Techniques for Increasing Security and Reliability of IP Cores Embedded in FPGA and ASIC Designs

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Daniel Michael Ziener

Erlangen 2010

Als Dissertation genehmigt von der Technischen
Fakultät der Universität Erlangen-Nürnberg

Tag der Einreichung: ..... 02. Juni 2010
Tag der Promotion: ..... 27. Juli 2010
Dekan: ..... Prof. Dr.-Ing. Reinhard German
Berichterstatter: ..... Prof. Dr.-Ing. Jürgen Teich
..... Prof. Dr. sc.techn. Andreas Herkersdorf

Acknowledgments

I would like to express my sincere gratitude to my advisor, Professor Jürgen Teich, for his guidance, and encouragement throughout this work. His scientific, technical, and editorial advice was essential for my work as an academic researcher. I would also thank Professor Andreas Herkersdorf for the fruitful cooperation with him and his chair and for agreeing to be the co-examiner of this work. My thanks also go to all my colleagues for the discussions of my research work, especially Marcus Bednara, for the numerous scientific and editorial advises, and Moritz Schmid for his critical review of this thesis and his fruitful feedback and discussion. Finally, I would like to thank my family and friends for their support and encouragement during the past years.

Daniel Ziener
Erlangen, May 2010

Contents

1 Introduction

The focus of this work are faults and attacks in embedded systems, as well as methods to cope with their associated overhead. This chapter gives a motivation for the topic of this thesis. Also, terms and definitions in the field of security and reliability are given. Finally, the major contribution of this work are summarized.

1.1 Motivation

Since the invention of the transistor, the complexity of integrated circuits continues to grow rapidly. First, only basic functions like discrete logic gates were implemented as integrated circuits. With improvements in chip manufacturing, the size of the transistors was drastically reduced and the maximum size of a die was increased. Now, it is possible to integrate more then one billion transistors [Xil03] on one chip.

In the beginning, electric circuits (e.g., a central processing unit) consisted of discrete electronic devices which were integrated on *printed circuit boards* (PCBs) and consumed a lot of power. The invention of integrated circuits in the end of the 1950s laid also the cornerstone of the development of embedded systems. For the first time, the circuits were small enough and consumed less power, so that applications embedded into a device, like production machines or consumer products became possible. An embedded system is considered as a complete special purpose computer that may consist of one or more CPUs, memories, a bus structure and special purpose cores.

The first integrated circuits were able to integrate basic logic functions (e.g., AND-, OR-gate) and flip-flops. With further integration, complex circuits, like processors, could be implemented into one chip. Today, it is possible to integrate a whole system

with processors, buses, memories and specific hardware cores on a single chip, a so called *system-on-chip* (SoC).

These small, power and cost efficient, but manifolded applicable embedded systems finally took off on their triumphal course. Today, embedded systems are included in most electrical devices, from the coffee machine over stereo systems to washing machines. The application field of embedded systems spans from consumer products, like mobile phones or television sets, over safety critical applications, like automotive or nuclear plant applications, to security applications, such as smart cards or identity cards.

As integration density grew, problems with heat dissipation arose. The embedding of electronics into systems with small place and reduced cooling possibility, or the operation in areas with extreme temperature, intensify this problem. Furthermore, an embedded system which is integrated into an environment with moving parts is exposed to shock. Thermic and shock problems have a high influence on the reliability of the system. On the other hand, a system that steers big machines or controls a dangerous process must have a high operational reliability. These are all reasons that design for reliability is gaining more and more influence on the development of embedded systems.

However, what is the need for reliability, if everyone may alter critical parameters or shut down important functions? To solve these problems, we need access control to the embedded system. But, today, embedded systems are also used to grant access to other systems or buildings. One example are chip cards. Inside these cards, a secret key is stored. It is important that no unauthorized persons or systems are able to read this secret key. Thus, an embedded system should not only be reliable but also secure.

Integration of functions for the guarantee of reliability and security features increases also the complexity of the integrated system enormously and thus design time. On the other hand, the market requires shorter product cycles. The only solution is to reuse cores, which have been designed for other projects or were purchased from other companies. The number of reused cores constantly increases. The advantages of IP core (Intellectual Property cores) reuse are substantial. For example, they offer a modular concept and fast development cycles.

IP cores are licensed and distributed like software. One problem of the IP cores distribution, however, is the lack of protection against unlicensed usage, as cores can be easily copied. Future embedded systems should also be able to prevent the usage of unlicensed cores or the core developers should be able to detect their cores inside an embedded system from third party manufactures.

Considering todays embedded systems, the integration of reliability and security increasing functions depends on the application field. In the area of security-critical systems (e.g., chip cards, access systems, etc.), several security functions are implemented. We find additional reliability functions in systems where human life or valuable assets are at stake (e.g., power plants, banking mainframes, airplanes, etc.).

On the other hand, the problem of all these additional functions is the requirement for additional chip area. For cost-sensitive products which are produced in huge volumes, like mobile phones or chip cards, the developer must rethink to integrate such additional functions.

Today, CMOS technologies for integrated circuits have reached the deep-submicron area. CMOS designs manufactured in deep-submicron technologies are very sensitive against ionized radiation (which may cause soft errors), operating point variation by means of temperature or supply voltage fluctuations, as well as parasitic effects, which results in statical leakage currents [ITR07] [Mic03].

Future circuits manufactured in deep-submicron technology can be integrated with a much higher complexity and more cores than with today's technologies. To achieve a short time to market of future products, the usage of IP cores become more and more important. This will boost the trade with IP cores, which also arises the question of their security against unlicensed usage. Also, the percentage of costs for area overhead for additional security and reliability functions will decrease with increasing chip area. These facts show that reliability and security of IP cores will become more and more important for future system development. They have motivated this thesis entitled: ***"Techniques for Increasing Security and Reliability of IP Cores Embedded in FPGA and ASIC Designs"***

Why Security?

Security becomes more and more important for computers and embedded systems. With the ongoing integration of personal computers and embedded systems into networks and finally into the Internet, security attacks on these systems arose. These networked, distributed devices may now compute sensitive data from the whole world and the attacker does not need to be physically present. Also, the increased complexity of these devices increases the probability of errors which can be used to break into a system. Figure 1.1 shows a classification of different types of attacks related to computer systems. This information is obtained form the CSI Computer Crime and Security Survey [Ric08], where 522 US-companies reported their experience with computer crime. Further, the integration of networking interfaces into embedded devices, for which it would not be obviously necessary lead to strange attacks, for example that someone can break into the coffee machine over the Internet and alter the composition of the coffee [Wri08].

Within the last decade, the focus of the embedded software community paid more attention onto security of software-based applications. Today, most of the software updates fix security bugs and provide only little additional functionality. At the same time, the number of embedded electronic devices including at least one processor is increasing.

The awareness of security in digital systems lead to investigation of secure communication standards, for example SSL (Secure Socket Layer) [FKK96], the im-

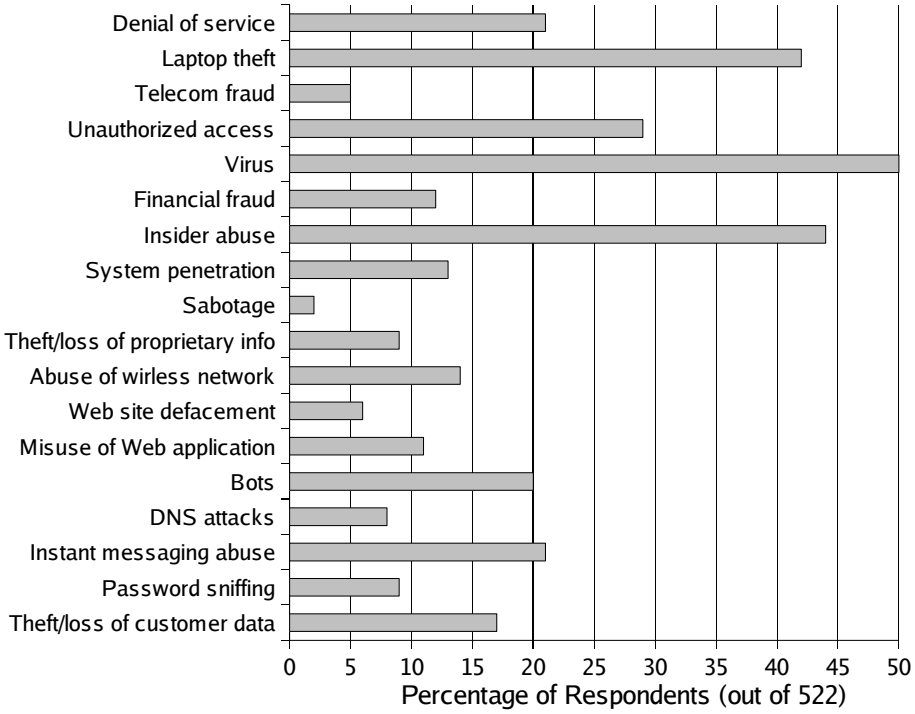


Figure 1.1: Security attacks reported in the CSI Computer Crime and Security Survey [Ric08], where 522 US-companies reported their experience with computer crime for the year 2008.

plementation of cryptographic methods, for example AES (Advanced Encryption Standard) [Fed01], a better review of software code to find vulnerabilities, and the integration of security measures into hardware. Nevertheless, Figure 1.2 shows that the vulnerability of digital systems increased rapidly over the last years. The main cause for vulnerability are software errors through which a system may be compromised. The software of embedded systems moves from monolithic software towards module-based software organized and scheduled by an operating system. By means of modern communication structures like the Internet, the software on embedded systems may be updated, partially or completely. These update mechanisms and the different communication possibilities open the door for software based attacks on the embedded system. For example, the number of viruses and trojans on mobile phones increased rapidly over the last years. One main gateway for these attacks are buffer overflows. A wrong jump destination or a wrong return address from a subroutine might cause an execution of infiltrated code (see also Section 1.4.1).

However, also hardware errors can lead to the vulnerability of a system. For example, Kaspersky shows that it is possible that the execution of appropriate instruction sequences on a certain processor can lead to an adoption of control of the system

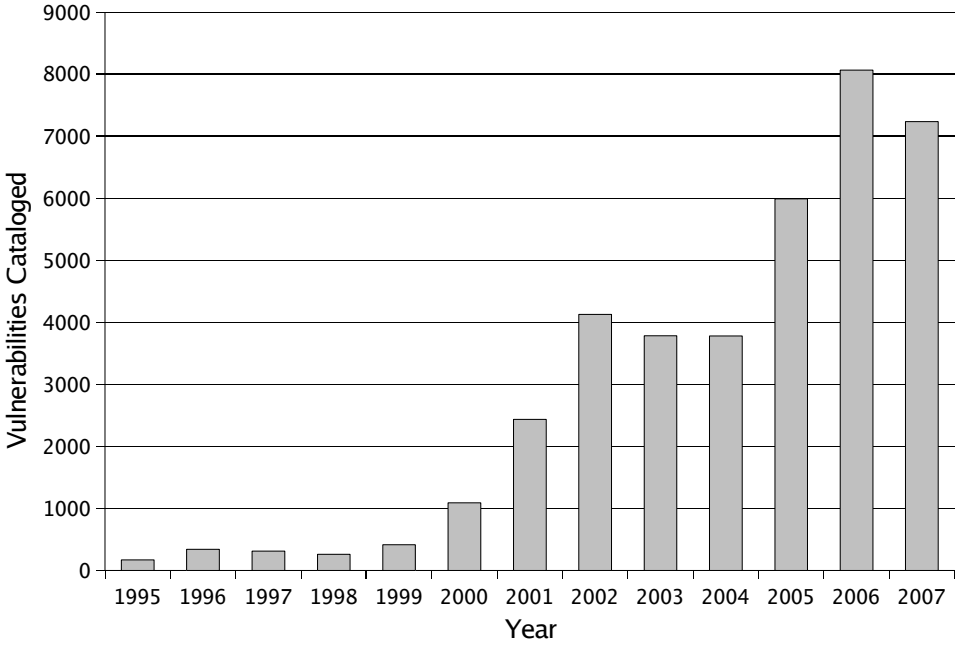


Figure 1.2: Vulnerability of digital systems reported to US-CERT between 1995 and 2007 [US-08].

by an attacker [KC08]. In this case, it does not matter which operation system or software security programs are running on the system.

A common objective for attackers are sensitive data, which are stored inside a digital system. To reach this objective, attackers are not only bound to software attacks. Hardware attacks, where the digital system is physically penetrated to gather information over the security facilities, or extract sensitive information are also practical. If an embedded device stores secure data, like a cryptographic key, attackers may try to read out this secret data by physically manipulating the processor on the embedded device. This may be done by differential fault analysis (DFA) [BS97] or by specific local manipulation on control registers inside the processor (see also Section 1.4.2). The attackers goal thereby is to execute infiltrated code or deactivate the protection of the secured data which may result from the manipulation of the program counter.

Another relevant security aspect in embedded systems is *intellectual property protection* (IPP). In this work, mainly copyright is in focus. Due to shorter design cycles, many products can only be developed with acquired hardware cores or software modules. Those companies selling these cores and modules naturally have a high interest in securing their products against unlicensed usage. Figure 1.3 shows the estimated percentage of unlicensed software used in different areas of the world. Also, calculated revenue losses are shown. Additionally, many unlicensed hardware IP cores are

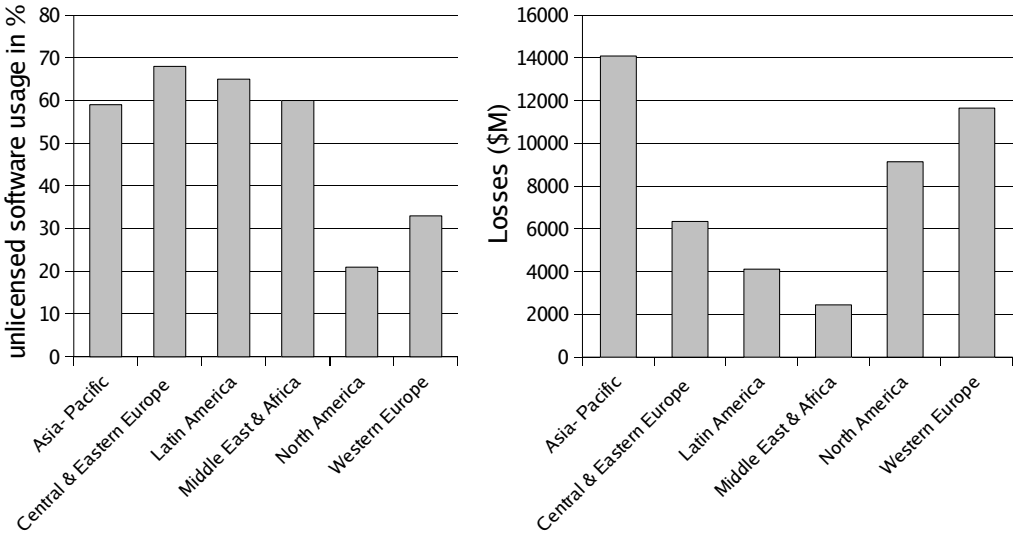


Figure 1.3: On the left side, the percentage of the usage of unlicensed software is shown in different areas of the world. On the right side the corresponding losses in million US-Dollars are depicted [All07].

used in products. At the RSA conference in 1998, it was estimated, that the adversity of the usage of unlicensed IP cores approaches 1 Billion US$ per day [All00].

Why Reliability?

