- and write-enable input ports of the module.

Therefore, any activity is ongoing when an enable port is set to high. Such an activity lasts exactly one clock cycle, then in case of a read-procedure, the result is provided. In case of a write-procedure the data is already stored. It can be distinguished if a read, two read or a write procedure is done. Read or write accesses take not more than one clock cycle.

5.2.5.2 Possible fault injections

- The enable lines `re1`, `re2` and `we` can be disturbed. The integer unit will probably read a wrong value. In the case of writing, the register file may store either a wrong value (delay, stuck-at-one) or do not store anything (stuck-at-zero).
- With the usage of a saboteur, the input addresses of a read procedure `raddr1` and `raddr2` can be manipulated. The register file will probably provide wrong or invalid data.
- The output data `rdata1`, `rdata2` can be manipulated directly.
- When the input data `wdata` is manipulated, the register file is forced to store a wrong value. But this injected fault will not propagate until the value is read again.

5.2.6 Gaisler General Purpose Input/Output - GRGPIO

The General Purpose Input/Output (GPIO) consists of pins that are directly connected with the CPU. They can be used as inputs and outputs for any purpose. On the LEON3 they are of arbitrary size that is only defined by the developer. The ML507 Virtex5 FPGA that is used here provides 13 pins. So a 13-bit value can be send over the GPIO at a single clock cycle.

The module GRGPIO that implements the GPIO for the LEON3 processor converts signals that are sent over the AMBA bus by the integer unit directly to the outgoing pins of the processor design.

Next, an example is provided how the GPIO can be used by the software that is executed on the LEON3 processor:

```
int *gpio_reg = (int*) 0x80000800;

int gpio_write()
{
  //Write to GPIO
  gpio_reg[3] = 0;          //IMASK
  gpio_reg[2] = 0xFFFFFFFF; //DIR
  gpio_reg[1] = 0x1;        //DATA
}
```

By accessing the data-part of the array above, it can be read from the GPIO.

5.2.6.1 Implementation of the activity sensor

The activity of the GRGPIO is monitored by observing the register contents of the module. If they are changing then the module is considered as active.

There are two registers within the module. The synchronized register `r` and the combinatorial register `rin`. The data that is applied to the input ports is directly connected to the combinatorial register and at every clock cycle stored in the synchronized register.

So the sensor is easily implemented by comparing the register contents. For example,

```
if( act_grgpio_signals.r.din1 /= act_grgpio_signals.rin.din1 or
    act_grgpio_signals.r.din2 /= act_grgpio_signals.rin.din2 or
    act_grgpio_signals.r.dout /= act_grgpio_signals.rin.dout) then
  v.active := '1';
else
  v.active := '0';
end if;
```

Compared to the sensor methods that were used before, this is a new approach to measure the activity. The activity is only signalled if the register content changes at a rising clock edge. Therefore, the activity is only shown for one clock cycle again.

Since only the data registers are supervised, any activity is not shown if the configuration registers of the GRGPIO are changed. For example, the direction of the channels or the GPIO mask are part of the configuration.

5.2.6.2 Possible fault injections

The propagation of injected faults on any outgoing ports depend on the connected components that are used in combination with the GPIO. However, the effect of a fault injection could be made visible by using a measurement unit (e.g. a digital analyser) on the FPGA.

A fault injection on any ingoing ports, can be analysed more easily. For example, if a known signal is attached to the outside ports. The consequence of faults like stuck-at-one, stuck-at-zero are predictable, though. Therefore, in this thesis no fault injections are undertaken at the GPIO.

5.2.7 AES coprocessor

The LEON3 Processor was extended with a cryptographic coprocessor that implements the Advanced Encryption Standard (AES). The sources for this coprocessor were taken from Open

Cores [28] and an AMBA bus interface was developed to connect the AES coprocessor to the LEON3 processor.

The usage of the AES coprocessor is shown in the appendix in Section 8.5. The functionality of the AES is started in C with the following code lines:

```
void do_aes()
{
  aes_reg->ctrl = 0x3;  //Set Mode (Encyrption Bit0 =1) and Start (Bit1 = 1)
  while(aes_reg->ctrl != 0x1); //Wait for result
}
```

The address to access the registers of the coprocessor `AES_ADDR` is set in hardware and depends on the AMBA slot-id, where the coprocessor was placed into the design.

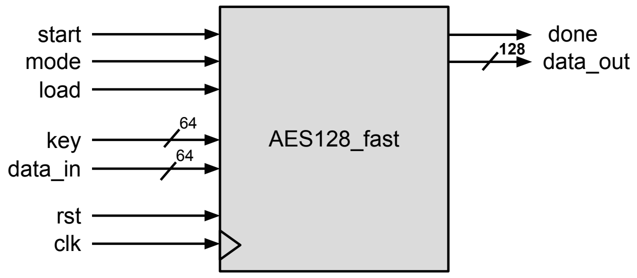


Figure 5.24: Block diagram of the AES module.

5.2.7.1 Implementation of the activity sensor

The activity sensor observes the signals `load`, `start` and `done`. In this implementation a finite state machine (FSM) is started, when the start bit in the control register is set. This FSM controls the module and starts with the load signal. It is sufficient, to define the activity of the AES module with the starting signal `load` and keep it until the signal `done` is set.

5.2.7.2 Possible fault injections

In science it is very popular to attack several specific registers, to gain the key of the encryption process. The AES module i s not fault injected directly within this thesis. But in the experiments it will be observed if the result of the AES computation can be disturbed by injecting faults into the other modules.

5.3 Fault injection unit

The fault injection unit is built up with the modular fault injector (MFI) of Grinschgl et al. [18]. A description of its functionality and parts is shown in Section 4.3. For this thesis the following parts from the MFI were taken:

- The fault injection controller: It is connected to an additional GPIO port to communicate with the PowerPC processor of the development board. The fault injection patterns are sent from the software of the PowerPC to the controller. This is later described in Chapter 6, showing the experiments.
- Single bit saboteurs are brought onto several signals of the modules that were selected for activity observation. The specific signals are presented later in Chapter 6.
- Two triggers are put together for the activation of the saboteurs. The first trigger is connected with the register that stores the current program counter of the execution stage of the LEON3 processor. It is activated, when a run with a fault injection is started. A second trigger is activated when the first trigger fires. The second trigger is a time trigger. It counts the number of clock cycles that pass since it was activated. When a preset count is reached it fires as well. This hit signal is routed back to the controller. When the second trigger fires, the controller activates the saboteurs that were specified through the pattern.

Figure 5.25 shows the architecture of the modular fault injector that is implemented parallel to the activity measurement unit into the hardware design.

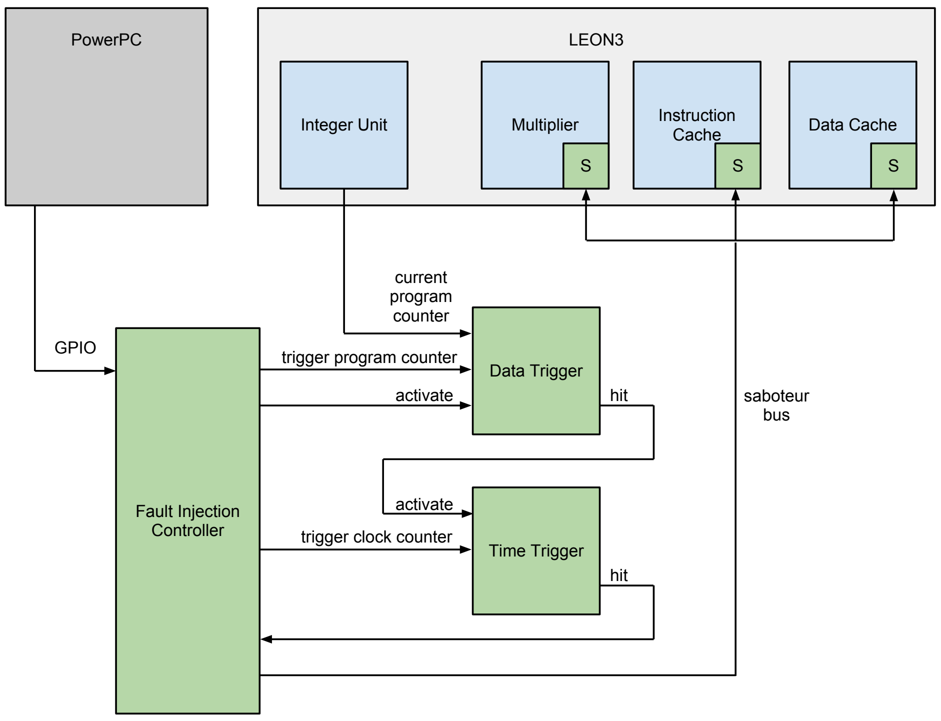


Figure 5.25: Modular fault injector with trigger configuration and saboteurs (S) in multiplier, data cache and instruction cache.