In an integrated circuit, *permanent errors* and *transient faults* may occur. The difference between defects, faults, and errors is described in Section 1.2.2. Permanent errors are known since the invention of integrated circuits. Major causes of permanent errors are production defects and design errors. On the other hand, a transient fault corrupts the correct value of a signal for a short period of time. After the effect's duration, the correct value is usually recovered and the circuit is not physically damaged. Transient faults can be caused by on-chip perturbations, like power supply noise or external noise [NX06]. The main cause for external noise inside the earth's atmosphere are high energy neutrons from cosmic ray interactions and alpha particles from nuclear reactions. In space, the main source for radiation are high energy cosmic rays and high energy protons from trapped radiation belts [Joh00]. Transient faults caused by radiation are also called *soft errors* in the literature (see also Section 1.3.4).

Dynamic memory structures are particularly sensitive to transient faults. Since the 1970s, developers have dealt with soft errors in large dynamic memory structures [GWA79]. With the ongoing shrinkage of transistor sizes and on-chip structures and the reduced power supply voltage (see Figure 1.4), the problem of transient faults

becomes even more important. Meanwhile, transient faults do not only occur in memories, even logic and registers in IP cores suffer from a decreased reliability caused by transient faults. Mitra and others [MSZ+05] show that the contribution of the estimated *soft error rate* (SER) of various elements for typical modern designs (e.g., microprocessors, network processors, and controllers) is:

- 49% for sequential elements, like flip-flops or latches,
- 40% for unprotected SRAM (Static RAM), and
- 11% for combinatorial logic elements.

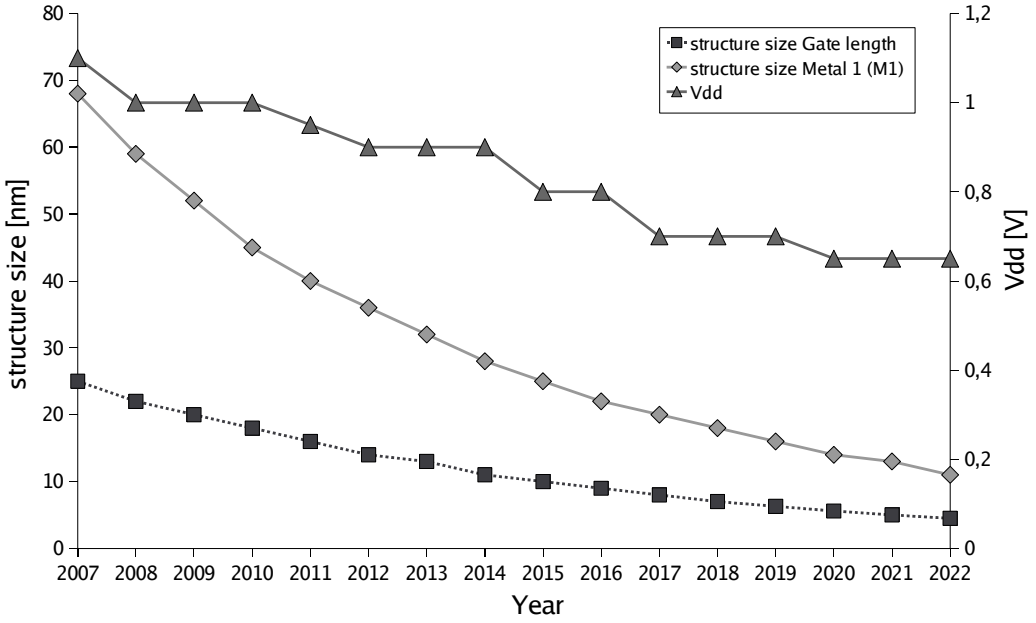


Figure 1.4: Estimated shrinkage of structure sizes and reduction of supply power (V_{dd}) for future integrated circuits. The values are taken from the High-Performance Logic Technology Requirements Table in [ITR07].

Baumann shows in [Bau05] that the system soft error rate (which can be compared to soft errors per area) for *DRAMs* (dynamic RAMs) is mostly independent from technology scaling (transistor and structure size). The system soft error rate for *SRAMs* and combinatorial/sequential logic is dramatically increased with the reduction of the feature size in new technologies. The multi bit soft errors rate is also increasing with further technologies [SSK+06]. This shows us that transient faults and soft errors are not limited to dynamic memory structures, and in the future, IP cores must deal with an increased soft error rate.

Another challenge to build reliable embedded systems in the future is the increasing process variability. The random dopant fluctuation of transistors will increase in future technologies, because of the discreteness of dopant atoms in the gate channel [Bor05]. The left side of Figure 1.5 shows the mean number of dopant atoms in a transistor channel over varying technology sizes. In the 32 to 16 nm technology generation, we will have only tens of dopant atoms. A small variation of dopant atoms may cause a huge variation of the transistor properties. The second source for transistor variations is sub-wavelength lithography, which results in line-edge roughness and several other effects, which may cause transistor variations [Bor05]. The right hand side of Figure 1.5 shows possible transistor variation increasing in the future.

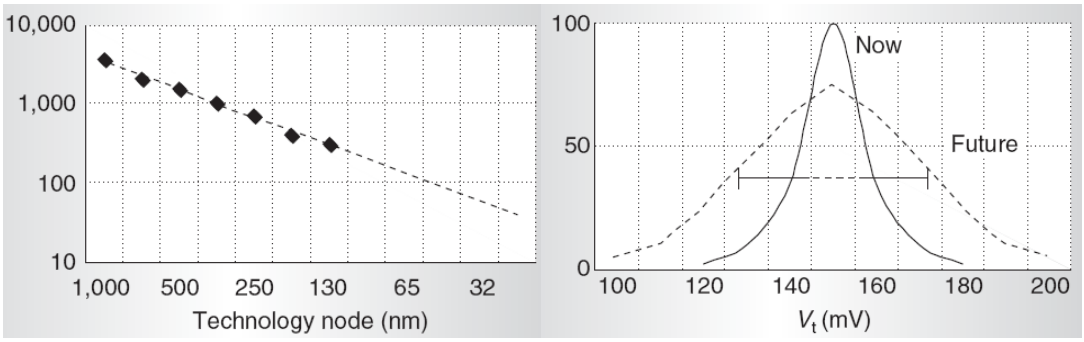


Figure 1.5: On the left side, the mean number of dopant atoms in a transistor channel is shown over different technologies. The right side shows the actual and the possible future variation of the threshold voltage V_t of transistors. Both figures are taken from [Bor05].

Also, the power dissipation density will increase into dimensions of over $100\frac{W}{cm^2}$. Future technologies expand the distribution of physical parameters (e.g., t_{ox}, L_{eff}, W_{eff}, doping, V_t) disproportionately. The timing is only predictable in a small range because of the variation of the wire delays by increased synchronization errors. These errors are caused by different voltage or clock islands, and by massive capacitive/inductive crosstalk. The consequences are decreased reliability and a lifetime of complex, very large scale integration systems and dramatically decreasing yield of todays strategies. A design flow assuming the worst-case is not applicable in the future, because this results in a design with a large power consumption which has a high impact on reliability.

This shows us that transistors will get more and more unreliable in the future. The great challenge is to design reliable systems from unreliable components [Bor05]. In conclusion, reliability in embedded systems is getting increasingly important. In the past, the need for additional functions to improve the reliability of the system by monitoring and correcting errors was only given for safety-critical systems which must have a high fault tolerance like banking mainframes, control systems of nuclear

plants or chip cards. In the future, the need for reliability-preserving and -increasing techniques will also become substantial for consumer products, like personal computers or, in our case, embedded systems.

Why IP Cores?

With every new chip generation, the logic density and thus the chip complexity in terms of transistors per chip rapidly increases (see Figure 1.6). This growth is higher than the design productivity increase of the last years. Additionally, the market requires shorter product cycles, which intensify this problem. This creates the design productivity gap between what we are able to build and what we can design. To close the gap, many innovations in design technologies are applied to increase the productivity. One of these innovations is the reuse of IP cores, which boosts the productivity of a design team by 200% according to [ITR07]. Only by reusing IP cores are we able to keep up future design productivity with the technologies improvements in chip manufacturing (see Figure 1.6).

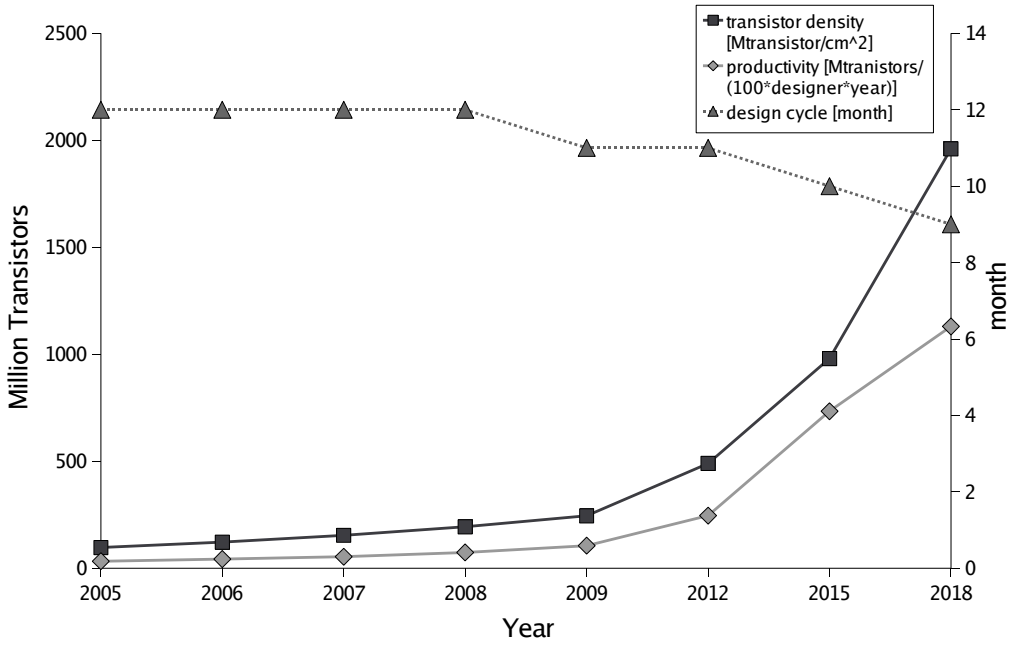


Figure 1.6: The increasing transistor density in $\frac{MTranistors}{cm^2}$ and the productivity in $MTransitors$ per design year with a team of 100 designers are shown. Also, the design cycle in months is depicted [ITR05].

Previously designed cores, like CPUs, buses, or cryptographic cores can be reused by new projects or sold as IP cores to other users. The advantage besides the increased productivity is that designers or whole companies have the possibility of specializing

on specific cores which may introduce an additional unique feature. Many companies base their business model on the sale of IP cores (e.g., ARM). Figure 1.7 shows the trend of rising core reuse in digital designs.

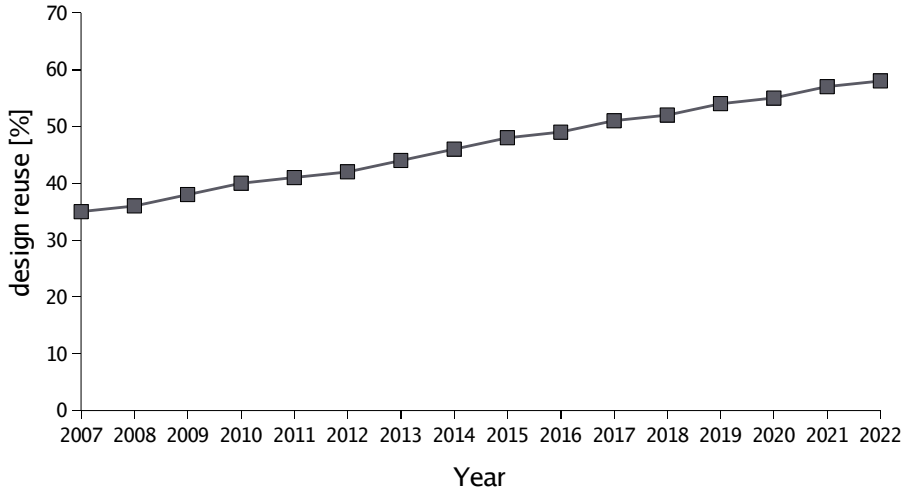


Figure 1.7: The percentage of reused IP cores compared to all designed logic will be raised in the future [ITR07].

IP cores can be delivered at different design levels. Possible distribution levels are *RTL* (e.g., VHDL or Verilog code), *logic* (e.g., EDIF netlists), or *device level* (e.g., layouts for ASICs or bitfiles for FPGAs). To improve the design and trade of IP cores as well as the interface between IP cores, the *Virtual Socket Interface (VSI) Alliance* was founded in 1996 [See99]. The VSI Alliance accounts for significant barriers for the trade with IP cores. One of these barriers is the lack of protection against unlicensed usage [All00].

Future IP cores should not only be resistant against unlicensed usage. They should also integrate state of the art reliability and security features at IP core level, such as autonomic error detection and correction methods.

Why FPGAs?

FPGAs (Field Programmable Gate Arrays) have their roots in the area of *PLDs* (Programmable Logic Devices), such as *PLAs* (Programmable Logic Arrays) or *PALs* (Programmable Array Logics). Today, FPGAs have a significant market segment in the microelectronics and, particularly in the embedded system area. The advantages of FPGAs over ASICs are their flexibility, the reduced development costs, and the short implementation time. Also, developers have a limited implementation risk, a), because of the easy possibility to update an erroneous design and b), because of the

awareness, that the silicon devices are proofed and the underlying technology operates correctly under the specified terms.

The main advantage of FPGAs is their reconfigurability. The demand for flexibility through reconfigurability will rise according to ITRS [ITR07] from 28% of all functionalities in 2007 until to an estimated 68% in the year 2022. Note that ITRS also takes into account software running on a microprocessor which can be updated. Furthermore, many FPGA devices support dynamic partial reconfiguration, which means that during runtime, the design or a part of it can be reconfigured. With this advantage, we can envisage new designs with new and improved possibilities and properties, like an adaptive design, which can adapt itself to a new operation environment. Unfortunately, dynamic reconfiguration is currently used rarely due to the lack of improved design tools which increases the development costs for dynamic reconfiguration. But now, improved design tools for partial reconfiguration are starting to become available, like the ReCoBus-Builder [KBT08, KHT08] or Xilinx Planahead [DSG05]. Nevertheless, dynamic reconfiguration for industrial designs is in its infancy, and it will take several years to use all the great features of FPGAs.

In the last years, the main application area of FPGAs were in small volume embedded systems and rapid prototyping platforms, where ASIC designs can be implemented and verified before the expensive masks are produced. Nevertheless, the FPGA usage in higher volume market rises, mainly due to lower FPGA price, higher logic density and lower power consumption. Furthermore, due to shorter time-to-market cycles (see Figure 1.6) and rising ASIC costs, FPGAs are breaking more and more into traditional ASIC domains. On the other hand, FPGAs are becoming competitors in the (reconfigurable) DSP domain with multi-core and coarse-grain reconfigurable architectures, as well as from graphic processing units (GPU) where DSP algorithms are adapted to run on these architectures. Nevertheless, these architectures suffer from the lack of flexibility and today, only FPGA technology is flexible enough to implement a heterogeneous reconfigurable system-on-a-chip.

Why ASICs?

Besides the advantages and the success of FPGAs, there still exists a huge market for traditional *ASICs* (Application Specific Integrated Circuit). ASICs are designed for high volume productions, where small cost-per-unit is important, as well as in low power and high performance applications and designs with a high logic density.