5.4 Automatic module analysis

To understand the functionality of a single hardware module easier, a MATLAB script was written that investigates the input and output signals of such a module. The input and output signals are recorded using the HDL simulator ModelSim. The simulator generates a VCD-file (Value Change Dump) that contains the record of selected signals over the simulation period. During the simulation, the module that is investigated is started several times with different environment parameters. The MATLAB script parses the VCD-file and extracts and filters the information that is needed to specify the important signals and timings to declare a module activity. The knowledge, when a module may be activated and when it will be deactivated, has to be provided by the user. These are:

1. **A start point:** The program counter whose related code instruction will propably activate the module.
2. **An end point:** This can be a known return value of the module or the next program counter

5.4.1 Simulation of the modules

As mentioned previously, ModelSim is used as HDL simulator. The LEON3 sources can be simulated out of the box and were only extended with the AES coprocessor and the AMBA AHB interface, described in Section 5.2.7.

A do-file specifies the configuration for ModelSim to create the VCD-file during simulation. It contains all the signals that shall be recorded. Next, an abridgement shows the content of a do-file:

Listing 5.1: Excerpt of the do-file for the VCD signal recording of the data cache module (vcd_dcache_miss.do)

```
vcd file vcd_dcache_miss.vcd

vcd add /testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.e.ctrl.pc(31)
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.e.ctrl.pc(30)
                                                        ...
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.e.ctrl.pc(0)


vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dci.enaddr
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dci.eenaddr
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dci.nullify
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dci.read
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dci.write
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.mexc
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.hold
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.mds
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.cache
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.idle
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.scanen
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.testen

vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.data(0)(31)
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.data(0)(30)
                                                                  ...
vcd add /testbench/cpu/l3/cpu(0)/u0/p0/m0/c0/dcache0/dco.data(0)(0)
```

The first lines define the program counter of the integer unit (`r.e.ctrl.pc`). Then the control signals that actually should be investigated are specified. And finally the delivered data of the operation, which is in this case the data that the data cache delivers.

In this case, the program counter and the result is the a-priori knowledge, needed to define the module activity that is searched.

Listing 5.2: Excerpt of the code that instantiates data cache

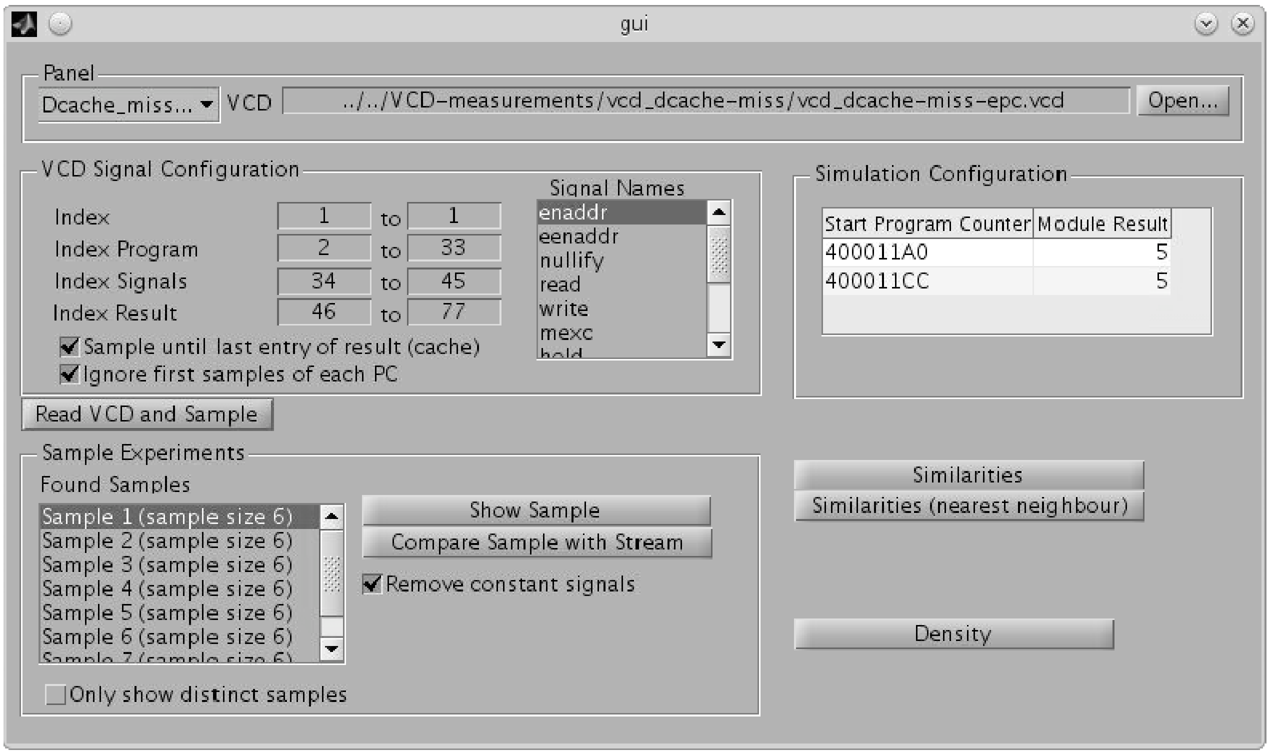
```
400011a0:        c2 04 23 04      ld   [ %l0 + 0x304 ]

400011b4:        d8 03 63 00      ld   [ %o5 + 0x300 ]
400011b8:        c2 04 23 04      ld   [ %l0 + 0x304 ]

400011cc:        c2 04 23 04      ld   [ %l0 + 0x304 ]
```

After the simulation of the Leon3 HDL design the file `vcd_dcache_miss.vcd` will be created.

5.4.2 Preparation of the simulation results

The MATLAB script takes the previously generated VCD-file as input, parses it and analyses the gained signal information.

To handle the functionality of the MATLAB script and the VCD-file, a graphical user interface (GUI) was build. The position and ranges of the signals within the VCD file can be set here manually or chosen from a previously stored setting.


gui

Panel: Dcache_miss... VCD ../../VCD-measurements/vcd_dcache-miss/vcd_dcache-miss-epc.vcd

VCD Signal Configuration

Index	1	to	1
Index Program	2	to	33
Index Signals	34	to	45
Index Result	46	to	77

[✓] Sample until last entry of result (cache)
[✓] Ignore first samples of each PC

Signal Names: enaddr, eenaddr, nullify, read, write, mexc, hold

Simulation Configuration

Start Program Counter	Module Result
400011A0	5
400011CC	5

Sample Experiments — Found Samples: Sample 1 (sample size 6), Sample 2 (sample size 6), Sample 3 (sample size 6), Sample 4 (sample size 6), Sample 5 (sample size 6), Sample 6 (sample size 6), Sample 7 (sample size 6)

[✓] Remove constant signals
[ ] Only show distinct samples

Figure 5.26: GUI for the MATLAB script to analyse the VCD data.

The GUI, that controls the functions of the script provides the following features:

1. With a click on the button "*Read VCD file and sample*" the VCD-file specified at the top of the GUI is parsed. A list of signal states for every time stamp in the VCD-file is created. The provided program counters in the table at the left hand side of the GUI illustrate the beginnings of each activation and the corresponding results of the operation mark the end. The signal states between each start and end are stored as a sample. 20 activations of the modules means that 20 samples are created.

2. A created sample can be selected and shown. Table 5.1 shows an example of such a sample.

Signal	t_1	t_2	t_3	t_4	t_5
enaddr	0	1	1	1	1
eenaddr	1	1	0	0	0
nullify	0	0	0	0	0
read	1	1	0	0	0
write	0	0	0	0	0
mexc	0	0	0	0	0
hold	1	1	0	0	0
mds	1	1	1	0	1
cache	0	0	0	0	0
idle	1	1	0	0	0
scanen	0	0	0	0	0
testen	0	0	0	0	0

Table 5.1: A single sample of read procedure with cache-miss of the data cache.

A sample consists of signal changes that happen between activation and deactivation of the module. The time dimension was not only discretized, it was actually removed. The LEON3 processor may stop or hold any modules until a certain value is loaded while the module is already activated. So the time can be usually of arbitrary length, especially when it comes to memory access. To remove any dependency on this behaviour a sample consists only of successive signal changes.

3. A sample can be selected and compared with the whole signal stream. This function calculates the euclidean distance of the sample signals with the whole record and can show how many similar sample are there. Figure 5.27 shows an example. At the points, where the distance is zero the same samples are found in the stream. If the points are close to zero, there exists a slight difference. This functionality enables to analyse and detect module activations that are very different to all the other samples. But they can vary in their length since a single change in one signal adds a further discrete row, like a new time stamp.

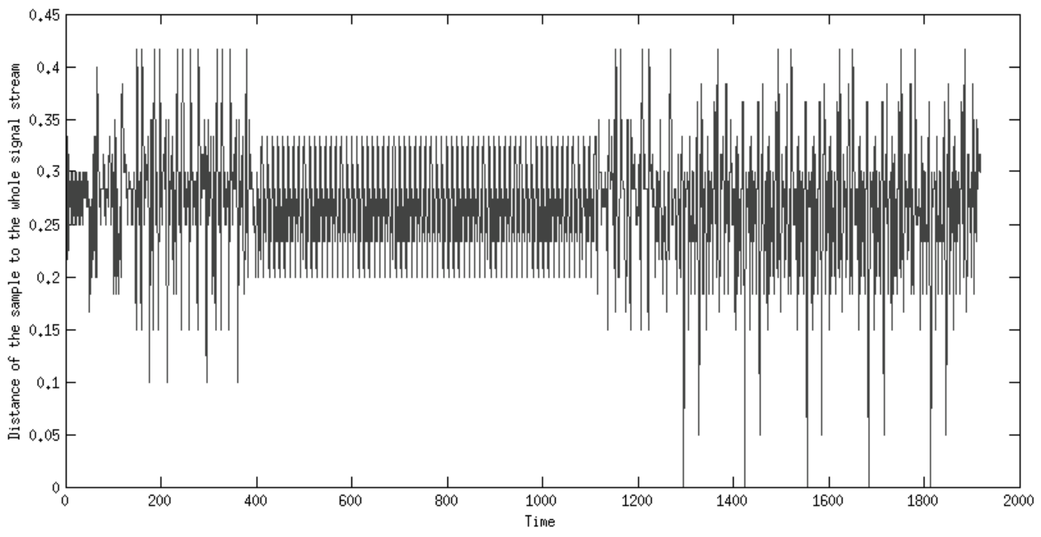


Figure 5.27: The distance of sample 1 of a recorded read procedure of the data cache.

4. The analysis of all samples by creating density matrices. They represent the amount of zero or one values at certain positions of each signal within the samples. The density matrices are ploted and marks within the plots indicate important signal changes. The information that those plots offer is used for the implementation of the activity sensor.

5.4.3 The density of the samples

To gain the densitiy matrices out of the previously collected samples, the following steps had to be done by the MATLAB script:

1. **The elimination of constant signals:** In the example in Table 5.1, it can be seen that signals like `scanen` or `testen` are always zero. In fact, they are zero in all recorded samples. They are therefore unnecessary and can be removed from further investigation.

2. **The calculation of the densities of samples with equal size:** In the first step, the different sample sizes are treated separately. That means for every sample size a distinct density is calculated. For every `0`-value in a sample, a counter at exactly this position in the `0`-density matrix, is increased. The same procedure is undertaken for `1`-values.

   Figure 5.28 shows an example, where a measurement of all loading instructions of the data cache within a specific function returned samples with 6 and 9 columns.

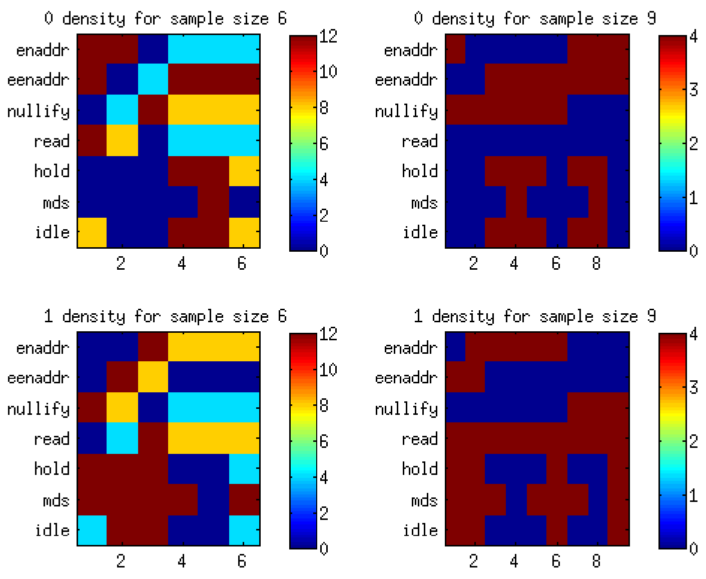


Figure 5.28: The density matrices for the measurement of a read procedure.

While this density matrices look confusing at the first sight, they show a very logical information. The densities (values within the matrices) are illustrated by colors. In the first row they two matrices shows the appearance of `0` values within the samples. Colours close to red show a higher appearance, colours close to blue indicate a minor presence. In the next row, this is shown for the `1`-values. Of course, the addition of `1` and `0` density would be a constant matrix.

3. **The final densitiy matrices** are gained by merging the matrices of different sizes. It has the size of the biggest density matrix gained previously. This is done, by adding up the values in the bigger matrix, where the signal row of the smaller matrix fits. This fitting position is gained by retrieving the shortest distance of a signal vector of the density matrix to the final matrix.

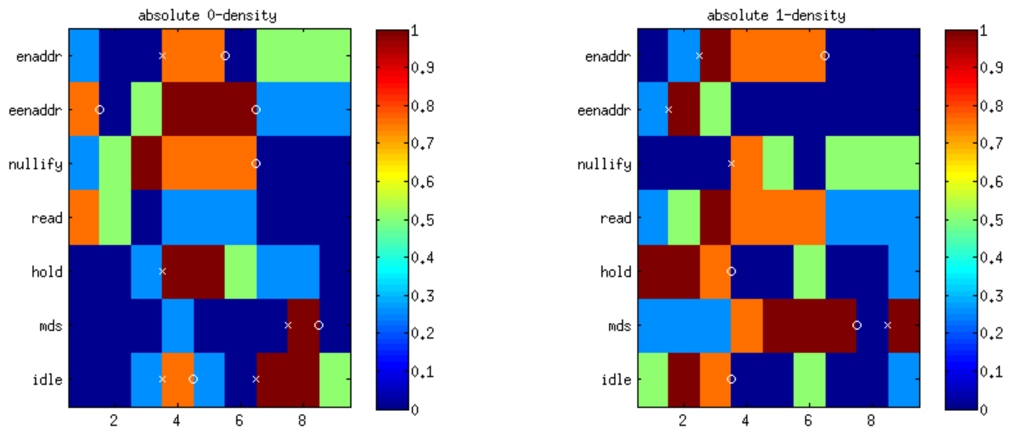


Figure 5.29: The final densities for the measurement of a read procedure.

The size of the density matrix can be reduced, if there are columns in the matrices, that do not contain any bigger raising signals or lowering signals. Since they do not contain relevant signal changes, they can not provide feature information.

At the end, the important signal transitions can be detected. Such a transition is an apparent raise from a low density to a high density and vice versa. They are called features.

5.4.4 Automatic generation of an activity sensor in VHDL

A VHDL sensor file can be generated automatically out of the features, gained from the density matrices. The features have to fulfil certain requirements:

- They must exist in both density matrices at the same position.
- Each feature must have an opposite feature (raising feature, lowering feature) in the same position.

With the found features, the basis for an activity sensor can be created automatically in VHDL for the investigated module. Listing 5.3 shows an example of the sensor that was build out of the found features in Figure 5.29.

Listing 5.3: Except of the automatic created VHDL code

```
comb: process(act_Dcache_signals, activate, r)
  variable v : reg_type;
begin
  v := r;
  if(activate = '1') then
    if(not act_Dcache_signals.eenaddr and
       act_Dcache_signals.enaddr = '1') then
      v.active := '1';
    elsif(act_Dcache_signals.mds = '1') then
      v.active := '0';
    end if;
  else
    v.started := '0';
  end if;

  rin <= v;     -- Drive register

  act_Dcache_active <= r.active;     -- Set Output
end process;
```

The basis consists primary on the first found features in the stream that generally initiate the activity and the features that indicate the end of the activity.

The features that indicate a raise in the signal stream can be combined with a simple `add`-operation. If the stream lowers instead, a `not`-operation will be put in front of the signal name.

However, the function of the automatic code generation depends on the search of correct features and therefore on the configuration of the threshold that indicates a raising or lowering transition.

5.5 Overview

Finally, Figure 5.30 should give an overview over the data path of this technique for the automatic analysis of multiple module activations.

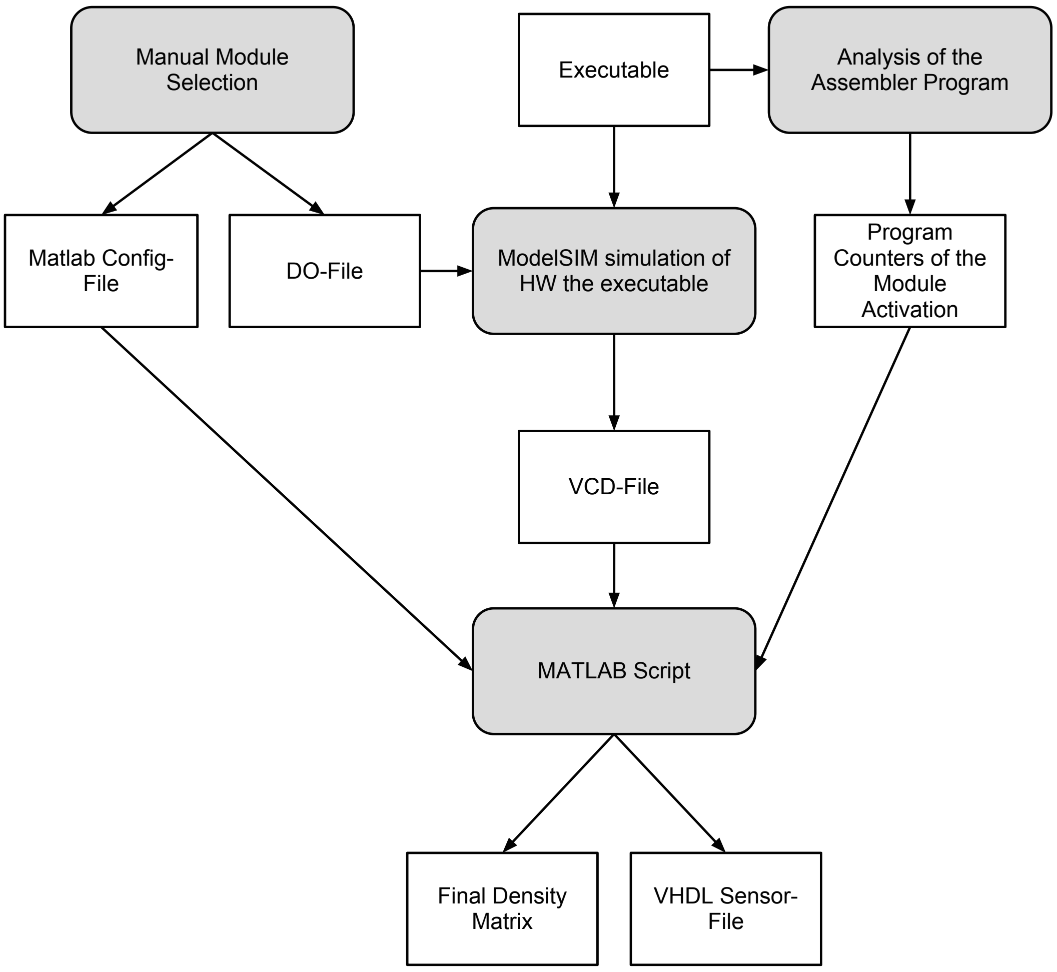


Figure 5.30: Data path of the automatic module analysis technique.