The implementation of a core on an ASIC instead of an FPGA (both 90 *nm* technology) may require 40 times less area, may speed up the critical path by a factor between 3 and 4, and may reduce the power by a factor of about 12 [KR06]. Here, we see that the big advantage of ASICs over FPGAs is the higher logic density, which results in significantly lower production cost per unit. The disadvantages of ASICs are the higher development and the higher initial production costs (e.g., masks, package design, test development [Kot06]). Therefore, the decision for using ASICs or

FPGAs due to minimization of the total costs is highly dependent on the production volume. Figure 1.8 shows a comparison of the total costs between ASICs and FPGAs in different technology generations over the production volume. The ASIC graphs start with higher costs due to the high initial production costs, but with a lower slope due to cheap production costs per unit. The initial cost of ASICs increases from technology generation to generation, mainly because of the increasing chip and technology complexity and logic density. FPGA designs have lower initial costs, but higher costs per unit. In summary, the total costs of a design using FPGA technology is lower until reaching a certain production volume point. However, according to Xilinx [RBD+01] this point is shifting for each technology generation in the direction of higher volumes.

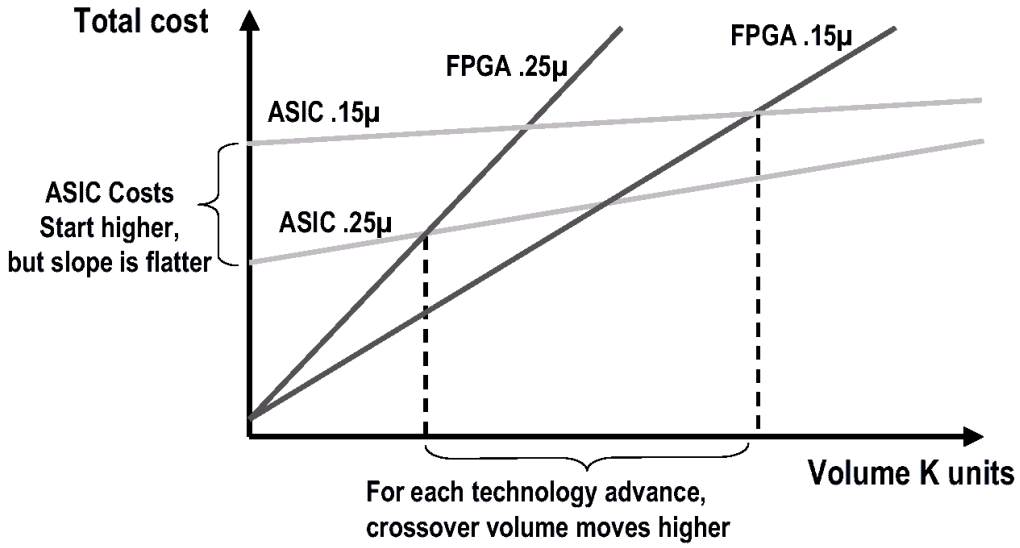


Figure 1.8: This figure from [RBD+01] shows a comparison of the total costs of FPGAs and ASICs in different technology generations over the production volume. With every new technology generation, the break even point between the total costs of FPGAs and ASICs designs is shifted more and more to the ASIC side. As on implication, one may expect the market for FPGAs to grow.

Nevertheless, besides the total costs discussion, there exist many design solutions, especially in the area of embedded systems, which can only be implemented using ASIC technology. Examples include very low power designs and high performance designs.

Before summarizing the major contributions of the thesis with respect to the above topic, a set of definitions is in order.

1.2 Definitions

In this section, we introduce necessary definitions of terms with respect to security and reliability of embedded systems that will be throughout this thesis. First, definitions in the field of dependability and the difference between defects, faults, and errors are outlined. After the categorization of faults and errors, definitions stemming from the area of security attacks are presented. Finally, different types of overhead, which are indispensable for additional security and reliability functions, are described.

1.2.1 Dependability and its Attributes

The dependability of a system is defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as: *"... the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers ..."* [IFI90]. According to Laprie and others [ALR01], the concept of dependability consists of three parts: the threats to, the attributes of, and the means by which dependability is attained (see Figure 1.9).

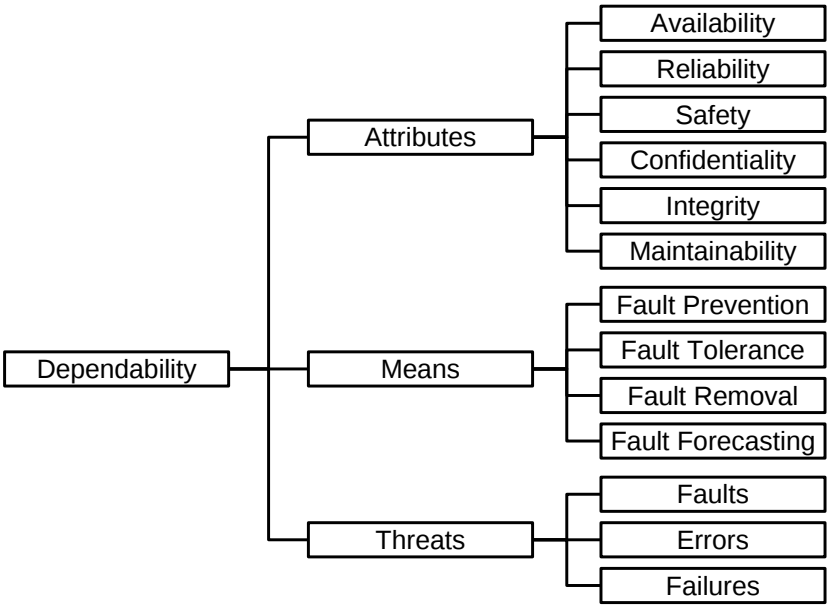


Figure 1.9: The relationship of dependability between attributes, threats and means [ALR01].

The attributes of dependability are a way to assess the trustworthiness of a system. The attributes are: *availability*, *reliability*, *safety*, *confidentiality*, *integrity*, and *maintainability*.

Availability

The availability is considered as the readiness for correct service [ALR01]. This means that the availability is a degree of the possibility to start a new function or task of the system. Usually, the availability is given in the percentage of time that a system is able of serving its intended function and can be calculated using the following formula:

$$Availability = \frac{Total\ Elapsed\ Time - Down\ Time}{Total\ Elapsed\ Time} \tag{1.1}$$

Availability is also often measured in *"nines"*. Two nines means an availability of 99%, three nines means 99.9% and so on. Table 1.1 shows the maximal downtime within a year for different availability values.

Availability	Percentage	8-hour day	24-hour day
Two nines	99%	29.22 hours	87.66 hours
Three nines	99.9%	2.922 hours	8.766 hours
Four nines	99.99%	17.53 mins	52.60 mins
Five nines	99.999%	1.753 mins	5.260 mins
Six nines	99.9999%	10.52 secs	31.56 secs

Table 1.1: The maximal annual downtime of a system for different values of availability, running either 8 hours or 24 hours per day [Rag06].

Reliability

Reliability is defined as the ability of a system or component to perform its required functions under well-defined conditions for a specified time period [Ger91]. Laprie and others transcribe the reliability with the continuity of correct service [ALR01]. Important parameters of reliability are the failure rate and its inversion, the *MTTF* (mean time to failure). Other parameters, like the *MTBF* (mean time between failures) include the time which is necessary to repair the system. The MTBF is the sum of MTTF and the *MTTR* (mean time to repair).

Safety

Safety is the attribute of a safe system. This means that the system cannot lead to catastrophic consequences for the users or the environment. Safety is relative, the elimination of all possible risks is usually impossible. Furthermore, the safety of a system cannot be measured directly. It is rather a subjective confidence of the system. Whereas availability and reliability avoid all failures, safety avoids only the catastrophic failures, which are only a small subset.

Confidentiality

The confidentiality of a system describes the absence of unauthorized disclosure of information. The International Organization of Standardization (ISO) defines the confidentiality as *"ensuring that information is accessible only to those authorized to have access"* [ISO05]. In many embedded systems (e.g., cryptographic systems), it is very important to secure the information (e.g., the secure key) stored inside the system against unauthorized access. But also the prevention of unlicensed usage of software programs or hardware cores are topics of confidentiality. Confidentiality is, like safety, subjective and cannot be measured directly.

Integrity

Integrity is the absence of improper system state alternation. This alternation can be an unauthorized access to alter system information inside the system, which are necessary for the correctness of the system. Furthermore, the system state alternation can also be a damage or modification of the system. System integrity assures that no part of the system (software or hardware) can be altered without the necessary privileges. Also, the IP core verification to ensure the correct creator and the absence of unauthorized supplementary changes can elevate the integrity of a system. Integrity is the precondition for availability, reliability and safety [ALR01].

Maintainability

Maintainability is the ability to undergo repairs and modifications. This can be done to repair errors, meet new requirements, make further maintenance easier, or to cope with a changed requirement or environment. A system with a high maintainability may have a good documentation, a modular structure, is parameterizable, uses assertions and implements built-in self tests.

Security

Security is defined as a combination of the attributes (1) *confidentiality* (the prevention of the unauthorized disclosure of information), (2) *integrity* (the prevention of the unauthorized amendment or deletion of information), and (3) *availability* (the prevention of the unauthorized withholding of information) [ITS91]. An alternative definition for security could be the absence of unauthorized access to the system state [ALR01]. The prevention or detection of the usage of unlicensed software or IP cores can also be seen as a security aspect (confidentiality) as well as the prevention of the unauthorized alteration of software or IP cores (integrity). Like safety, security shall prevent only a class of failures which are caused by unauthorized access or unauthorized handling of information.

1.2.2 Fault, Error, Failure

Faults, *errors*, and *failures* are the threats which affect the dependability (see Figure 1.9).

Failure

A system is typically composed of different components. Each component can be further subdivided into other components. All of these system components may have internal states. If a system delivers its intended function, then the system is working correctly. The intended function of a system can be described as an input/output or interface specification which defines the behavior of the system on the system boundaries with its users or other systems.

The system interface specification may not be complete. For example, it is specified that an event occurs on the output of the system, but the time of this event to occur is not exactly specified. So, the system behavior can vary without violating the specification. If the specification is violated, the system fails. A failure is an event which occurs when the system deviates from its interface specification (see Figure 1.10).

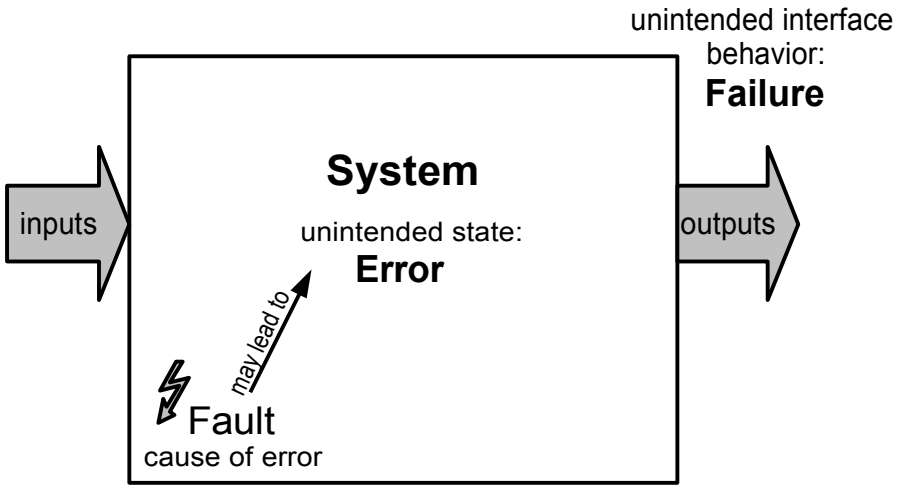


Figure 1.10: Faults may lead to an error, which may also lead to a system failure.

Errors

If the internal state of a component deviates from the specification (the specification of the states of the component), the component is erroneous and an error occurs. An error is an unintended internal state whereas a failure is an unintended interface

behavior of the system. An error may lead to a failure. But it is also possible that an error occurs and does not lead to a system failure, because of the component is currently not used or the error is detected and corrected fast enough. Errors can be transient or permanent. Transient errors caused by transient faults usually occur in systems without feedback. In systems with feedback, an error might be permanent by affecting all following states. In this case, the error only disappears by a reset or by shut down of the system.

Faults

A fault is defined as a defect that has the potential to cause an error [Jal94]. All errors are caused by faults, but a fault may not lead to an error. In the latter case, the fault is masked out and has no impact on the system.

For example, consider the control path of a processor core. A fault like a single event transient fault, caused by an alpha particle impact, occurs on one signal of the program counter between two pipeline stages. If the time of occurrence is near the rising active clock edge, an error may occur. Otherwise, if the time of occurrence is far away form the rising edge of the clock, the fault does not lead to an error. The erroneous program counter value can now lead to a system failure, if the wrong subroutine is executed and the interface behavior differs from the specification. Otherwise, if an error detection technique, like a control flow checker, as introduced later in Chapter 4, is used, the error can be detected after the fault appearance, and the error may be corrected by a re-execution of the corresponding instruction. But, this additional re-execution needs several clock cycles to restore the error free state. For real-time systems with very critical timing requirements, the possible output events might be too late and the system thus might still fail.

1.2.3 Fault and Error Categorization

Faults can be categorized into different classes. The main classes are: persistence, nature, and origin (see Table 1.2) [ALRL04, Kop97].

Faults			Errors	
persistence	nature	origin	location	effect
permanent sporadic transient periodic transient	chance intentional	development runtime	e.g. data path control path memory	value timing

Table 1.2: An overview of different faults and error classes.

The persistence of a fault can be *permanent* or *transient*. The class of transient faults can be further subdivided into *sporadic* and *periodic* faults. A permanent fault

is, for example, a broken wire. Alpha particle radiation on a chip can cause sporadic transient faults, and jitter in a clock signal is a periodic transient fault. It is important to know that transient faults can also lead to permanent errors. The nature of the fault can be *chance* or *intentional*. A chance fault occurs randomly with a specific probability, like faults from radiation. An intentional fault can be a security attack to a system or a faulty operation from the user. Intentional faults can be further subdivided into *malicious* intentional faults and *non-malicious* intentional faults (more about this in Section 1.2.5). The origin of a fault can be in the development phase of the system or at runtime. Physical phenomena like lightning strokes belong to runtime faults, whereas, design faults are caused in the development phase.

Errors can be categorized into different error classes (see Table 1.2). Here, we distinguish between the location and the effect class. Errors can be classified according to the location or components of their occurrence, for example, data path or control path errors. Value errors and timing errors belong to the effect class. For example, a value error occurs when an incorrect value of a register is caused by a single event upset, whereas a timing error occurs if the delay of a signal is too large, caused, for example, by a too high temperature.

There exist many other definitions of fault and errors classes in literature. The presented classes above present a minimal intersection between these different definitions.

1.2.4 Means to Attain Dependability

Means are ways to increase the dependability of a system. There exist four means, namely *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*.

Fault Prevention

Fault prevention deals with the question “How the occurrence or introduction of faults can be prevented?”. Design fault might be prevented with quality control techniques during the development and manufacturing of the software and hardware of a system. Fault prevention is further closely related to maintainability. Transient faults, like single event effects, might be reduced by shielding, radiation hardening, or larger structure sizes. Attacks might be prevented by security measures, like firewalls or user authentication. To prevent the usage of unlicensed programs or IP cores, the code (source, binary, or netlist code) could be delivered encrypted and only the authorized customer has the right cryptographic key to decrypt the code. To prevent the impart of the key, techniques like dongles or an authentication with MAC-Address can be used.

Fault Tolerance

A fault-tolerant system does not fail, even if an erroneous state is reached. Fault tolerance enables a system to continue operation in the case of one or more errors. This is usually implemented by error detection and system recovery to an error-free state. In a fault tolerant system, errors may occur, but they must be handled correctly to prevent a system failure.

The first step towards a fault tolerant system is *error detection*. Error detection can be subdivided into two classes: *concurrent error detection* and *preemptive error detection* [ALR01]. Concurrent error detection takes place at runtime during the service delivery, whereas preemptive error detection runs in phases where the service delivery is suspended. Examples for concurrent error detection are error codes (e.g. parity or CRC), control flow checking, or razor flip-flops [ABMF04].

Also, redundancy belongs to this class of error detection. One may distinguish three types of redundancy: *hardware*, *time* and *information* redundancy. To detect errors with hardware redundancy, we need at least two units where both results are finally compared. If they divert, an error occurred. On time redundancy, the system executes the same inputs twice, and both results are compared after the second execution. Information redundancy uses additional information to detect errors (e.g., parity bits). More information about redundancy methods can be found in Section 2.3.

BISTs (Built In Self Tests) or start-up checks belong to the preemptive error detection class.

The next step is the recovery from the erroneous state. Recovery consists of two steps, namely *error handling* and *fault handling*. Error handling is usually accomplished by *rollback* or *rollforward*. Rollback is done by using error-free states which are stored on certain checkpoints to restore the state of the system to an older error-free state. Rollback is attended by delaying the operation. This might be a problem in case of real-time applications. Rollforward uses a new error-free state to recover the system.

If the cause of the error is a permanent or periodic temporal fault, we need fault handling to prevent the system from running into the same error state repeatedly. This is usually done by *fault diagnosis*, *fault isolation*, *system reconfiguration* and *system reinitialization*. For example, in case at permanent errors in memory structures, the faulty memory column is identified and this column is switched to a reserved spare column. After the switch over, the column content must be reinitialized.

It is important to notice that fault tolerance is a recursive concept. The techniques and methods which provide fault tolerance should obviously themselves be resistant against faults. This can, for example, be done by means of replication.

Fault Removal

During the development phase and during the operational runtime, fault removal might be performed. At the development phase, fault removal consists of the following steps: *verification*, *diagnostics*, and *correction* [ALR01]. This is usually done by debugging and/or simulation of software and hardware. For the verification of fault tolerant systems, fault injection (see Section 4.2) can be used.

Fault removal during the operational runtime is usually done by *maintenance*. Here, faults can be removed by software updates or by the replacement of faulty system parts.

Fault Forecasting

Fault forecasting predicts feasible faults to prevent or avoid the fault or decrease the effect of the fault. This can be done by performing an evaluation of the system behavior with respect to fault occurrence and effects. Modeling and simulation of the system and faults are a common way to achieve this evaluation.

1.2.5 Security Flaws and Attacks

Faults affecting the security of a system are also called *flaws* [LBMC94]. In this work, the term flaw is used as a synonym of a fault, which leads to the degeneration of the security of a system. A flaw is therefore a weakness of the system which could be exploited to alter the system state (error). A threat is a potential event which might lead to this alternation and therefore to a security failure. The process of exploiting the flaw by a threat is called an *attack* (see Figure 1.11). A security failure occurs when a security goal is violated. The main security goals are the dependability attributes *integrity*, *availability*, and *confidentiality*. The difference between a flaw and a threat is that a flaw is a system characteristic, whereas a threat is an external event.

A flaw can be *intentional* or *inadvertent*. Intentional flaws can further be *malicious* or *non-malicious*. An intentional malicious flaw is, for example, a trojan horse [And72]. An intentional non-malicious flaw could be a communication path in a computer system which is not intended as such by the system designer [LBMC94]. An inadvertent flaw could be, for example, a bug in a software program, which enables unauthorized persons with specific attacks to read protected data.

Like other faults, flaws can also be further categorized using the origin of the flaw and the persistence. The origin can be during the development (e.g., the developer implement a back door to the system), or during operation or maintenance. Usually, the flaws exist for a longer period of time (e.g., from the flaw arise until the flaw is disappeared by a security update). But also special flaws exists, which only appear on certain situations (e.g., the year 2000 problem; switching from year 1999 to 2000).

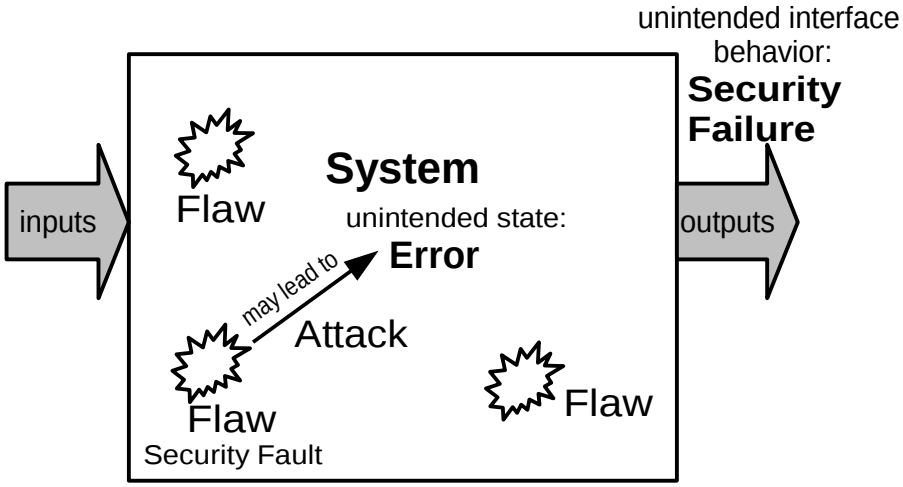


Figure 1.11: Flaws are security faults, which lead to errors if they are exploited by attacks. The state alternation in case of an attack may lead to a security failure.

Attacks can be classified using the security goals or objective of the attack into *integrity*, *availability*, and *confidentiality attacks*. Integrity attacks break into the system and change part or the whole system (software or hardware) or of the data. The goal of availability attacks is to make a part or the whole system unavailable for user requests. Finally, the goal of confidentiality attacks is to gather sensitive or protected data from the system.

Furthermore, if an attack is successful, new flaws can be generated as a result from the attack. For example, a flaw in software is exploited by a code injection attack (see Section 1.4) and the integrity of the system is injured by adding a malicious software routine. This software routine opens now intentional malicious flaws, which can be used by confidentiality attacks to gather sensitive data.

To describe all attacks using this terminology is not easy. For example, a copyright infringement where someone unauthorized is copying an IP core. The result of the attack is a reversal of confidentiality. Here, the sensitive data is the core itself. The erroneous state is the unauthorized copy of the IP core. But what is the flaw which makes the attack possible? Here, we must assume that the ability to easily copy an IP core is the flaw. This example teaches us that flaws exist even on reasonably secure systems and cannot be easily removed. On every system we must deal with flaws, which might affect the security as well as other faults which might affect the other areas of dependability.

1.2.6 Overhead

Methods for increasing security and reliability in embedded systems often have the drawback of additional overhead. To evaluate the additional costs of these methods, we can use the following criteria:

- *Area overhead* (hardware cost overhead),
- *Memory overhead*, and
- *Execution time overhead* (CPU time).

Analysis and quantification of the additional costs significantly depends on the given architecture and the implementation on a specific target technology.

Area Overhead

The straightforward method for measuring the area overhead of an additional core is to measure the chip area occupied by the core. Unfortunately, this method can only compare cores implemented in the same technology with exactly the same process (lateral dimensions). To become independent of this process, the *transistor count* may be used. However, information about the area of the signal routing is not included here. In most of the technologies and processes, signal routing requires little additional area because of the routing layers are above the transistors (in the third dimension). This also depends on the specific process.

The number of transistors, however, is a reasonable complexity indicator, only if the compared cores use the same technology (e.g., CMOS, bipolar). To compare the hardware costs of a core mapped onto different technologies, the *gate count* can be used. Here, the number of logical (two input) gates is used to describe the hardware costs. For comparing cores between ASIC and FPGA technologies, the count of *primitive components or operations*, like n-bit adders or multipliers, can be used.

Using primitive components or operations for the description of the overhead, one is independent of the underlying technology and process. Describing hardware cost in a higher hierarchy level, like primitive components or operations, however, is more inaccurate with respect to the real hardware costs than describing the overhead in chip area. The resulting chip area of the used primitive components depends highly on the technology and process and also on the knowledge of the chip designer or the quality of the tools.

Memory Overhead

The memory overhead for methods increasing the security and reliability can be measured by counting the additional *ram bits* used by the corresponding method. Memories embedded on the chip, so called internal memories, use hardware resources on

the chip and so they contribute to the area overhead. Nevertheless, the content of memories can be relatively easily shifted to a cheaper external memory, for example an off-chip system DRAM. So, we decided to handle the memory overhead separately. It must be taken into account that internal memory has higher hardware costs at the same size, but a lower latency. External memory is usually cheaper, but it involves additional hardware costs on the chip, as for example a DRAM controller. If a DRAM with the corresponding controller already exists on the chip, these resources might be shared to reduce the hardware cost.

Execution Time Overhead

Some methods for increasing the security and reliability have additional latency. This means that the result of the protected core or software appears later on the outputs than on the unprotected one. For hardware cores, latency is usually counted in additional clock cycles. For software programs, latency can be expressed in additional instructions which must be executed by the processor or in additional execution time of the processor. For example, some existing methods for control flow checking [GRRV03] generate additional instructions that are inserted into the original program running on the processor which is monitored. This might cause a timing impact for the user program which impact can be measured by additional execution time of the processor. The execution time depends on the processor and the number of executed additional instructions.

Also, if no additional software is executed on the processor and the processor is enhanced with additional hardware, some methods can stall [ZT09] the processor pipeline, slow down the execution of the user program, or insert additional pipeline steps [Aus99] without executing additional instructions.

For processor architectures, the execution time overhead can be measured by counting the additional pipeline steps. If the processor architecture executes one instruction in one pipeline step (in the best case one clock cycle), the number of additional executed instructions are also given in the number of additional pipeline steps.

1.2.7 IP Cores and Design Flow

The reuse of IP cores is an important step to decrease the *productivity gap*, which emerges from the rapid increase of the chip complexity and the slower growth of the design productivity. Today, there is a huge market and repertoire of IP cores which can be seen in special aggregation web sites, for example [Reu] and [Est], which administrate IP core catalogs.

The delivery format of IP cores is closely related to the *design flow*. The design flow consists of different *design flow* or *abstraction levels* which transfer the description of the core from the level where the core is specified into the final implementa-

tion. The design flow is dependent from the target technology. The FPGA and the ASIC design flow look similar, however, there exist differences at some steps.

Figure 1.12 shows a general design flow for electronic designs with FPGA and ASIC target technologies. This design flow view can be embedded into a higher system model view for *hardware/software co-design*, for example the *double roof model* introduced by Teich [TH07]. The depicted design flow implements the logic synthesis and the following steps in the double roof model. Furthermore, the different abstraction levels are derived from the *Y-diagram*, introduced by Gaijski [GDWL92].

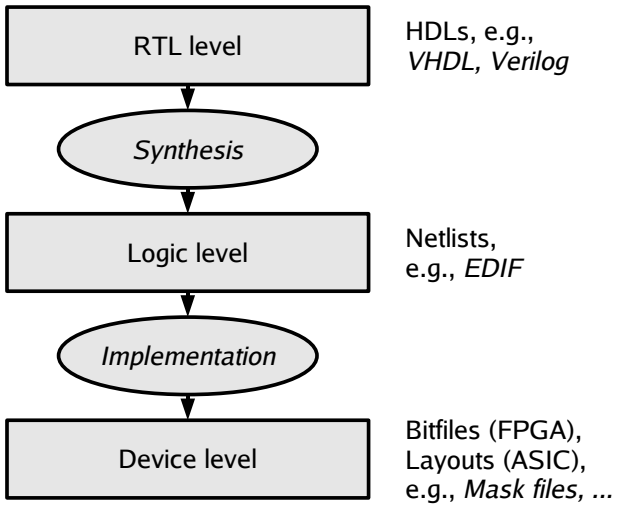


Figure 1.12: A general design flow for FPGA and ASIC designs with the synthesis and implementation steps and the different abstraction levels.

The different abstraction levels are the *register-transfer level*, the *logic level*, as well as the *device level*. Designs specified at the register-transfer level (RTL) are usually described in *Hardware Description Languages* (HDLs) like VHDL or Verilog, whereas designs at the logic level are usually represented in *netlists*, for example, *Electronic Design Interchange Format* (EDIF) netlists. At the device level, FPGA designs are implemented into bitfiles, while ASIC designs are usually represented by their chip layout description. The transformation of an HDL-model into an netlist-model is called logic *synthesis*, whereas the transformation of a netlist-model into a target depended circuit is called *implementation*. The implementation consists of the aggregation of the different netlist cores and subsequent *place and route* step. The *technology mapping* can be done in the synthesis or in the implementation step, or in both. For example, the Xilinx FPGA design flow maps the logic to device dependent *primitive cells* (LUT2, FF, etc.) in the synthesis step, whereas the mapping of these primitive cells to slices and *configurable logic blocks* (CLBs) is done in the implementation step [Xilb].

IP cores can be delivered at all different abstraction levels in the corresponding format: on the RTL as *VHDL* or *Verilog* code, on logic level as *EDIF* netlist, or on the device level as *mask files* for the ASIC flow or as FPGA depended (partial) *bitfiles*.

IP cores can be further categorized into *soft* and *hard cores*. Hard cores are ready to use and are offered into a target depended layout or bitfile. All IP cores which are delivered into an HDL or netlist format belongs to the soft cores. These cores need further transformations of the design flow to be usable. The advantages of soft cores are their flexibility for different target technologies and their can be parameterizable. However, the timing and the area overhead are less predictable compared to hard cores due the impact of the needed design tools. *Analog* or *mixed signal* IP cores are usually delivered as hard cores.

1.3 Faults in Embedded Systems

The faults inside a system-on-chip can be categorized into *permanent degeneration faults*, *manufacturing faults*, and *design faults*, as well as *transient faults*. Transient faults are *single event effects* (SEE) or temporary conditions on the chip, like power fluctuation or interconnect noise (see Table 1.3). Security flaws can be both, permanent or transient (see Section 1.4).

fault	fault class			error class
	persistence	nature	origin	effect
hot-carrier effect	permanent	chance	runtime	timing/value
electromigration	permanent	chance	runtime	timing/value
TDDB	permanent	chance	runtime	timing/value
manuf. stuck-at faults	permanent	chance	manufacturing	value
manuf. delay faults	permanent	chance	manufacturing	timing
design faults	permanent	chance	development	timing/value
SEU, SET	transient	chance	runtime	value
SEL	permanent	chance	runtime	value
internal noise	transient	chance	runtime	value

Table 1.3: Categorization of different faults which may appear in an embedded system.

1.3.1 Degeneration Faults

Degeneration faults are, for example, caused by the *hot-carrier effect* [GHB07], by *electromigration* [CRH90], or by *time-dependent dielectric breakdown* (TDDB)

[San]. All these faults are *permanent chance runtime faults* which at first lead to timing errors and later, particularly electromigration, to value errors like open or short circuits.

Electromigration is caused by ion movement in the direction of the current flow [NX06, CRH90]. This leads to voids, which are able to open signal lines as well as mounds which have the ability to short the signal with an adjacent signal. Especially power signal lines suffer from electromigration, but also other signals are affected by the phenomenon. High temperature accelerates this effect.

Due to gate channel shrinking in every new process generation, the electrical field strength is increasing as well. This along with higher temperature leads to a higher tunneling rate of electrons or holes into the gate oxide. This so called *hot carrier effect* [NX06, GHB07] can lead to a drift of the threshold voltage of the transistor, which affects the timing behavior. If the transistor switching behavior of the critical path in a design is affected, this effect can lead to timing errors.

Another degeneration effect is *time-dependent dielectric breakdown* (TDDB) [San, Cro79]. During the operation of a CMOS transistor, the gate oxide is exposed to an electrical field. Caused by irregulations of the structure of the oxide, charges are trapped inside the oxide. This leads to a disturbed electrical field, where the field strength is intensified or alleviated locally. During the lifetime of the chip, this disturbance is increasing, due to more trapped charges. Localized, the electrical field can reach a extremely higher field strength, which leads to the dielectric breakdown after reaching a certain threshold value. This means that the oxide is destroyed by an electrical and a thermal runaway. Also this effect is accelerated by higher electrical fields and higher temperature.