6 Experiments

6.1 The software on the LEON3 processor

The C program that is running on the LEON3 processor has to fulfil several objectives for an accurate experimentation:

1. All observed modules must be activated several times. First, because it shows the function of the activity measurement unit. Secondly, because it offers enough target possibilities to study the outcome of fault injections based on activity analysis.
2. The results of all calculations should be used in the further program flow. Either through a comparison with an equivalent calculation (redundancy) or an check. If an injected fault has an influence on the result of any operation, it raises the probability of a fault propagation.

The following functionality is build into the software:

- `fac(n)` and `fac2(n)`: These are two different functions to calculate the factorial of an input integer. One function forces the operators to load the values from the memory by using the `volatile` keyword. This leads to the activation of the data cache. The other function stores and reads its intermediate results from the register file. In the end, both calculations use multiple times the multiplier.
- `binom(n,k)` and `binom2(n,k)`: These are two different functions to calculate the binomial coefficient. They are using the two existing factorial functions. One binomial function uses the factorial functions with the forced memory accesses, the other one allows the fast usage of the registers. They use heavily the multiplier because of the factorial and the divider to calculate the binomial coefficient.
- `fib(int n)`: A function that calculates the Fibonacci sequence until a given position.
- `calc(x,y)`: This function calls the previous described mathematical operations. Based on the input parameters `x` and `y` the sub-functions are called and the results are compared. A wrong result leads to a different program flow.
- `aes(x)`: The return value of the function `calc` is sent to the AES coprocessor. The result of the encryption is then available to the PowerPC for verification.
- `gpio(x)`: As with the AES coprocessor the result of the `calc` function is also sent to the GPIO pins of the LEON3 processor.

6.2 The emulation observance software on the PowerPC processor

The software, written in C, which is running on the PowerPC is able to observe the emulated hardware (the LEON3 processor) with the implemented GPIO communication interface. The software consists of the follwing three main parts:

- The activity measurement functions
- The fault injection controller functions
- The experiment functions.

In the next sections the details of the main functionality is described. In Section 6.3 the performed experiments and the results are presented.

6.2.1 Activity analysis

The registers of the activity measurement unit can be set over the GPIO interface. This is described in Section 5.1. The function `setActRegister()` is responsible to set a program counter value that defines the start time of the activity observation period and another that defines the end time. Further, by using a mask it can be predefined which modules should be monitored.

The function `copyActFifoToMemory()` downloads the recorded data from the activity measurement unit to the local memory of the PowerPC. The data consists of the information which module is activated or deactivated with the current clock cycle. An example is the excerpt of the following output:

```
clk 95    | modules: 00000000
clk 96    | modules: 00001000
clk 103   | modules: 00001010
clk 110   | modules: 01001010
clk 111   | modules: 01000010
clk 115   | modules: 00000000
```

It shows the activity data as a binary array. The positions of the modules within the array can be found in Figure 5.5 in Section 5.1.4. The second bit represents the multiplier. It is activated at the 103th clock cycle and deactivated at the 115th. At clock cycle 96 the data cache is read. The clock cycle count is the number of clocks that has passed since the integer unit of the LEON3 processor reached the program counter, set previously as start program counter.

Using this data the following functionality can be provided:

- The function `calculateWorkload()` calculates the total activation time of each module. The activation time is defined as the number of cycles a module's sensor notifies the activation. In the first column the module is shown by its identifier. This are the same numbers as defined in Figure 5.5 in Section 5.1.4. The second column (titled "*Activated*") describes the number of activations for each module. The third column contains the total number of active cycles. The idea is that a module that is longer active than others, may provide a bigger target for fault injections. Also different strategic hints can be extracted. Fault Injections on modules that are activated only a few times should concentrate on this activation periods.

```
Workloads achieved:
  Total time: 1552
  | Module       | Activated      | Time active    |
  |--------------|----------------|----------------|
  |    0 (AES)   |       1        |       16       |
  |    1 (MUL)   |       56       |       296      |
  |    2 (DIV)   |       2        |       81       |
  |    3 (DC-R)  |       87       |       181      |
  |    4 (DC-W)  |       60       |       66       |
  |    5 (IC)    |       17       |       134      |
  |    6 (RF)    |       202      |       1028     |
  |    7 (GPIO)  |       0        |       0        |
```

- The module records are stored separately to get start time, end time and duration of each module activation.
- The activity data that is stored locally after a golden run can be compared with the data of a run that was affected by an injected fault, with the function `compareActivityRecords()`.

Further, the function `copyLoggerFifo()` downloads the memory content of the logger. It contains the program counter sequence of the integer unit of the LEON3 processor. This logger sequence is the primary control mechanism to detect influences of fault injections.

6.2.2 Fault injection control

The fault injection controller has to be configured before any faults can be injected. The function `initAttack()` is responsible for the following adjustments:

- The triggers have to configured. The first trigger observes the program counter. If the specified value is reached, the second trigger is activated. This one counts the number of clock cycles until its specified value is reached. The second trigger activates the saboteurs.

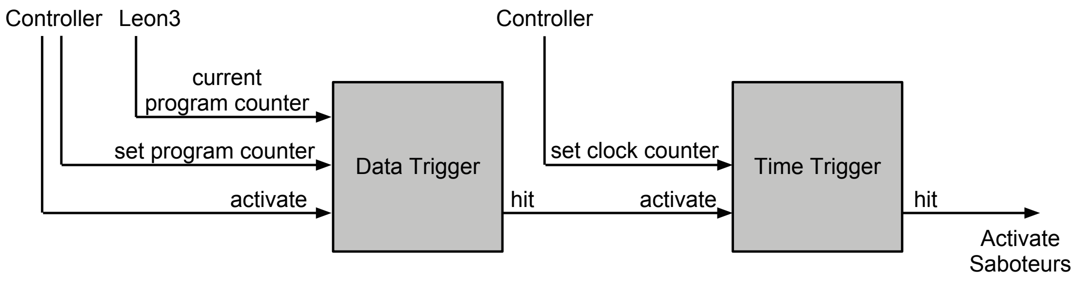


Figure 6.1: Trigger adjustment of the fault injection unit.

- The selection of the specific saboteurs that have to be activated when the triggers are shot. This is done by setting a fault injection pattern. Every bit in the pattern that is set to 1, depicts an activated saboteur.
- The mode of the saboteur is selected. The available modes of the saboteurs are shown in Table 2.1.

- The time duration of the saboteur activation in number of clock cycles.

After setting the configuration of the triggers and the saboteurs, the design can be reset. After the reset the processor executes automatically and the fault is injected at the specified location at the specified timing.

6.2.3 Experiment environment

For the experiment several functions were implemented to guarantee a broad spectrum of experiment possibilities.

First, the function `goldenRun()` executes a golden run, an execution of the SoC until the end without any influence of faults. At the end of the run, the memory content of the activity measurement unit and the logger is downloaded. The logger contains the sequence of program counters and describes the complete program flow of a fault-free run.

Next, the function `doExperiment()` chooses the experiment strategy. The signals that are sabotaged, the time and duration of the fault, the fault injected run and the fault classification is done here. Fault classification is done based on the following results:

1. If the LEON3 does not terminate within a given time window, the result is classified as a timeout, shown as *T/O*.

2. If the LEON3 does terminate, the control flow of the execution is compared with the flow of the golden run. If there is any deviation, the experiment is classified as "differing PC Flow", in the output stated as *PC Flow*.

3. If the sequence of program counter is equal to the golden run sequence, the result of the AES computation is checked. A wrong result is shown as *AES*.

4. If all previous checks are correct, then the sequence of module activations are compared with the golden run. Any irregularities are shown as *Act Diff.*

5. At last the program counter of the LEON3 processor is read out again. If everything went fine, the program counter normally stops at `0x40000808`. A different program counter shows that it may resulted in a trap, which is an exception in the SPARC specification [4]. Results under this classification are shown as *End.*

6. If an unknown and unspecified problem occurred during the execution, the result is shown as *Problem.*

7. If everything went fine and no previous check failed, the result is classified as *OK*.

After the classification the SoC is reset and the next signal and timing is chosen for the next experiment. During the experiments a statistic is created that shows which signals had which influence.