1.3.2 Manufacturing Faults

Manufacturing faults are caused by permanent physical defects, which occur during the manufacturing process. These defects are, process defects, like *missing contact windows*, *parasitic transistors*, or *oxide breakdown*, as well as material defects, like *bulk defects* (crack, crystal imperfections), or *surface impurities* [BA02]. Also packaging defects, like *seal leaks* belong to the manufacturing defects. These physical defects lead to *stuck-at-0*, *stuck-at-1* or *stuck-at-open* faults as well as *bridge* faults [BA02] which may lead to value errors.

Also, signals which are after manufacturing too slow to meet given timing constraints are manufacturing faults which may cause timing errors. All manufacturing faults are *permanent chance faults* but emerge during the manufacturing process.

1.3.3 Design Faults

Design faults are permanent chance development faults which are caused by an incorrect specification or implementation of the developer. However, also design tools can cause design faults.

Design faults may occur in different abstraction levels, from the system architecture to the transistor level. A design fault in higher abstraction levels has naturally a higher impact on the system, e.g., a too slow microprocessor then on the RTL, or transistor level.

1.3.4 Single Event Effects

Single event effects (SEE) are *sporadic chance runtime faults* which are mainly caused by different types of energetic external radiation. This radiation can cause transient faults, like *single event upsets* (SEU) or *single event transient* (SET) [GSB+04], as well as permanent faults, like *single event latch-ups* (SELs) [MW04]. These faults usually cause value errors.

An SEU is a bit flip in a memory cell or register caused by the impact of an energetic particle, which generates a charge disturbance in the transistor channel. This effect is only of temporal nature and is non-destructive to the transistor. Mainly DRAM and SRAM suffer from these effects, but also registers and latches in IP cores are affected.

If the impact is located into combinatorial logic, a transient pulse on a combinatorial signal may occur. If this pulse is wider than the logic transition time, which is possible in current CMOS technologies, the pulse is propagated through the combinatorial logic to a register or a latch [AAN00, GSB+04]. This effect is called *single event transient* (SET). Due to the characteristics of the combinatorial logic, these faults can be masked out. If we have an AND gate and one input is set to zero, SETs on the other inputs are blocked. On the other hand, SETs can also be duplicated by combinatorial logic if the SET propagates over many paths with different delays to the register. Reaching the register or the latch, the SET pulse manifests only into an error, if the time of arrival and the pulse width overlaps with a clock impulse. Therefore, the error rate of SETs is in contrast to SEUs highly dependent on the clock frequency but also on the characteristics of the combinatorial logic. Buchner and others [BBB+97] show that for cores which operate at a high clock frequency, the errors caused by SETs dominate the errors caused by SEUs.

SEUs and SETs are also called *soft errors*, because of the transient, non-destructive nature of these effects. Compared to other faults, soft errors are responsible for most failures of electronic systems [MW04].

Beside the two transient soft error types, the permanent *single event latch-up* (SEL) effect exists. Because of the different doped layers of an CMOS circuit, an inherent parasitic thyristor might exist. This parasitic thyristor has no effect on the circuit if

it is not active. However, the thyristor can be ignited by a heavy ionized particle impact. The result is an shortcut from the power supply signal to ground which exists as long the power is switched on. If the fault is not detected fast enough, the circuit is destroyed by thermal runaway.

The sources of SEEs inside the earth's atmosphere are *low-energy alpha particles*, *high-energy cosmic particles*, and *thermal neutrons* [MW04]. Outside the earth's atmosphere, the sources of SEEs are *high energy cosmic rays* and *high energy protons* mainly from trapped radiation belts [Joh00].

The low-energy alpha particles are generated from decay of radiation elements which are inside of mold compounds and in lead bumps, used for flip-chips. These alpha particles have an energy of 2 to 9 *MeV* (million electron volt). To generate an electron-hole pair in silicon, 3.6 *eV* are required. Therefore, an impact of one of these alpha particles can generate approximately one million electron-hole pairs in its wake [MW04] (see Figure 1.13). This leads to a charge drift in the depletion region and a current disturbance. If the charge drift is high enough, the transistor state is inversed and, for example, a memory cell is flipped.

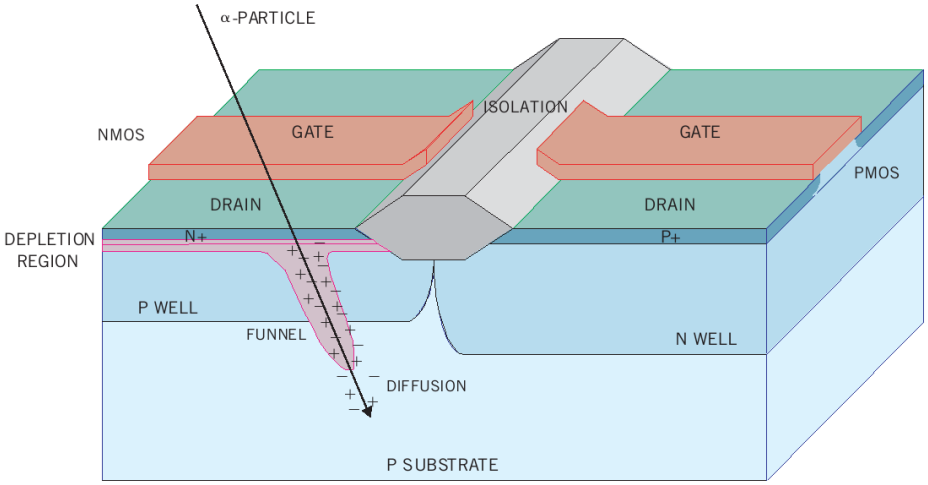


Figure 1.13: An alpha particle hits a CMOS transistor. The particle generates electron-hole pairs in its wake, which cause a charge disturbance. This event can cause an SEE fault [MW04].

High-energy cosmic particles react in the upper atmosphere or radiation belts and generate high-energy protons and neutrons. The generated neutrons have energies of 10 to 800 *MeV*, whereas the generated protons have an energy greater than 30 *MeV* [MW04]. Inside the earth atmosphere, we must deal with high-energy neutrons from the upper atmosphere whereas in space mainly all SEEs are produced from high-energy protons [Joh00]. If these particles collide with the silicon nuclei, further ionized particles are generated. Protons in space environment, which exists there

in energies below 600 MeV as well as ionized particles from silicon collisions can generate electron-hole pairs in the substrate. Unlike alpha particles, these ionized particles have usually a higher energy, which results in a higher electron-holes rate. The neutron effect depends on the altitude. In an airplane, the effect can be 100 to 800 times worse than on see-level [MW04].

Thermal neutrons are low-energy particles which come from the upper atmosphere and have reached the thermal equilibrium due to the loss of their kinetic energy. These neutrons usually have an energy of 25 meV. If the Boron-10 isotope is present on the chip, which appears in large quantiles in *BPSG* (Boron-Phosphor-Silicate-Glass) dielectric layers, these neutrons are easily fetched by the isotope. This event results in fission where an alpha particle and a gamma ray is generated, which might lead to the SEE. If the Boron-10 isotope is present, then this is the main cause of SEEs [MW04].

SEEs are becoming increasingly important, because of the sensitivity of integrated circuits to radiation is increasing due to smaller structure size and decreased power supply voltage. Both trends result in reduction at the charges stored inside a node which increases the probability of the appearance of an SEE [AAN00, NX06].

1.4 Attacks on Embedded Systems

There exist two ways for categorization of attacks. The first way is to categorize attacks by the violated security goals. The other way is to describe how the attack is realized and which way the attacker chose to compromise the system [Rag06, RRKH04].

Using the first categorization schema, the main security goals are *integrity*, *availability*, and *confidentiality* (see Figure 1.14 above, and Section 1.2.5). Attacks which compromise integrity can be further subdivided into *manipulation of data*, *manipulation of software* or *IP cores*, as well as *forging of authorship*. Attacks which may paralyze a system compromise the availability. Attacks to compromise the confidentiality of a system can be subdivided into *gathering of sensitive data* like passwords, keys, program code, or IP cores, and *getting access control* to a system. Additionally, *copyright infringement* compromises the confidentiality of the author of the core.

The means used to launch the attacks or the ways how the attack is realized can be categorized into *invasive* and *non-invasive attacks* (see Figure 1.14 below). Both groups can further be subdivided into *logical* and *physical attacks* [RRKH04]. Physical attacks typically require relatively expensive equipment and infrastructure, especially physical invasive attacks. Whereas for logical attacks, usually only a computer or the embedded system is needed.

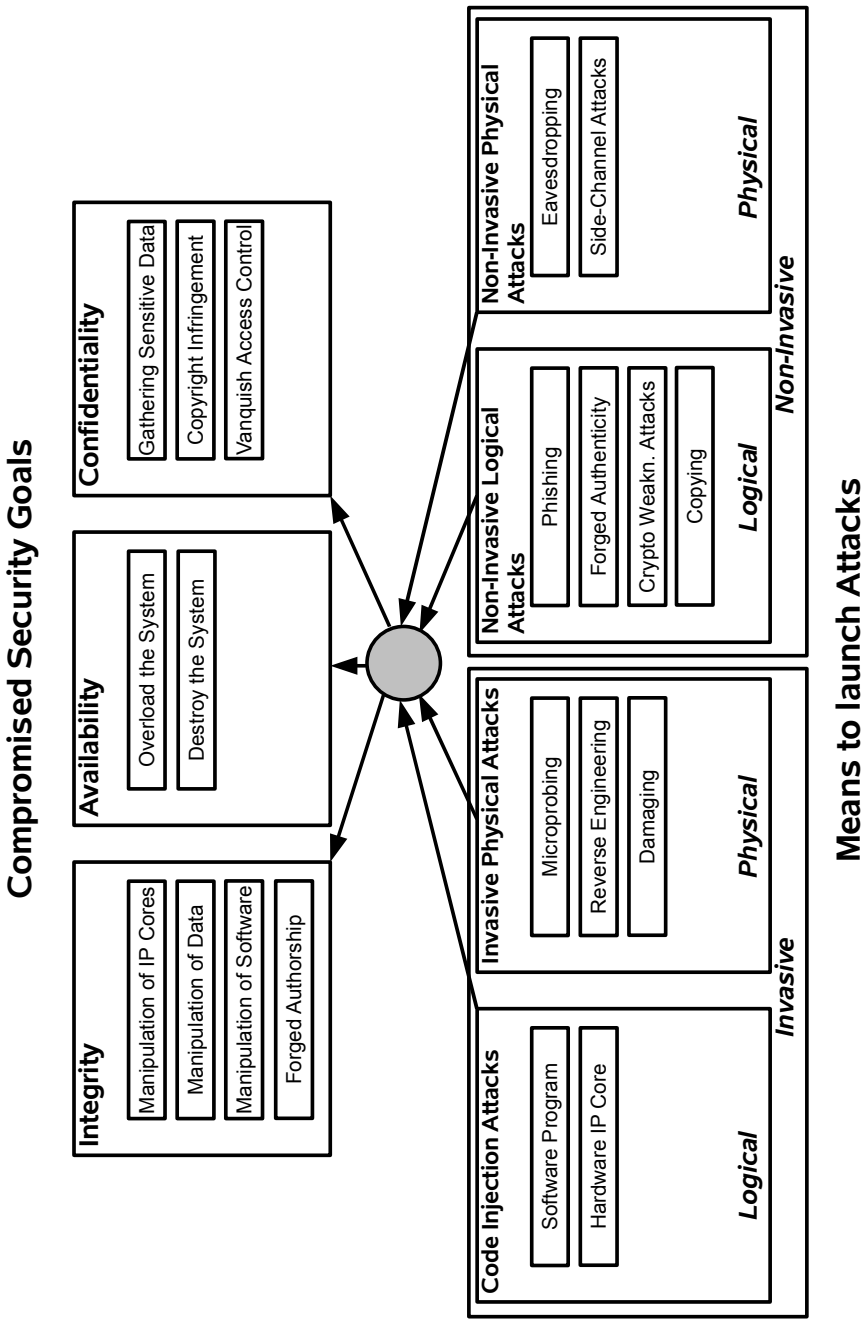


Figure 1.14: Attacks can be categorized with the compromised security goals or the attack goals (above) and with the means to launch the attack (below). The different means of attacks can invalidate different security goals.

1.4.1 Code Injection Attacks

Code injection assaults are attacks where the code integrity of a system is compromised. This can be the integrity of software as well the integrity of executed bitfiles in a reconfigurable system, such as FPGAs. The goals of code injection attacks are manifolded. The demolished integrity of further program code or sensitive data, the paralysis of the system as well as getting access to sensitive data are in the foreground of the attacker.

Code injection attacks bring the system under the control of the attacker. Programs inserted by the attacker, may easily read or alter the sensitive data and forward the data to interfaces where the data can be collected.

To gain control over a system, the attacker must first insert a routine, which performs the intended behavior, into memory. This routine may, for example, read out secured data, deactivate security protection, open gateways for the attackers, or load another infiltrated code from the Internet. The malicious code can be inside the processed input data which is loaded into the memory by the processor. The second step is bringing the processor in a state to execute the inserted attacker's code. This can be done by manipulation of the program flow.

One way to achieve this is by utilizing *buffer overflows* for *smashing stacks*. Most programs are structured into subroutines with its own local variables. These variables and also the arguments and the return address are stored in memory segments called stacks. The return address is usually the first on the stack and the local variables are concatenated on the bottom. Normally, like in the C programming language, the content of array variables are written from bottom to the top, and if the range is not checked, the return address can be overwritten (see Figure 1.15). The attacker can manipulate the input data in a way that the return address is overwritten with the address of his malicious code. On the return, the malicious code is executed [Ale96, PB04]. Another possibility is to overwrite the frame pointer address instead of the return address [klo99].

Heap-based buffer overflows are another class of code injection attacks. The memory heap is the dynamically allocated memory, in C managed by `malloc()` and `free()`, in C++ by `new()` and `delete()`. The heap consists of many memory blocks which are usually chained together by a double linked list. These memory blocks are managed by the dynamic memory allocator, which can insert, unlink or merge blocks together. The information (pointer to the previous and next block) of the linked lists is stored in a header for each block.

A heap-based buffer overflow may overwrite this header information in a way that one pointer of the double linked list points to the stack segment before the return address [Rag06]. If this block is now freed by the dynamic memory allocator, the header information of the previous and next block are updated. Because one pointer points to the stack segment due to the attack, the stack is updated in a way that the return address is overwritten with the address of a heap block, which can now be

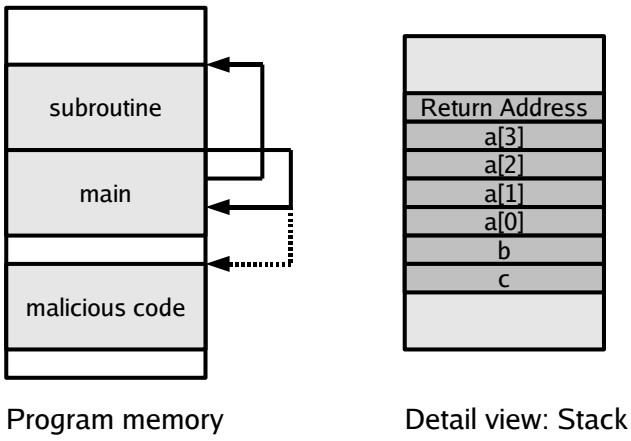


Figure 1.15: On the left side, a part of the program memory is shown. Normally, the subroutine is called and after its execution, the program counter jumps back to the main program after the call instruction. However, if the return address in the stack is overwritten by a buffer overflow of the vector `a[]` (see right side), the erroneous return destination may become the entry point of the malicious code (dashed line).

executed after the control flow reaches a return [Rag06, PB04]. There exist many other different possibilities to utilize heap-based buffer overflows [Con99, Dob03].