6.3 Fault injections

As described in the concept in 4.6, there are several parameters that have to be chosen for a fault injection. The main parameters are:

- The primary target module and a specific input or output signal of that module.
- The timing of the fault injection
- The duration of the injected fault
- The type of the fault injection

When a design should be tested excessively the fault space is simply too big. So faults must be selected on a random basis. During the experiments the following strategies were applied:

1. An injection of a permanent fault. The fault is injected when the start program counter is reached. The fault lasted until either a timeout was reached or the processor finished its execution.
2. A fault injection at a randomly selected time during the observation period. The fault was active for 2 clock cycles.
3. Based on the activity information, faults were injected during the runtime of a module. This fault was active for 2 clock cycles.
4. During the activity of a module a permanent fault was introduced. But the saboteurs were deactivated after the deactivation of the module.

6.3.1 Saboteurs

Next, Table 6.1 shows the signals that were equipped with saboteurs. The listed signals are all main input and output signals of the chosen modules. Instead of data buses, like the result port of the multiplier, only one descriptive signal was taken. The software that is executed on the LEON3 processor has no if-conditions that are based on bigger or smaller comparisons, but some equal/unequal queries. So a single bit flip meets the objective of forcing a change in the program flow, when the fault propagates to an if-condition.

Multiplier	Divider	Data cache input	Data cache output
muli.start	divi.start	dci.size(0)	dco.set(0)
muli.mac	divi.signed	dci.size(1)	dco.set(1)
muli.signed	divi.flush	dci.enaddr	dco.mexc
muli.flush	holdn (div)	dci.eenaddr	dco.hold
holdnx (mul)	divo.nready	dci.nullifiy	dco.mds
mulo.nready	divo.ready	dci.lock	dco.werr
mulo.ready	divo.icc(0)	dci.read	dco.cache
mulo.icc(0)	divo.icc(1)	dci.write	dco.idle
mulo.icc(1)	divo.icc(2)	dci.flush	dco.scanen
mulo.icc(2)	divo.icc(3)	dci.flushl	dco.testen
mulo.icc(3)		dci.dsuen	
mulo.result(0)		dci.msu	
		dci.esu	
		dci.intack	

Icache input	Icache output	Register file
ici.rbranch	ico.set(0)	rfi.wren
ici.fbranch	ico.set(1)	rfi.waddr(0)
ici.inull	ico.mexc	rfi.wdata(0)
ici.su	ico.hold	rfi.ren1
ici.flush	ico.flush	rfi.raddr1(0)
ici.flushl	ico.diagrdy	rfo.data1(0)
ici.pnull	ico.mds	rfi.ren2
	ico.idle	rfo.data2(0)
		rfi.raddr2(0)
		rfi.diag(0)

Table 6.1: The signals that were equipped with saboteurs.

6.3.2 Fault injection with permanent faults

As a first analysis only permanent faults were introduced into the processor. The saboteurs were activated when the program counter of the integer unit of the LEON3 processor reached the start program counter. The saboteurs stayed active until the LEON3 processor terminated its program execution. The following saboteur modes were tested:

- Stuck-at-zero: The signal line is kept at `0`.
- Stuck-at-one: The signal line is kept at `1`

The fault injection on each signal shows if there exists an observable outcome. As an example, a fault on a few signals may only influence the runtime of the processor. The data cache signal `dci.flush` instructs the data cache to invalidate the currently stored data. So the next reading procedure is directed to the main memory and the loading sequence will only take longer. But there is no big influence in the control path of the program. The outcomes of the permanent faults are shown in Table 6.2.

Nr.	Signal	Stuck-at-1	Stuck-at-0
0	muli.start	PC Flow	T/O
1	muli.mac	OK	OK
2	muli.signed	OK	OK
3	muli.flush	T/O	OK
4	holdnx (mul)	PC Flow	T/O
5	mulo.nready	T/O	T/O
6	mulo.ready	End PC	OK
7	mulo.icc(0)	OK	OK
8	mulo.icc(1)	OK	OK
9	mulo.icc(2)	OK	OK
10	mulo.icc(3)	OK	OK
11	mulo.result(0)	PC Flow	PC Flow
12	divi.start	PC Flow	T/O
13	divi.signed	OK	OK
14	divi.flush	T/O	OK
15	holdn (div)	PC Flow	T/O
16	divo.nready	T/O	T/O
17	divo.ready	PC Flow	T/O
18	divo.icc(0)	OK	OK
19	divo.icc(1)	OK	OK
20	divo.icc(2)	T/O	OK
21	divo.icc(3)	PC Flow	T/O
22	dci.size(0)	T/O	T/O
23	dci.size(1)	End PC	PC Flow
24	dci.enaddr	T/O	T/O
25	dci.eenaddr	Act Diff	OK
26	dci.nullifiy	T/O	OK
27	dci.lock	OK	OK
28	dci.read	T/O	T/O
29	dci.write	T/O	T/O
30	dci.flush	OK	OK
31	dci.flushl	T/O	OK
32	dci.dsuen	T/O	T/O
33	dci.msu	PC Flow	T/O
34	dci.esu	OK	OK
35	dci.intack	OK	OK
36	dco.set(0)	T/O	T/O
37	dco.set(1)	PC Flow	T/O
38	dco.mexc	T/O	T/O
39	dco.hold	End PC	OK
40	dco.mds	T/O	OK
41	dco.werr	T/O	OK
42	dco.cache	End PC	OK
43	dco.idle	OK	OK
44	dco.scanen	T/O	PC Flow
45	dco.testen	PC Flow	T/O
46	ici.rbranch	OK	PC Flow
47	ici.fbranch	T/O	PC Flow
48	ici.inull	T/O	T/O
49	ici.su	PC Flow	T/O
50	ici.flush	OK	OK
51	ici.flushl	OK	OK
52	ici.pnull	T/O	OK
53	ico.set(0)	T/O	T/O
54	ico.set(1)	T/O	T/O
55	ico.mexc	T/O	End PC
56	ico.hold	End PC	OK
57	ico.flush	OK	OK
58	ico.diagrdy	OK	OK
59	ico.mds	T/O	T/O
60	ico.idle	PC Flow	PC Flow
61	rfi.wren	PC Flow	T/O
62	rfi.waddr(0)	T/O	T/O
63	rfi.wdata(0)	T/O	T/O
64	rfi.ren1	PC Flow	T/O
65	rfi.raddr1(0)	T/O	T/O
66	rfo.data1(0)	T/O	T/O
67	rfi.ren2	End PC	T/O
68	rfo.data2(0)	T/O	PC Flow
69	rfi.raddr2(0)	PC Flow	T/O
70	rfi.diag(0)	T/O	T/O

Table 6.2: The outcomes of permanent faults on each signal.

Since each saboteur is activated at the beginning of the observation period, the manipulation will influence the execution as soon as possible. But it can be seen in Table 6.2 that there are several signals that did not have any influence when being manipulated. For example, the signals `muli.mac` or `mulo.icc` did result in *OK* in both cases. This means that the signal value has no influence within the observation period when it stays constant. However, it may be possible that a signal change is required to show influence.

6.3.3 Transient faults (randomly and activity based)

In the next step transient faults were injected. The time duration of a single fault is set to 2 clock cycles. After the two cycles the saboteur is deactivated and the signal behaves normal as usual. The used modes were again stuck-at-zero and stuck-at-one.

Two different experiments were started:

- **Fault injections at random timings:** During the whole observation period the timing for a fault injection was always selected on a random basis.
- **Fault injections at a timing where a module is active:** A saboteur on an active module is activated during the activity period.

6.3.3.1 Randomly timed fault injections

The saboteur on every signal was activated 50 times at different timings. The timing for the injection of the fault was selected on a random basis, but it was within the observation period. The fault lasted for 2 clock cycles and was then removed. At every activation the LEON3 processor was in a different state. Therefore, it can be expected that the results must somehow differ. Figure 6.2 shows the results for stuck-at-zero faults. There are many experiments that resulted in *OK*. This are the blue bars which are the majority in the graph. The *OK* classification means that no influence of the injected fault was detected. There are only few signals that show a dependence on the faults.

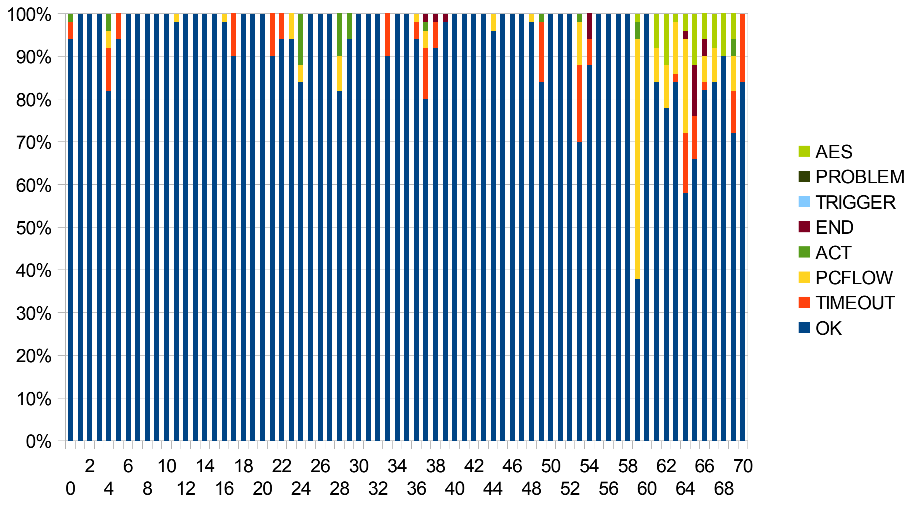


Figure 6.2: Statistic of the fault classifications for randomly injected stuck-at-zero faults.