Arc injection or *return-into-libc* is an attack where a system function is exploited to spawn a new process which performs the attacker's desired operations. The name arc injection came from the inserting a new arc (control flow transfer) into the control flow graph (CFG) of a program. In the standard C library (*libc* on UNIX-based systems), there exists a function called `system()`. This function validates a process call given as argument and after successful validation starts its execution as a new thread. The memory location of this standard function is usually known, and therefore also the starting address to bypass the validation of the argument. The return pointer of the stack can now be manipulated by using a stack-based buffer overflow to jump to the desired destination in the system function to execute a malicious process. The name of the malicious process can be transferred to the system function by using registers [PB04]. This attack is useful if the architecture or operating system prevents the stack or heap memory area from execution.

Shacham generalized the *return-into-libc* attacks to show that it is possible to do malicious computation without injecting malicious code [Sha07]. The idea is that due to shared libraries, e.g., libc, many analyzable and attackers known instruction snippets are in the memory. Shacham proposes that an attacker can build an arbitrary

program from these snippets which can do arbitrary computation. This can be done by analyzing, for example, the *libc* library for code snippets which end with a *return* instruction. Moreover, Shacham shows that for the x86 architecture, it is possible to use only parts of instructions. The return instruction of the x86 architecture is a one byte instruction encoded with `0xc3`. However, other instructions which are longer consist also of this byte value. By starting the sequence in the middle of an instruction, the original instruction alignment is bypassed which enables the attacker the usage of additional new instruction sequences. From these building block, the attacker can build a program by chaining these snippets together by overwriting the register which stores the return address. This so-called *return-oriented programming* has been successfully transferred to other processor architectures, e.g., SPARC. In [BRSS08], a compiler is introduced which is able to construct return-oriented exploits from a general propose language. In summary, Shacham shows that preventing the injection of code is not sufficient for preventing malicious computation.

Pointer subterfuge is an attack technique where pointer values are changed. There exist four varieties: *function pointer clobbering*, *data pointer manipulation*, *exception handler hijacking* and *virtual pointer smashing* [PB04].

Function pointer clobbering modifies a function pointer so that the pointer directs to the malicious code. When the control flow reaches the modified function call, the attacker's function is called and his code is executed.

Data pointer modifications can be used for arbitrary memory writes. This technique can be combined with other code injection attacks to launch complex attacks.

Exception handler hijacking modifies the thread environment block (in MS Windows) that points to the list of registered exception handler functions. Because of the fact that the list is stored on the stack, the entries can be easily manipulated to utilize stack based buffer overflows. This technique can be put to work to transfer the control flow to a malicious function. Within Linux, function pointers in the *fnlist* can be replaced to have a similar effect.

Virtual pointer smashing replaces the virtual function table used in the C++ implementation of virtual functions. The virtual function table is used in C++ at runtime to implement dynamic dispatch. Every C++ object has a virtual pointer, which points to the appropriate virtual function table. By modifying the virtual pointer to direct to an attacker's virtual function table, malicious functions can be called when the next virtual function is invoked.

1.4.2 Invasive Physical Attacks

Invasive physical attacks physically tamper the embedded system with special equipment. Trivial physical attacks only compromise the availability of the system or damage a part or the whole system by physical destruction. Also, switching off the power supply voltage or cutting wires belongs to these trivial attacks.

Other invasive physical attacks aim to read out confidential data or the implementation of IP cores as well as the manipulation of the circuit or data to get access to sensitive data. These attacks have in common that expensive special equipment is used. The realization of these attacks requires days or weeks in specialized laboratories. The first step is the *de-packing* of the circuit chips. This is usually done with special acids [KK99, Hag].

After removing the packaging, the layout of the circuit can be discovered with optical microscopes and cameras. By removing each metal layer the complete layout of the chip can be captured in a map [KK99]. The gathered informations of the layer reconstruction can be used for *reverse engineering* the circuit, which gives competitors the possibility to optimize their product or to obtain more information about the implementation to launch other attacks.

Further information can be collected by *micro-probing* the circuit. This can be done by manual micro-probing, where metal probes have electrical contact to signal wires on chip. This is usually done on a *micro-probing workstation* with an optical microscope [KK99].

Due to the decreased lateral structure dimensions in todays circuit technologies, manual micro-probing is nearly impossible. But there exist advanced technologies, like *ion* or *electron beams*, as well as *infrared laser* which make micro-probing also possible in todays chip manufacturing technologies. With a *focused ion beam* (FIB) the chip structure can be scanned in a very high resolution. The beam hits the chip surface where electrons and ions are sputtered off and can be detected by a *mass spectrometer* [DMW98]. With increased beam intensity, the beam can also remove chip material with high resolution (see Figure 1.16). This can be used to cut signal wires or drill holes to reach signal wires in underlying layers. These holes can be filled with platinum to bring the signal to the surface, where it can be easily micro-probed. With an *electron beam tester*, the activity on the signal wires can be recorded, if the clock frequency is drastically reduced (under 100 kHz). Finally, with the *infrared laser*, the chip can be scanned from rear, because the silicon substrate is transparent in the infrared wavelength range. With the photon current, internal states of transistors or activity on signal wires can be read out [AK96, Ajl95].

These advanced technologies can be used to launch a variety of attacks. In focus are smart cards with implemented cryptographic algorithms. Most of the time, it is the attackers goal to read a secret key. One example is to read out the memory content using bus probing. The problem here is the generation of the successive addresses to get a linear memory trace. The attacker can bypass the software by destroying and deactivating the transistor gates which are responsible for branches and jumps with an FIB. The result is a program counter with can only linearly count up, which fits perfectly for this attack [KK99]. Other attacks are reading ROM, reviving and using test modes, ROM overwriting by using a laser cutter, EEPROM overwriting, key retrieval using gate destruction, memory remanence, or probing single bus bits, as well as EEPROM alternation [KK99, Hag].

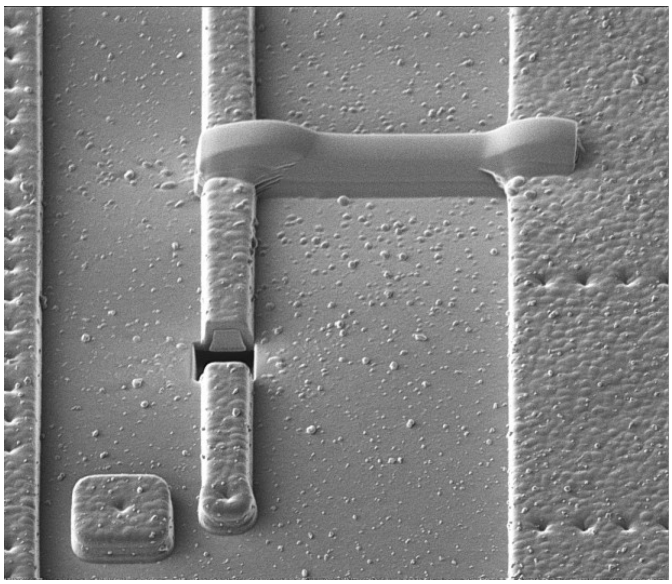


Figure 1.16: A secondary electron image recorded with a *focused ion beam* (FIB). The FIB previously interrupts a signal wire [Fra].

1.4.3 Non-Invasive Logical Attacks

To the non-invasive logical attacks belong the following attacks: *phishing*, *authenticity forging*, *attacking cryptographic weaknesses*, and *copying*. The goal of phishing is to gather sensitive information, like passwords, credit card numbers, or whole identities. By means of social engineering, such as fake web sites, emails, or instant massages to imitate a trustworthy person. The victim gives sensitive data away, believing that the attacker is not a harmful person. Phishing belongs to the authenticity forging attacks. Other authenticity forging attacks are DNS or certificate spoofing (manipulation). The difference to phishing is that systems and not persons are cheated.

Cryptographic attacks exploit weaknesses of cryptographic algorithms, e.g., *symmetric ciphers*, *asymmetric ciphers*, or *hashing algorithms* as well as *protocol stacks*. The goals of these attacks are access to sensitive data or to break into a system. More about cryptographic attacks can be found in [FS03] or [RRKH04].

Finally, copying attacks are attacks were sensitive data, like health data, personal data and works, which are protected by copyright, such as music, texts, programs, or IP cores, are copied without authorization. These attacks, especially the gathering of sensitive data and copyright infringement, target the security goal confidentiality.

1.4.4 Non-Invasive Physical Attacks

Eavesdropping and *side-channel attacks* belong to the class of non-invasive physical attacks which normally do not impair the system. Eavesdropping is the interception

of conversations or data transmissions by unintended recipients. The attacker can gather sensitive information which is transmitted using electric media, e.g., email, instant messenger, or telephone. Sometimes a combination of eavesdropping and cryptographic weakness attacks are used to monitor sensitive data. For example, sniffing passwords in a *WEP* (Wired Equivalent Privacy) encrypted *WLAN* (Wireless Local Area Network).

Information of cryptographic operations in embedded systems can be gathered by side-channel attacks. Usually, the goal is to get the secret key or information about the implementation of the cryptographic algorithm. Cryptographic embedded systems are particularly vulnerable to these attacks, because the attacker has full control over the power and clock supply lines. The different side-channel attacks are *timing analysis*, *power analysis*, *electromagnetic analysis*, *fault analysis*, and *glitch analysis* [Hag, KLMR04, RRKH04].

Timing analysis attacks are based on the correlation of output data timing behavior and internal data values. Kocher [Koc96] showed that it is possible to determine secret keys by analyzing small variations in the execution time of cryptographic computations. Different key bit values cause different execution time, which makes a read out and reconstruction of the key possible.

Power analysis attacks are based on the fact that different instructions cause variations in the activities on the signal lines, which result in different power consumption. The power consumption can be easily measured by observing the current or the voltage on a shunt resistor. With *simple power analysis* (SPA) [KJJ99], the measured power consumption is directly interpreted to the different operations in a cryptographic algorithm. With this technique, program parts in a microprocessor, for example DES rounds or RSA operations, can be identified. Because of the execution of these program parts depend on a key bit, the key bits can be read out. *Differential power analysis* (DPA) [KJJ99] is an enhanced method which uses statistical analysis, error correction and correlation techniques to extract exact information about the secret key.

Similar to power analysis techniques, information about the key, data, or the cryptographic algorithm or implementation can be extracted by electromagnetic radiation analysis [AARR03].

During different fault analysis (DFA) attacks, the system is exposed to harsh environment conditions, like heat, vibrations, pressure, or radiation, to enforce faults which result in an erroneous output. Comparing this output to the correct one, the analyst gains insight into the implementation of the cryptographic algorithms as well as the secret key. With DFA attacks, it was possible to extract the key from a DES implementation [BS97] as well as public key algorithm implementations [BDH$^+$98, BS97]. The last one shows that a single bit fault can cause fatal leakage of information which can be used to extract the key. Pellegrini and others show that using fault analysis, where the supply voltage of an processor is lowered, it is possible to reconstruct several bits from a secret key of the RSA cryptographic algorithm [PBA10]. For this

attack, they used the RSA implementation of the common OpenSSL cryptographic library and a SPARC Leon3 core, implemented on a Xilinx Virtex-II Pro FPGA. The supply voltage of the FPGA is lowered till sporadic bit errors occur on the calculation of the signature using the FPGA's hardcore multiplier.

Glitch attacks also belong to the class of DFA attacks. Here, additional glitches are inserted into the clock signal to prevent the registering of signal values on the critical path. The simplest attack is to increase the clock frequency. One goal of glitch attacks can be the prevention of branches in a microprocessor program, because of the calculation and registering of the new branch target address is a long combinatorial path on many processor implementations [KK99].

1.5 Contributions

It has been shown in the previous sections of this chapter that security and reliability in embedded systems are very important topics, and that their significance is likely to increase in the future. Additionally, this and the next chapter show that the topic involves a wide research area with a large number of annual publications. After this introduction, major contributions of this thesis with respect to these topics are outlined, concentrating on two major concerns, namely:

- ***Identification and Watermarking of IP Cores*** and
- ***Control Flow Checking***.

The first topic is mainly about security issues, and is concerned with countermeasures against unauthorized copying of IP cores, which compromises the confidentiality of the core's author. The second topic mainly treats reliability issues, but also security issues. In particular, soft errors (see Section 1.3.4), but also some degeneration faults (see Section 1.3.1) and code injection attacks (see Section 1.4.1) are discussed. Physical attacks, which affect the control flow of a program running on an embedded system (see Section 1.4.2 invasive physical attacks and Section 1.4.4 differential fault analysis), are focused on as well. The key scientific contributions of this dissertation are summarized as follows:

- **Watermarking and Identification Techniques for FPGA IP Cores:** Unlike most existing watermarking techniques, the focus of our techniques lies on ease of verification, even if the protected cores are embedded into a product. Moreover, we have concentrated on higher abstraction levels for embedding the watermark, particularly at the logic level, where IP cores are distributed as netlist cores. With the presented watermarking methods, it is possible to watermark IP cores at the logic level and identify them with a high likelihood and in a reproducible way in a purchased product from a company that is suspected

to have committed IP fraud. To the best of our knowledge, this is only possible for netlist cores using our watermarking techniques. Moreover, approaches to identify an embedded core by looking at special core properties without using a watermark are presented as well [ZAT06]. Using these techniques, we cover the protection of IP cores on all different abstraction levels from RTL to device level without restrictions on the verifiability in a given product.

The investigated techniques establish the authorship by verification of either an FPGA bitfile (see also [ZAT06, SZT08]) or the power consumption of a given FPGA (see also [ZT06, ZT08b, ZBT10b, ZT07a, ZT07b]). Both the FPGA bitfile and the measurement of the power consumption can be easily obtained from a given product. However, if the product vendor uses encrypted bitfiles, only the watermark verification using the power consumption is possible with the proposed methods. On the other hand, if the FPGA embeds multiple watermarked cores, the different power watermarking signals superpose which makes extraction of watermarks harder or even impossible. To circumvent this obstacle, we propose multiplexing methods [ZBT10a]. Furthermore, we would like to remark that the proposed *power watermarking* technique, which is the only watermark verification method for netlist cores and the first one available for bitfile cores which works with bitfile encryption. Another watermarking technique for encrypted bitfiles, introduced in [KMM08], uses thermal dissipation for verification and is commercially available as the product *DesignTag* from *Algotonix*. The commercialization underlines that there is a real demand for IP protection by such watermarking techniques.

In summary, the key contributions of this part of the dissertation are:

- Extended theoretical watermarking and identification models
- Non-invasive verification using solely the given product
- Watermarking and identification of IP cores published as HDL, netlist, and bitfile cores
- Identification and watermarking techniques using the bitfile for verification
- Novel watermark verification techniques based on the exploration of power consumption

- **Control Flow Checking for Embedded RISC Processors and General IP Cores:** The goal of control flow checking is to recognize, analyze, and correct sporadic and/or permanent errors that occur in the control path of embedded CPUs or IP cores. Our vision is to define autonomously behaving elements [SBE+07b, SBE+07a] that resolve functional errors of the control paths locally

inside the core at runtime and prevent false instructions or transitions from being executed, which could lead to an erroneous state (error-resilience). The corresponding autonomous elements are called *control flow checkers*. They are supposed to recognize, evaluate, and even correct errors during runtime which affects the control flow of an embedded CPU or IP core (see also [ZT08a, ZT09]).

We present techniques for fast error detection inside embedded processors, which is important because errors should be detected even before they can manifest into registers or memories. This is achieved by monitoring the control flow in the first pipeline stages of the processors which makes it possible to correct errors by a simple re-execution of the erroneous instructions without time and area expensive restoration of registers and memory states. This allows us to build a lightweight checker unit with very low area and only moderate memory overhead.

Proposed is a unique control flow checking framework based on a modular approach. Modularity allows to add or remove features in order to reduce the overhead, to increase the error coverage, or add error correction measures. Using our control flow checking approach, the developer of a core is able to decide about trade offs between different features and their respective overheads.

Moreover, we point out the possibility of *fast error correction* and the integration of these techniques with the methods for *data path protection*. This combination with data path protection scheme introduced in [BZS+06] enables an embedded RISC processor to detect disturbances, originating from the underlying semiconductor technology as, for example, single event effects, and to react accordingly. Both aspects, the fast error correction and the combination with data path protection, are new and go beyond the state-of-the-art of existing control flow checking techniques. Other benefits of this approach are that it is not necessary to change the application code or insert any additional instructions. Therefore, there is no performance impact on error-free execution. The approach can be easily integrated into the software development flow and thus, the presented technique is completely transparent to the program developer.

This work provides new control flow methods mainly for embedded RISC-CPUs, which are also applicable for general IP cores. The focus in this context is put on methods that can be easily integrated into ASIC designs. Naturally, these methods can also be used in FPGA designs, however the necessary reliable underlying FPGA structure is not part of this work.

In summary, the key contributions of this part of the dissertation are:

- New methods for lightweight control flow checking
- Detection of all errors which affect the control flow

- Control flow checking techniques for embedded RISC CPUs and general IP cores using programmable hardware checker units
- Modular approach with support for fast error detection and correction and low hardware overhead
- Support of direct and indirect control flow instructions
- Integration with methods for data path protection for embedded CPUs
- Prototypical implementation for the SPARC Leon3 processor

Both topics address problems which have a high significance for embedded systems today and the importance tends to rise in the future with the ongoing technology improvements and the integration of more and more cores to very complex systems-on-a-chip. In particular, IP core watermarking addresses the issue of copyright protection in an increasing market of IP cores, whilst control flow checking addresses problems including soft errors, even in combinatorial logic in with the occurrence of soft errors is a new challenge.

On the other hand, hardware overhead and easy integration into processor design and application development flows are of utmost importance in many cost-sensitive systems, particularly in embedded systems. Both methodologies, watermarking and control flow checking, have a relatively low hardware overhead and are easy to integrate into current design flows. From the design flow point of view, this is an evolution, not a revolution, which means that today's design tools can be used furtheron and without compromise.

Finally, this work provides case studies for FPGA implementations and experimental evaluations. Besides the demonstration of feasibility of the proposed methods, these case studies indicate the amount of overhead for an FPGA implementation.

1.5.1 Overview of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2 gives an overview of related work for security and reliability in embedded systems. The focus lies on methods for watermarking and IP protection of IP cores, defenses against code injection attacks, measures against faults and errors, as well as methods for control flow checking in embedded CPUs and general RISC cores.

Chapter 3 provides new methods for IP core watermarking and identification. After introducing a novel theoretical model for watermarking and the threats to it, methods for bitfile and power watermarking are presented. The bitfile watermarking

section presents methods using the lookup table content to identify the core (also presented in [ZAT06]), as well as an implementation of watermarking using functional lookup tables (also presented in [SZT08]). The next section treats power watermarking. First, the model of the communication channel is introduced and then the basic power watermarking methods. After that, several new methods which increase the decoding robustness are presented (partly also presented in [ZT06], [ZT08b], [ZBT10b], [ZBT10a], [ZT07a], and [ZT07b]). Finally, experimental results for the bitfile and power watermarking are given.

Chapter 4 deals with techniques for control flow checking. The focus lies on methods checking the correctness of control flow instructions issued during the execution of programs for embedded RISC CPUs. Using the suggested methodology, any error of illegal or faulty direct or indirect jump and branch instruction can be detected for a given program code at runtime. Moreover, techniques for checking the control flow of general IP cores are presented. Furthermore, hardware concepts for a generic modular control flow checking framework which may be tightly attached to a given CPU are proposed. Finally, a prototypical implementation for the SPARC Leon3 processor and a case study which implements a turbo encoding/decoding system, also presented in [MWB+10], are shown. Most of the control flow checking methodology for embedded RISC processors is also presented in [ZT08a] and [ZT09].

Chapter 5 concludes the work and gives an outlook for future work.

2 Related Work

This chapter provides an overview of related work on security and reliability of embedded systems. Because of this is a very vast topic, we concentrate on issues related to *IP core protection* as well as *control flow checking* for embedded CPUs and IP cores.

The chapter is structured as follows: First, methods which increase the security of embedded systems are discussed. This includes methods for IP protection related to IP core watermarking and identification as well as measures against code injection attacks, related to control flow checking. Second, methods which increase the reliability of embedded systems are discussed. In focus are measures against single event effects and degeneration faults. Finally, approaches for control flow checking are presented.

2.1 Security: IP Protection

Intellectual property (IP) denotes the absolute right on an *intangible asset*, like music, literature, artistic works, discoveries, inventions, words, phrases, symbols, designs, software, or IP cores. The owner of the IP can license his work to other people or companies. IPs are protected by law with patents, copyrights, trademarks, and industrial design rights.

Drimer defines the following protection or defense categories against IP theft or fraud [Dri09]: *social, reactive*, and *active protections*.

Social protection means that IP works are protected by laws, non-disclosure agreements, copyrights, trademarks, patents, contracts, and so on. The deterrents are con-

viction by a court of law and the loss of a good reputation. However, these deterrents are only effective if the misconduct can be proven and the appropriate laws exist. Furthermore, the laws must be enforced which is handled differently from country to country.

Reactive protection means that the theft or fraud cannot be prevented, however, it can be detected and delivers evidence of the misconduct. Some reactive protection mechanisms deliver only suspicious facts which, however, may be enough to trigger further investigations. Furthermore, the persistence of reactive protection mechanisms might deter would-be attackers.

Active protection means that physical or cryptographic mechanisms prevent the theft or fraudulent usage of the protected work. This category has the highest deterrent degree. However, these mechanisms can be broken by attacks. Often the attack can be proven if the misconduct is detected.

In this work, we concentrate on the protection of the IP of hardware cores. These so called *IP cores* are distributed like software and can easily be copied. Some core suppliers encrypt their cores and deliver special development tools which can handle encrypted cores. The disadvantage is that common tools cannot handle encrypted cores and that the shipped tools can be cracked so that unlicensed cores can be processed. Another approach is to hide a signature in the core, a so-called *watermark*, which can be used as a reactive proof of the original ownership. There exist many concepts and approaches on the issue of integrating a watermark into a core.

In general, hiding a unique signature into user data, such as pictures, video, audio, text, program code, or IP cores is called *watermarking*. Embedding a watermark into multimedia data is achieved by altering the data slightly at points where human sense organs have lower perception sensitivity. For example, one can remove frequencies which cannot be perceived by the human ear by coding an audio sequence into an MP3 file. Now, it is possible to hide a signature into these frequencies without decreasing quality of the coded audio sequence [BTH96].

One problem of watermarking is that for verification, the existence and the characteristic of a watermark must be disclosed, which enables possible attackers to remove the watermark. To overcome this obstacle, Adelsbach and others [ARS04] and Li and others [LC06] presented so-called *zero-knowledge watermark* schemes which enable the detection of the watermark without disclosing relevant information.

The watermarking of IP cores is different from multimedia watermarking, because the user data, which represents the circuit, must not be altered since functional correctness must be preserved. A *fingerprint* denotes a watermark which is varied for individual copies of a core. This technique can be used to identify individual authorized users. In case of an unauthorized copy, the user, the copied source belongs to, can be detected and the copyright infringement may be reconstructed. Watermarking procedures can be categorized into two groups of methods: *additive methods* and *constraint-based methods*.

In additive methods, the signature is added to the functional core, for example, by using unused lookup-tables in an FPGA [LMSP98]. The constraint-based methods were originally introduced by [KLMS+01] and restrict the solution space of an optimization algorithm by setting additional constraints which are used to encode the signature.

A survey and analysis of watermarking techniques in the context of IP cores is provided by Abdel-Hamid and others [AHTA04]. Further, we refer to our own survey of watermarking techniques for FPGA designs [ZT05]. A survey of security topics for FPGAs is given by Drimer [Dri09] who also maintains the *FPGA design security bibliography* website: `http://www.cl.cam.ac.uk/~sd410/fpgasec/`.

In order to compare different watermarking strategies, some criteria are defined in the following [HP99]:

Functional correctness: This is the most important criteria. If the watermark process destroys the functional correctness, it is useless to distribute the core.

Resource overhead: Many watermarking techniques need some extra resources. Some to generate and store the watermark itself, some because of the degradation of the optimization results from the design tools. The ratio between the original and the watermarked core's resource demand is defined as the resource overhead.

Transparency: The watermark procedure should be transparent to the design tools. It should be easy to integrate the watermarking step into the design flow, without altering common design tools.

Verifiability: The watermark should be embedded in such a way that the authorship can be verified easily. It should be possible to read out the watermark only with the given product and without any further information from the design flow which must be requested from a company suspected of IP fraud.

Difficulty of removal: The watermark should be resistant against removal. The effort to remove the watermark should be greater than the effort needed to develop a new core, or the removal of the watermark should cause corruptness of the functionality of the core. Watermarks which are embedded into the function of the core are in general more robust against removal than additive watermarks.

Strong proof of authorship: The watermark should identify the author with a strong proof. It should be impossible that other persons can claim the ownership of the core. The watermarking procedure must be resistant against tampering.

In this section, we first discuss IP protection methods using core encryption. After that, related work using additive and constraint based watermarking methods is presented.

2.1.1 Encryption of IP Cores

The goals of active IP protection for cores are, first, that the core cannot be used without a proper license and, second, that the core is protected from unauthorized modifications. The cores can be delivered in encrypted form and are decrypted by design tools. Other approaches for FPGAs use an encrypted configuration bitfile which is decrypted on the FPGA.

Encrypted HDL or Netlist Cores

One solution is to deliver encrypted IP cores to the customers and integrate de- and encryption functions into the EDA tools. The customer buys the encrypted core and obtains the appropriate key from the IP core developer or vendor. This technique is applicable for IP cores of all *abstraction* or *technology levels* (RTL – HDL cores, logic level – netlist cores, device level – bitfile/layout cores). However, if, e.g., an HDL core should be protected at all abstraction levels, the synthesis tool must produce an encrypted netlist. This must be done for all steps: decryption of the core, processing, and encryption. It is important that the customer only has access to the encrypted data, which means that the EDA tool routines must be protected against read out attacks.

The problem is that no consistent industrial standard exists which handles encrypted IP cores [Dau06]. This complicates the interoperability of IP cores and EDA tools.

Today, *symmetric* and *asymmetric cryptographic approaches* are used. Using symmetric cryptographic approaches, the en- and decryption is done with the same key. The advantage of this approach is the reduced computational complexity compared to asymmetric approaches. One problem is the secure distribution and communication of the key. Furthermore, EDA tools must deal with different keys for different IP vendors, and if one key is cracked, usually all IP cores of the corresponding vendor have lost their protection. Nevertheless, this approach is used, for example, by Xilinx to encrypt some of their parameterizable HDL IP cores, e.g., the Microblaze processor softcore [Xild].

Methods using asymmetric cryptography are also known as *public key cryptography* which need two keys, the *private* and the *public key*. The private key is for decryption inside the EDA tools, where as the encryption key is publicly available and is used by the IP core vendor. The EDA vendor creates the key pairs and embeds the private key in his tools. The IP core developer can now use the public key for the encryption. The advantage is that the private decryption key may not be transferred

over untrusted communication channels and is only known by the EDA vendor. The disadvantage is that asymmetric approaches have a high computational complexity which results in long runtime for decryption up to several hours for IP cores [Dau06]. Another drawback is that the IP vendor must create a separate version for each EDA tool, which is encrypted with the corresponding public key of the EDA vendor.

Dauman, Vice President of the *Synopsys' Synplicity Business Group*, introduced a hybrid approach [Dau06]. The IP core is encrypted with a symmetric cryptographic method, like *Triple-DES*, or *AES* using a key which is generated by the IP vendor. This key, now referenced as the *data key*, is encrypted with an asymmetric cryptographic method, like *RSA* [RSA78], with the public key of the EDA vendor. This approach is similar to the *PGP* approach [Zim95] for cryptographic privacy and authentication of messages. The decryption is done with the (decrypted) data key and the cryptographic method which is specified by the IP vendor. Inside the EDA tools, there exist different symmetric cryptographic routines for the decryption of the core. The advantage is that the decryption with a symmetric algorithm is very fast and the computational complex asymmetric method is only used for the data key which is very small compared to the whole IP core. Synplicity suggested this approach as future industry standard and includes this method called *ReadyIP* into the product *Synplify Premier* [Syna].

In 2007, a industry-wide panel discussion [Wil07] provided some insight into the perception of encrypted IP cores of the EDA industry. The conclusion was that the current social-based protection works well for large cooperations. A better solution is desirable but not necessarily urgent. However, they express their reservation to small companies or startups which are not known in the community and might not be willing to sell IP cores to these companies.

Barrick argued against the usage of encrypted netlist cores due to their hidden costs [Bar]. The disadvantages are the fixed constraints, the prevention to reuse parts of the logic for other cores, slower simulation speed or inaccurate behavioral models, restriction of the choice of EDA tools, and fewer debugging possibilities. However, sometimes encryption can be worth due to reduced acquisition costs.

Encrypted FPGA Configurations

Another kind of IP protection is the encryption of the FPGA *configuration* or *bitfile*. The bitfile is stored in a *non-volatile memory*, e.g., a PROM, and transferred to the FPGA encrypted. Inside the FPGA during the configuration, the bitfile is decrypted. This approach prevents copy attacks for bitfile designs and protects the bitfile from reverse engineering.

The first suggestion of this method was in 1995 by a patent from Austin [Aus95]. The first FPGA devices which offered configuration encryption was the Actel's 60RS family. However, all FPGAs had the same permanent key, which prevents no copy attacks. Furthermore, the key was also stored in the software. Consequently, it was easy

for attackers to extract the key from the software. Xilinx introduced configuration decryption with a Triple-DES hardcore for Virtex-II devices in the year 2000. The user defined key can be stored and updated into an FPGA internal battery-backed SRAM. Today, bitfile encryption is supported by many high-end FPGA families. Some FPGA devices, such as the Altera Stratix II/III, can be configured to always perform decryption. This prevents the configuration with bitfiles which are not encrypted with the proper key.

There exist two different key storing techniques: *volatile* and *non-volatile*. Volatile key storing uses low power SRAMs which are powered by an external battery. Attackers must keep powering the key storage during the attack, which is more complicated. On an attacker's error, the key is cleared and the bitfile cannot be loaded. The disadvantage is the increased printed circuit board space and costs for the external battery. Non-volatile key storage uses fuses, flash, or EEPROMs. The problem is that these technologies must be combined with the latest CMOS technology on the same chip, which affords in a non-standard manufacturing step. The results are increasing costs and more complex verification strategies.

An important aspect of methods using encrypted FPGA configuration bitfiles is the *key management* which includes the generation and the distribution of keys. Kean suggests a method where the FPGA can encrypt and decrypt bitfiles with hardware cores and a permanent embedded key [Kea01]. The FPGA is able to encrypt the bitfile on the first programming and store this encrypted bitfile in a non-volatile memory. Upon every FPGA configuration during the power-up cycle, the bitfile is loaded and decrypted in the FPGA. The advantage is that the key never leaves the FPGA.