However, it can be seen instantly that signal 59 (`ico.mds`) is very sensitive to stuck-at-zero faults. Only 38% of the fault injections were not detected. The major part resulted in a difference of the program flow. This corresponds to its functionality that is described in Section 5.2.4. The signal `ico.mds` announces that the output of the instruction cache was successfully loaded from the memory and is now provided at the output ports. This signal is low active. A wrong instruction is probably provided and this seem to have some dependence on the program flow. In this experiment 93% of the injected faults were not activated.

Next in Figure 6.3 shows the same graph for stuck-at-one faults.

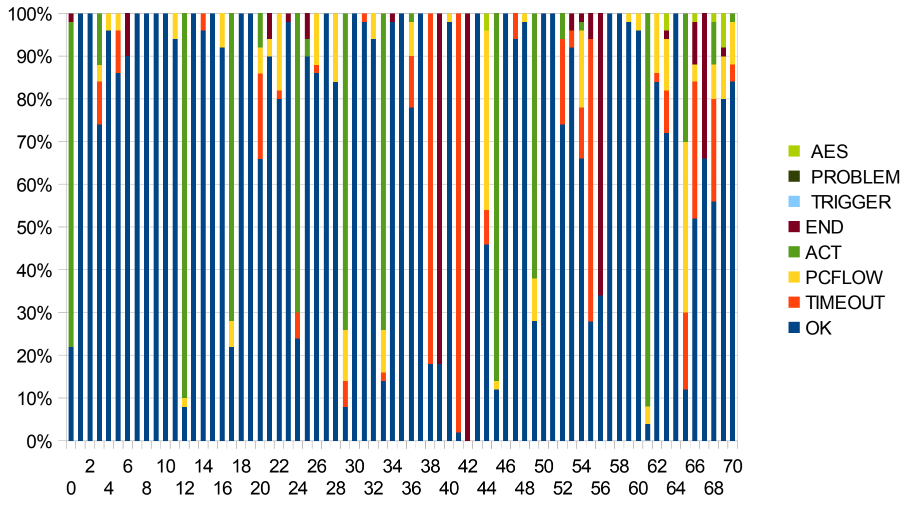


Figure 6.3: Statistic of the fault classifications for randomly injected stuck-at-one faults.

It can be seen immediately that faults which put a signal to high, have a much higher influence to the system. There are many faults that were only classified as a difference in the activity measurements. This is mainly because the injection forced a module to start its activity, which the sensors recognized. Since a difference in the control flow of the executed software or a difference in the AES result is higher prioritised, a difference in the activity statistics has not a huge impact. However, fault injections at signal 38 (`dci.mexc`) and signal 41 (`dco.werr`) mainly result in timeouts.

6.3.3.2 Activity based timing for fault injections

Next the same experiment was done based on the activity measurement data. Instead of randomly selected timings, the saboteurs were activated during the activity of its associated module. The selected activity period and the time within this activity period was selected randomly. Every saboteur was again activated 50 times to achieve a fair comparison with the former method.

Figure 6.4 pictures the statistic for stuck-at-zero faults. Compared to the former method it can be seen that there is a higher impact on timeouts and on changes in the program counter sequence (control flow).

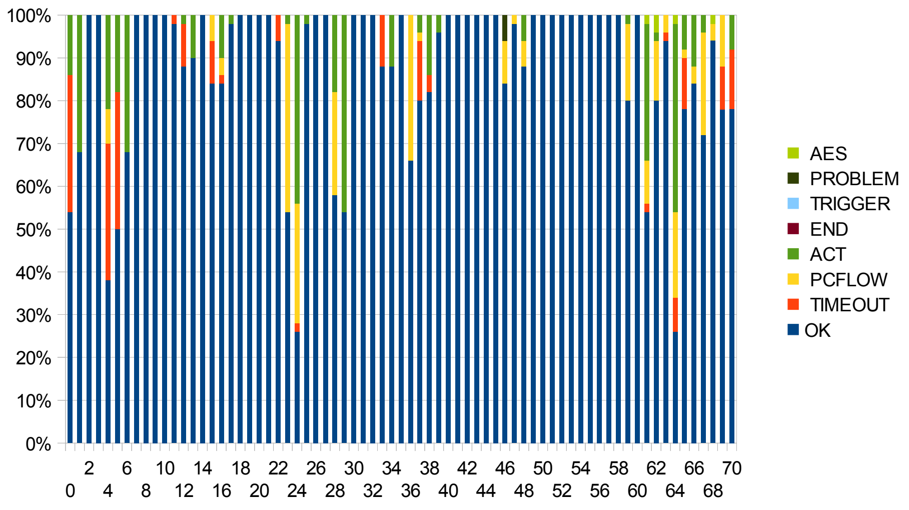


Figure 6.4: Statistic of the fault classifications for stuck-at-zero fault injections based on activity timing.

Figure 6.5 pictures the same experiment for stuck-at-one faults. The same injections as previously generated timeouts, but some new signals show the same tendency during the activation period. These are signal 3 (`muli.flush`) and signal 14 (`divi.flush`).

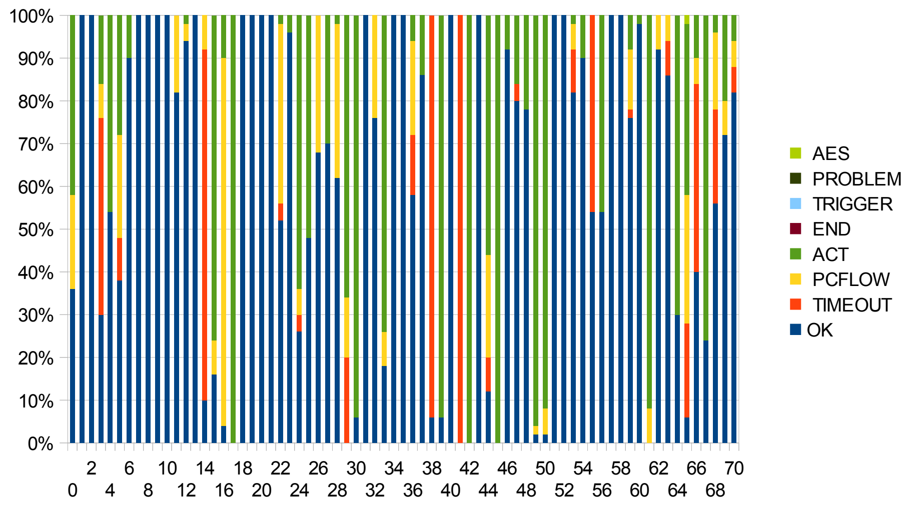


Figure 6.5: Statistic of the fault classifications for stuck-at-one fault injections based on activity timing.

Signal 16 (`divo.nready`) has here a huge dendency to provoke a control flow change. Simply because the integer unit is waiting for a result and an invalid result is presented with this fault.

6.3.4 Results

Table 6.3 shows the percentage of each classification of all injected faults.

Experiment	Saboteur Mode	OK	Timout	PC flow	Act	End	AES
Random	Stuck-at-zero	93%	2,4%	2,51%	0,67%	0,42%	1%
Random	Stuck-at-one	73,24%	6,5%	4,36%	10,87%	4,73%	0,28%
Activity	Stuck-at-zero	87,21%	2,9%	3,94%	5,7%	0,08%	0,14%
Activity	Stuck-at-one	61,13%	7,69%	7,1%	24%	0%	0,02%

Table 6.3: Results of the experiments with transient fault injections.

If the results of the experiments using transient fault injections are compared, the following observations and conclusions can be made:

1. The number of undetected faults is lower when faults are injected during the activity of modules. Even though the number of undetected faults has still the majority, as Table 6.3 shows.

2. The number of timeouts and those faults that force a different control path of the executed software did raise, but only slightly.

3. The impact to module activations and deactivation is very huge. These are the faults that are classified as *Act*. They have produced a different activity profile.

4. There are signals, on which a saboteur does not seem to result in a fault. They are the same as those that did not produce a fault in the experiment with a permanent activation of the saboteurs.

5. Saboteurs on specific signals tend to produce a timeout in the LEON3 processor. The signals `muli.flush`, `divi.flush`, `dco.mexc` and `dco.werr` have a huge sensibility. They have produced the majority of timeouts. Therefore, such signals must either be hardened by redundancy or a less sensible functionality.

6. The same observation in the case of control flow changes was made with the signals `divo.nready, dci.size(0), dci.size(1), dci.read, ico.mds`. The first three observations was made during the activity phase.

7. It is not sufficient only to test fault injections during the activity. Instead a random timed fault injection with a higher probability on activity times should be used, e.g. by using a Gaussian distribution.

6.3.5 Further observations

Some further observations could be made, because a deeper knowledge about the hardware description of the LEON3 processor was already available at the experimentation phase. Without this knowledge these observations would not have been possible so easily:

1. Fault Injections that suppress or manipulate the output signals `divo.ready` or `mulo.ready` are without any consequences. They usually indicate that the result is now provided at the output port `divo.result` and `mulo.result`. However, the integer unit of the LEON3 in the version 1.1.0-b4108 that was used for the experiments uses `divo.nready`

and `mulo.nready`. Both ports are marked as unused in the GRLIB IP manual [14] and are set and unset a few cycles sooner before the result is even provided.

2. The signal `dco.mds` usually indicates when a value that is loaded from the main memory is provided by the data cache. Unlike expected it does not show a big sensitivity to transient faults. Mostly because the integer unit was not respecting the output signal at this timing. It can be considered as stable to faults.

7 Conclusion

7.1 Summary

In this master's thesis a method was developed to accelerate fault injection campaigns in emulated environments.

Fault injection is an approach to test an hardware design under the occurence of faults. It can help the design developer to detect flaws and weaknesses in the design. The emulation of hardware on FPGAs allows a fast verification of the design. However, a complex system-on-chip design provides a too big spectrum of fault injection targets. In this thesis a new approach was taken to gain pre-injection knowledge. Related works in the field of fault injection optimization showed that pre-injection knowledge helps to avoid fault injections that are never activated. While there is some research work available in the field of hardware simulations, there is not much in the field of hardware emulation.

The activity of different components or modules of a system-on-chip design was supervised and recorded. These modules were extended with sensors. They observe if the module is active or not. The sensors can do this by listening to input and output signals and also to selected register values. Therefore, the sensors had to be individually customized to those modules. A central measurement unit records the activity data and provides a general configuration interface (GPIO) to any controllers.

The selected system-under-test for the case study is a LEON3 processor from Gaisler Research. This is a processor with a big spectrum of fault injection targets.

For the experimentation the digital design was further extended with a modular fault injector. Several signals of the observed modules were equipped with saboteurs. Upon activation, the saboteurs influenced their associated signal. The experiments were taken in several ways. First, the injection of permanent faults allowed the observation of the dependency of specific signals. It could be seen, that there are several signals that do not activate a fault or error when influenced. A next experiment was undertaken with random timed fault injections. It could be seen that there are signals that tend to trigger a timeout when sabotaged. Further some faults may influence the result of an AES computation without changing the program flow. Another experiment was undertaken with the knowledge of the activity measurement. The number of unactivated faults could be lowered. It could be seen that the same error has a completely different outcome when injected in a module during its operation.

It could be seen that the fault injection during activity helps to gain further knowledge about the fault dependency of a hardware design. But it does not suffice. In this thesis the same amount of faults were injected at random times as during the small periods of activity. That raised the number of fault injections during activity and kept faults during non-activity. Using this setting an overall coverage can still not be guaranteed but approximated very well.

7.2 Future work

7.2.1 Further experiments

With the given system far more different experiment can be conducted, to help the test developer detect flaws and weaknesses in the design. A few ideas would be:

- The injection of faults at or just before the activity start.
- The injection of permanent faults during the activity.
- The injection of faults at the end of the activity.

Also faults that put a signal on an unknown state ('X') or an uninitialized state ('U') were not undertaken during this thesis. It would have been possible with the used fault injection framework, though. The uninitialized state is a default value and the unknown state could emulate bus contentions or error conditions [1]. The fault injection framework does support all of the states, though.

7.2.2 Automatic detection of relevant signals for the sensors

The MATLAB script can create more precise state samples, by using the time information provided by the VCD file. This means, the samples would consist of time-continuous signal information. The merging problem of the density information matrices could be avoided. Nevertheless, to gain the information about the important signals, new methods must be tested.

7.2.3 Automatic insertion of sensors

The sensors that were created by hand with the help of the MATLAB script could be inserted automatically into the design. The lack of current scripts or programs to modify existing VHDL code efficiently is a primary reason for further study.

8 Appendix

8.1 Compilation of the software running on the LEON3 processor

The compilation of the C-source file to an executable is done with

```
$ sparc-elf-gcc -mcpu=v8 -msoft-float -O2 do_all.c -o do_all
```

For the simulation with ModelSim the `srec`-files have to be created. With the following command the executable can be copied to sram record files. They represent the executable within the RAM and are executed by the simulator.

```
$ sparc-elf-objcopy -O srec do_all sram.srec
$ sparc-elf-objcopy -O srec do_all sdram.srec
```

The `srec`-files have to be copied to the design directory `leon3-xilinx-ml507-without-ppc/` of the LEON3. This design does not include the PowerPC of the development board and can therefore be simulated.

```
$ cp sram.srec sdram.srec $(GRLIB)/designs/leon3-xilinx-ml507-without-ppc/
```

For synthetisation the software was stored within the PROM. This has to be done because the PowerPC occupied the SDRAM of the development board. The LEON3 processor used the DDR-RAM. The PROM can be programmed with the following commands

```
$ mkprom2 -v -mv8 -msoft-float -nocomp -baud 38400 -freq 80 -stack 0x4ffffff0 \
    -romws 12 -romsize 4096 -romwidth 16 -ddrram 256 -ddrfreq 190 -ddrcol 1024 \
    -nosram do_all
$ rm -f   $(GRLIB)/designs/leon3-xilinx-ml507
$ make -C $(GRLIB)/designs/leon3-xilinx-ml507 ahbrom.vhd FILE=‘pwd‘/prom.out
```

An object dump that contains the program in assembler language can be created with the following statement

```
$ sparc-elf-objdump -S -d do_all &> do_all.diss
```

8.2 Synthetization of the LEON3 with the Activity Unit and Fault Injection Controller for the FPGA

For the synthetisation the HDL sources in the design directory `leon3-xilinx-ml507` have to be set to its current path by entering

```
$ make scripts
```

This command generates the ISE project file `leon3mp.xise`. Next the program ISE can be launched by typing

```
$ ise leon3mp.xise
```

The sources of all used Xilinx IP Cores have to be added manually within the ISE design environment. This is done by clicking on *Project → Add Source.* The following IP-Cores have to be added:

- `/xps/ppc_sram_system/system.xmp` for the IP-Core that communicates with the PowerPC processor.
- `/act_fifo_ipcore/fifo_generator_v8_4.xco`, which is the design for the FIFO that is used to store the activity information during the golden run.
- `/ipcore_dir/fifo_generator_v8_3.xco`, which is the FIFO, used to store the program flow of the fault injection controller.

Finally, the synthetization can be started by selecting *leon3mp - rtl (leon3mp.vhd)* in the design tab (design hierarchy) of the ISE Project Navigator. Then in the process tab the entry *Generate Program File* must be double clicked to start the processes. In the end a bit file `leon3mp.bit` is generated. This bit file will be loaded into the FPGA.

8.3 Execution of the Activity Analysis with the PowerPC

First the FPGA needs to be programmed with the synthesized bit file. The program capable of doing this is started with the command

```
$ impact
```

The last device in the shown chain is the FPGA with the name `xc5vfx70t`. With a double click on this device the previously generated bit file `leon3mp.bit` can be assigned. And with a right click on the device the FPGA can be programmed. Finallys the program has to be disconnected by selecting *Output → Cable Disconnect.*

Next, the software that runs on the PowerPC of the FPGA development board needs to be compiled with the Xilinx Software Development Kit. It can be started with the command

```
$ xsdk
```

To create a new workspace and to compile the sources following steps must be done:

1. The selection of *File → Switch Workspace* allows the setting of the root directory of the PowerPC sources (e.g. `PM_ActivityAnalysis_Leon3/ppc_sw/`). After that the XSDK restarts.
2. The sources are then loaded by clicking *File → Import.* The selection has to be set to *Existing Projects into Workspace* and the same folder as previously has to be selected.

Next, the sources can be compiled by selecting the file `act/src/main.c` in the Project Explorer and by clicking on *Project → Build all.* A file named `Debug/act.elf` will be created.

In the shell the working directory has to be changed to the directory `ppc_sw/act/Debug`. In there, the Xilinx Microprocessor Debugger can be started and the software for for the PowerPC can be loaded.

```
$ xmd
XMD% connect ppc hw
XMD% dow act.elf
XMD% run
```

Now, the software is executed on the PowerPC, that resets the LEON3 processor. The LEON3 processor executes then the software that is stored in its PROM and the activity information is stored in the FIFO.

To see the output that is generated by the software a second shell must be opened. The following command opens a program that receives the output and is able to send a console input.

```
$ minicom
```

8.4 Simulation with ModelSim for the sampling

Within the design directory `leon3-xilinx-ml507-without-ppc` the ModelSim simulator can be started with

```
$ make scripts
$ make vsim
$ vsim testbench
```

The simulator starts. If only the signals shall be observed and analyzed manually the following commands last to record all signal changes.

```
> log /* -r
```

If selected signal changes shall be stored within a VCD-file, the signals have to be selected manually. The do-file that is chosen next contains all signals that are necessary for the sampling process for the multiplier module. A descriptive example of the content is shown in Section 5.4.1 in Listing 5.1. The `do`-command instruct the simulator to load the file.

```
do ../../../../VCD-measurements/vcd_smul/mul.do
```

Next the simulation can be started by typing

```
> run -all
```

If the do-file was loaded, then a VCD file is generated that contains all selected signal changes within the simulation. The usage of the VCD file within the MATLAB script is presented in Section 5.4.2. The MATLAB script generates the density matrices used to understand the functionality of the supervised module. Further it generates a sensor in VHDL.

8.5 Configurations

8.5.1 HDL Modules on the AMBA Bus

As described in the concept in Section 4 the LEON3 processor was extended with several new modules to extend the functionality. The most important extension is the activity measurement unit. Since the configuration of this module is also possible over the software that is executed on the LEON3 processor an AMBA interface was implemented. The debug booting message of the processor shows the I/O address of this module. Over this address the registers can be written.

```
# apbctrl: slv10: TUGRAZ                    ACTIVITY SENSORS HW APB/AMBA
# apbctrl:        I/O ports at 0x80000a00, size 256 byte
```

The following code sequence shows how the registers can be written using the I/O address. The program that runs on the LEON3 processor must include the following struct definition and can then write on the registers as shown exemplarily in the `main`-function.

Listing 8.1: Configuration of the activity measurement unit out of the LEON3 software

```
struct act_regs_t {
  volatile int ctrl;
  volatile int start;
  volatile int end;
};

//struct act_regs_t *act_regs = (struct act_regs_t *) 0x80000F00; // XC3s
struct act_regs_t *act_regs = (struct act_regs_t *) 0x80000A00; // ML507

int main() {
  act_regs->ctrl  = 0x000000FF; //All modules, end is program counter
  act_regs->start = 0x400011a4; //Set start program coutner
  act_regs->end   = 0x400011d0; //Set end program counter

  //Do stuff
}
```

Also an AES coprocessor was included into the design. The booting message contains the following description.

```
# apbctrl: slv13: TUGRAZ                    AES CORE OPEN CORES APB/AMBA
# apbctrl:        I/O ports at 0x80000d00, size 256 byte
```

The usage of the AES coprocessor is similar to the activity analysis module.

Listing 8.2: Example of using the AES coprocessor

```
struct aes_regs_t {
  volatile int ctrl;  // 0000
  volatile int res0;  // 0001
  volatile int res1;  // 0010
  volatile int res2;  // 0011
  volatile int dout0; // 0100
  volatile int dout1; // 0101
  volatile int dout2; // 0110
  volatile int dout3; // 0111
  volatile int din0;  // 1000
  volatile int din1;  // 1001
  volatile int din2;  // 1010
  volatile int din3;  // 1011
  volatile int key0;  // 1100
  volatile int key1;  // 1101
  volatile int key2;  // 1110
  volatile int key3;  // 1111
};
struct aes_regs_t* aes_reg = (struct aes_regs_t*) 0x80000D00;

void aes() {
  // aes_reg->din0 to din3 and key0 to key3 must be set already

  //Mode (Bit0 = 0 for encryption, 1 for decryption) and Startbit (Bit1)
  aes_reg->ctrl = 0x3;
  while(aes_reg->ctrl != 0x1); //Wait for result (signal done_s)

  //result is available on aes_reg->dout0 to dout3
}
```

Nomenclature

AES	Advanced Encryption Standard
AHB	AMBA High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application Specific Integrated Circuiut
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DDR-SDRAM	Double Data Rate SDRAM
ESA	European Space Agency
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GL	Gate Level
GPIO	General Purpose Input Output
GRLIB	Gaisler Research Library
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuit
IP	Intellectual Property
IU	Integer Unit
JTAG	Joint Test Action Group (= IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture)
LSB	Least Significant Bit
MEU	Multiple Event Upset
MFI	Modular Fault Injector
MMU	Memory Management Unit
MSB	Most Significant Bit
PC	Program Counter
PROM	Programmable Read-Only Memory
RAM	Random-Access Memory
RF	Radio Frequency
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random-Access Memory
SEL	Single Event Latch-up
SEU	Single Event Upset
SL	System Level
SoC	System on Chip
SUT	System Under Test
T/O	Time-out

UART	Universal Asynchronous Receiver/Transmitter
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Bibliography

[1] IEEE standard VHDL language reference manual - redline. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) - Redline*, 26.

[2] *Aeroflex Gaisler AB*. `http://www.gaisler.com`, July 2012.

[3] Aeroflex Gaisler, `http://www.gaisler.com/doc/grmon.pdf`. *GRMON - LEON3 Debug Monitor*, version 1.1.53 edition, July 2012.

[4] Aeroflex Gaisler, `http://www.gaisler.com/doc/Leon3%20Grlib%20folder.pdf`. *The LEON3 processor*, July 2012.

[5] Aeroflex Gaisler AB, `http://www.gaisler.com/doc/libio/bcc.html`. *BCC user's manual*, v1.0.12 edition, July 2012.

[6] Jason R. Andrews. *Co-verification of hardware and software for ARM SoC design*. Elsevier, 2005.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. Fundamental concepts of dependability. *Technical Report Series-University Of Newcastle Upon Tyne Computing Science*, 2001.

[8] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, February 2006.

[9] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Assembly-level preinjection analysis for improving fault injection efficiency. In *Proceedings of the Fifth European Dependable Computing Conference (EDCC-5)*, 2005.

[10] A. Bhattacharjee, G. Contreras, and M. Martonosi. Full-system chip multiprocessor power evaluations using FPGA-based emulation. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2008*, pages 335–340, August 2008.

[11] J.V. Carreira, D. Costa, and J.G. Silva. Fault injection spot-checks computer system dependability. *Spectrum, IEEE*, 36(8):50–55, August 1999.

[12] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2001.*, pages 250–258, 2001.

[13] J. Coppens, D. Al-Khalili, and C. Rozon. VHDL modelling and analysis of fault secure systems. In *Proceedings of Design, Automation and Test in Europe, 1998*, pages 148–152, February 1998.

[14] Aeroflex Gaisler. *GRLIB IP Core User's Manual*, 1.1.0 - b4112 edition, January 2012.

[15] Jiri Gaisler. A structured vhdl design method. In *Fault-tolerant Microprocessors for Space Applications*. Gaisler Research, 2004.

[16] A. Genser, C. Bachmann, J. Haid, C. Steger, and R. Weiss. An emulation-based real-time power profiling unit for embedded software. In *International Symposium on Systems, Architectures, Modeling, and Simulation, 2009*, pages 67–73, July 2009.

[17] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid. Automatic saboteur placement for emulation-based multi-bit fault injection. In *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, June 2011.

[18] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid. Modular fault injector for multiple fault dependability and security evaluations. In *14th Euromicro Conference on Digital System Design (DSD)*, pages 550–557, September 2011.

[19] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid. Efficient fault emulation based on post-injection fault effect analysis (PIFEA). In *IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS), 2012*, pages 526–529, August 2012.

[20] L. Kafka. Analysis of applicability of partial runtime reconfiguration in fault emulator in xilinx FPGAs. In *11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2008*, pages 1–4, April 2008.

[21] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, Günther Leber, and Johannes Reisinger. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. *Dependable computing and fault tolerant systems*, 10:267–288, 1998.

[22] R.V. Kshirsagar and R.M. Patrikar. A novel fault tolerant design and an algorithm for tolerating faults in digital circuits. In *3rd International Design and Test Workshop, 2008. IDT 2008.*, pages 148–153, December 2008.

[23] Seok-Lyong Lee, Seok-Ju Chun, Deok-Hwan Kim, Ju-Hong Lee, and Chin-Wan Chung. Similarity search for multidimensional data sequences. In *16th International Conference on Data Engineering, Proceedings*, pages 599–608, March 2000.

[24] Clive Maxfield. *The Design Warrior's Guide to FPGAs*. Academic Press, Inc., Orlando, FL, USA, 2004.

[25] Mentor Graphics, `http://model.com`. *ModelSim - Advanced Simulation and Debugging*, July 2012.

[26] G. Munkby and S. Schupp. Improving fault injection of soft errors using program dependencies. In *Testing: Academic Industrial Conference - Practice and Research Techniques, 2008*, pages 77–81, August 2008.

[27] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proceedings. 17th IEEE VLSI Test Symposium, 1999.*, pages 86–94, 1999.

[28] OpenCores. `http://opencores.org`, November 2012.

[29] A. Pellegrini, V. Bertacco, and T. Austin. Fault-based attack of RSA authentication. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 855–860, March 2010.

[30] A. Rohani and H.G. Kerkhoff. A technique for accelerating injection of transient faults in complex SoCs. In *14th Euromicro Conference on Digital System Design (DSD), 2011*, pages 213–220, September 2011.

[31] T.K. Tsai, Mei-Chen Hsueh, Hong Zhao, Z. Kalbarczyk, and R.K. Iyer. Stress-based and path-based fault injection. *IEEE Transactions on Computers*, 48(11):1183–1201, November 1999.

[32] Xilinx, `http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf`. *ML 507 Evaluation Plattform*, v3.1.2, may 16, 2011 edition. checked Nov. 2011.

[33] Xilinx, `http://www.xilinx.com`. *ISE Design Suite*, July 2012.

[34] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. A survey on fault injection techniques. In *The International Arab Journal of Information Technology*, volume 1, pages 171–186, July 2004.