Bosset and others [BGB06] propose a method for using partial reconfiguration for en- and decryption of FPGA bitfiles by user-defined soft cores. At power-up, the decryption core is initially loaded form the PROM which decrypts the bitfile with the user logic. Soudan and others [SAH] propose a method for the encryption of partially reconfigurable bitfiles using device-specific keys.

2.1.2 Additive Watermarking of IP Cores

Additive methods are watermarking procedures, where a signature is added to the core. This means that the watermark is not embedded into the function of the core. Nevertheless, the watermark can be masked, so it appears to be part of the functional part. Additive watermarks can be embedded into HDL, netlist, bitfile or layout cores.

HDL Cores

Additive watermarking for HDL cores seems to be very complicated, because of the human-readable structure of the HDL code. Hiding a watermark there is very difficult, because on the one hand, an attacker may easily detect the watermark, and

on the other hand, subsequently used design tools might remove the watermark during circuit optimization. However, it is not impossible to include an additive HDL component into the core, which may not removed by the design tools.

Castillo and others hide a signature into unused space of dedicated lookup table based memory [CPG+06]. To extract the signature, an additional logic monitors the input stream for a special *signature extraction sequence*. If this sequence is detected, the signature is sent to the outputs of the core. This approach was later generalized for other memory structures in [CPG+08]. The drawback is that distribution as an HDL core is not possible, because the signature extracting logic is easy to detect and to remove.

Oliveira presents a general method for watermarking *finite state machines* (FSMs) in a way that on occurrence of a certain input sequence, a specific property exhibits [Oli01]. The certain input sequence corresponds to the signature which is previously processed by cryptographic functions. A similar approach is presented by Torunoglu and others in [TC00] which explores unused transitions.

Fan provides a method where the watermark or signature is sent as a preamble of the output of the test mode [FT03]. Some ASIC circuits provide a special test mode which stimulates the core with special input patterns. To analyze the correctness of the core, the output of these input patterns are measured and compared to the correct patterns. The idea is to send the watermark sequence over the output port before the test sequence starts.

The disadvantage of these approaches is the usage of ports for signature verification. This works only if the ports are reachable. If the core is embedded into other cores, the ports of the watermarked core can be altered which falsifies or prevents the detection of the signature in the output stream. This applies also to the signature extraction sequence in the input stream.

Netlist Cores

To the best of our knowledge, there exist no publications on the use of additive watermarking at the level of netlist cores. In [ZT05] we presented the first two examples of how additional watermarking for netlist cores can look like. The first idea is to apply redundant logic in some paths of the core according to a signature. To verify the watermark, one can optimize the core so that the redundant logic is removed, show the differences and reconstruct the signature.

The second idea is to add false paths in the design which do not affect the following logic. The weakness of both ideas is that the design tools applied in subsequent steps use transformations which may destroy the watermark. Therefore, these ideas are not applicable.

Bitfile Cores

The approach of Lach and others watermarks bitfile cores by encoding the signature into *unused lookup tables* [LMSP98]. At first, the signature will be hashed and coded with an error correction code (ECC) to be able to reconstruct the signature even if some lookup tables are lost, e.g., during tampering. After the initial place and route pass, the number of unused lookup tables will be determined. The signature is split into the size of the lookup tables and additional LUTs are added to the design. Then, the place and route process will be started again with the watermarked design. Later, the approach was improved by using many small watermarks instead of a single large one [LMSP99]. The size of the watermarks should be limited by the size of a lookup table. The advantage is that small watermarks are easier to search for, and for verification, only a part of all of watermark positions must be published. With the knowledge of the published position, the watermark can be easily removed by an attacker. At the verification process, only a few positions of the watermark need to be used to establish the ownership. A second improvement is that a *fingerprinting technology* is added to the approach that enables the owner to see which customer has given the core away [LMSP01]. The fingerprinting technology is achieved by dividing the FPGA into tiles. In each tile, one lookup table is reserved for the watermark. The position of the mark in the tile encodes the fingerprint. For verification, it is possible to read out the content of the lookup table from a bitfile. So, these methods are easy to verify. It's more difficult to determine the position of the watermark in a tile, but it's still generally possible. However, if an attacker knows the position of the watermark, it is easy to overwrite it.

Saha and others present a watermarking strategy for FPGA bitfiles by subdividing the lookup table locations into sets of 2×2 tiles [SSK07]. The number of used lookup tables in a set is used as signature. From an initial level, additional lookup tables are added to achieve the fill level according to the signature. The input and output are connected to the *don't care inputs* of the neighboring cells. Kahng and others show in [KLMS$^+$98] that the configuration of the multiplexer of unused CLB outputs in FPGA bitfiles can carry a signature. The signature is embedded after the bitfile creation and by knowing the encoding of the bitfile. These configuration bits can be later extracted to verify the signature.

Van Le and Desmedt show that these additional watermark schemes for bitfile cores can be easily attacked by reverse engineering, watermark localization, and subsequent watermark removal [LD03]. A simple algorithm is introduced which identifies lookup tables or multiplexers whose outputs are not connected to any output pins. However, these attacks are only successful if reverse engineering of the bitfile is possible and the costs of reverse engineering are not too high. More about attacking costs is described in Section 3.2.

Finally, Kean and others present a watermarking strategy where a signature is embedded into an FPGA bitfile core or design [KMM08]. The read out of the signature

is done by measuring the temperature of the FPGA. This approach is commercially available as the product *DesignTag* from *Algotronix*.

2.1.3 Constraint-Based Watermarking of IP Cores

All optimization problems have constraints which must be satisfied to achieve a valid solution. Solutions which satisfy this constraints are the *solution space*. Constraint-based watermarking techniques represent a signature as a set of *additional constraints* which are applied to the hardware optimization and synthesis problem. These additional constraints reduce the solution space (see Figure 2.1) since the chosen solution must also satisfy the additional constraints. The same solution could be achieved with neglecting these additional constraints with probability P_c. The probability P_c of this event is given by the following formula:

$$P_c = \frac{n_w}{n_o} \tag{2.1}$$

where n_o is the number of solutions which satisfy the original constraints and n_w is the number of solutions which satisfy both the original and the additional constraints [KLMS+01, KHPC98]. If P_c is very small, a solution that also satisfies the additional (watermarking) constraints is a *strong proof* of the existence of the watermark.

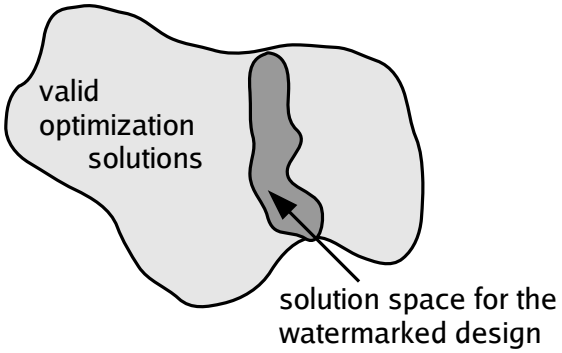


Figure 2.1: The solution space of an original and a watermarked design. If a design satisfies the original and the additional constraints, then the design is protected by a watermark. The probability that the additional constraints are satisfied by chance should be low to have a strong proof of authorship.

Qu proposes a methodology to make a part of the watermark – for constraint-based watermarking, some additional constraints – public which should deter attackers [Qu02]. The other parts, called *private watermark*, are only known by the core author and are used to verify the authorship in case that the public watermark was attacked. A similar approach is used by Qu and others to generate different fingerprints

by dividing the additional constraints into two parts [QP00]: The first part is a set of relaxed constraints which denote the watermark. By applying distinct constraints to the second part, different independent solutions can be generated which may be used as diverse fingerprinted designs.

Charbon proposed a technique to embed watermarks on different abstraction levels which he called *hierarchical watermarking* [Cha98]. The idea is, if an attacker is able to remove a watermark, for example, embedded into the layout of a circuit, the watermarks added at higher abstraction levels are still present. However, Charbon focused more on *layout, nets*, and *latch watermarking techniques* which are only applicable for ASIC layout cores.

The verification of a constraint-based watermark is usually done with the watermarked core as it is. This means the watermarked core can be purchased or published and from the distributed cores the watermark can be verified. However, if the core is combined with other cores and traverses further design steps, the watermark information is usually lost or it cannot be extracted.

Van Le and Desmedt [LD03] present an ambiguous attack (see Section 3.2) for constraint-based watermarking techniques. The authors add further constraints to the watermarked solution by allowing only a minimal increase of the overhead. The result is a slightly degenerated solution which satisfies many additional constraints. This means that in this solution, a lot of different signatures can be found which destroys the unique identification of the core developer. They choose, for example, the constraint-based watermarking approach for *graph coloring*. Further, this attack might be applicable to other constraint-based watermarking techniques.

As it was the case with additive watermarking strategies, constraint-based watermarking strategies are applicable for HDL, netlist, and bitfile cores.

HDL Cores

HDL code is usually produced by human developers or high-level synthesis tools. Both can set additional constraints to watermark a design. One approach is to use a watermarked *scan chain* [KP98]. Scan chains are usually used in ASIC designs to access the internal registers for debugging purposes. The use of scan chains in FPGA designs is rather unusual, but might be helpful in some cases. At first, a number will be assigned to each register, and the registers will be sorted. Now, a *pseudo random sequence* will be generated from the signature (one example is given in Section 3.2.2). Registers are selected with an algorithm which uses a random sequence as input. For K_c scan chains, the first K_c selected registers are chosen as the first register in each chain. Depending on the signature, we have a variation on the scan chains which can be used to detect the watermark. It is possible that an unfortunately chosen start of a chain could result in the allocation of more routing resources. Moreover, the maximum clock frequency for the scan chain can be limited. This approach is easy to verify, if the scan chains can be accessed from outside of the chip. Problems occur,

if the scan chain is only used internally or is not connected to any device. In such a case, there is no verification possibility.

Some work was done for watermarking *digital signal processing* (DSP) functions [RAMSP99, CD00]. This kind of watermarking has more in common with media watermarking instead if IP watermarking. Both approaches alter the function of the core slightly by embedding a watermark. In [RAMSP99], the coefficients of *finite impulse response* (FIR) filters are slightly varied according to the watermark. Additionally, the authors use different structures to build the FIR filter which also corresponds to the signature. In [CD00], these ideas are extended and proven correct by mathematical analysis.

Netlist Cores

An approach to watermark netlist cores is to preserve certain nets during *synthesis* and *mapping* [KHPC98]. Synthesis tools merge signals or nets together and produce new nets. Only a few nets from the synthesis input will be visible in the synthesis result. The technology mapping tool also eliminates nets by assembling gates together in a lookup table. Kirovski's approach enumerates and sorts all nets in a design. The first K_c (see previous section) nets of the input are chosen by the synthesis tools according to a signature. These nets will be prevented from elimination by the design tools by connecting these nets to a temporary output of the core. The new outputs from additional constraints for the synthesis tool, and the corresponding result is related to the watermark. A disadvantage is that it is easy to remove the additional logic. If the content of the lookup table is synthesized again, the watermark will be removed.

Meguerdichan and others presented a similar approach for netlist cores where additional constraints are added during the *technology mapping step* of the synthesis process [MP00]. In this approach, critical signals are not altered which preserves the timing and the performance of the core. The signature is encoded into the number of allowed inputs of a certain primitive cell, e.g., a gate or a lookup table. The primitive cells which are not in the critical path are enumerated, and according to the signature, the number of usable inputs are constrained.

Khan and others watermark netlist cores by doing a *rewiring* after synthesis [KT05]. Rewiring means that redundant connections between primitive cells are added in the netlist which makes other original connections redundant. These new redundant connections are removed.

Bai and others introduce a method for watermarking transistor netlists for *full custom designs* [BGXC07]. The transistors are enumerated and sorted into a list like in the approach above. Corresponding to the pseudo random stream generated from the signature, the width of the transistor gate is altered. If the transistor is assigned a '1' from the random stream, the transistor width is increased by a constant value.

Bitfile and Layout Cores

Additional placement, routing, or timing constraints can be added to watermark bitfile cores. To embed a watermark with placement constraints, Kahng and others place the *configurable logic blocks* (CLBs) in even or odd rows depending on the signature [KMM+98]. In this approach, the signature is transformed into even/odd row placement constraints. The placed core will be tested on preserving the constraints and, if necessary, CLBs are swapped. This method has no logical resource overhead and the additional costs of routing the resources are very small or tend to zero, because the placement is altered only marginally. The problem of verification is to extract the CLB placement information. Only if knowing how the CLBs correspond to the signature, the watermark can be verified. A strategy to achieve this is to uniquely enumerate the CLBs in an FPGA from the top left corner.

Kahng and others [KMM+98] propose a second approach by adding constraints to the router. The constraints achieve that a net selected by the signature is routed with some additional, unusual routing resources. These unusual resources can be, for example, *wrong way* segments. A wrong way segment is a segment in which the net goes to the wrong direction and then back in the right direction to form a backstrap. The authors claim that this is unlikely for a normal router, and so such a net can be verified as a watermarked net.

Furthermore, additional timing constraints can be used to watermark a core. Timing constraints limit the route and logic delay between two registers. Kahng and others propose a technique to select paths which have timing constraints according to the signature. The timing constraints for these paths are split into two separate constraints. For example, let a path have six logic gates and a timing constraint of 10 *ns*. The new constraint is 4 *ns* for the first 3 cells and 6 *ns* for the rest [KLMS+01].

Another approach by Jain and others measures the delay on selected paths and adds new timing constraints on these paths [JYPQ03]. The new constraints are chosen based on the measured delay by setting the last digit to a value of a bit from the signature. For example, let a path have a delay of 5.73 *ns*. If the coded bit is a '1', the new constraint for this path is 5.71 *ns*, if the coded bit is '0', the constraint is 5.70 *ns*.

Narayan and others present a watermarking approach of the layout by modifying the number of *vias* and *bends* of certain nets [NNC+01]. Like in other approaches, the nets are enumerated, and additional vias or bends are inserted according to the signature.

Saha and others present a watermarking scheme by altering the size of the repeaters according to the signature [SSK07]. In high performance ASIC designs, *repeaters* (a buffer for amplification of the signal) are inserted into critical nets to decrease the delay.

2.1.4 Other Approaches

Many other approaches exist for protecting IP cores or designs from unlicensed usage or alteration. For example, the *VHDL Obfuscator & Watermarker* [VIS] is able to obfuscate VHDL cores in a way that the algorithm is hidden, but leaves the core synthesizeable. This approach make reverse engineering and alteration of the core much harder. Further, by using different scrambling techniques, a watermark can be embedded in the obfuscated code. Clearly, this watermark can only be detected at the RTL in the HDL core and is lost once the core is synthesized.

Other approaches prevent the copying of bitfile designs by using *unique FPGA* or *board identification*. If the obtained bitfile is programmed on another board, the function will not work. The unique identification can be done by using a *non-volatile external device*, a *unique key* embedded into the FPGA, or by *physical unclonable functions* (PUFs) [Dri09].

Kessner uses a non-volatile CPLD for board identification [Kes00]. A bit sequence is calculated by a cryptographic algorithm implemented on the FPGA and additionally on the CPLD. If the results of both implementations are the same, then the design "knows" that it is executed on the "right" board and starts its operation. Similarly, *challenge-response approaches* are published in an Altera white paper [Altb] and a Xilinx application note [Xila]. A challenge consisting of a sequence produced from a *random generator* is sent to a cryptographic algorithm implemented into a non-volatile device. The response of the device calculated with a secure key is compared with the result of the same algorithm and key implemented on the FPGA. The application is enabled if both results are the same. Couture and Kent propose a method where the IP core reads out a secure token, stored in a non-volatile memory in periodic time-lags [CK06]. Inside the token, the type and the life span of the license is encoded. For preventing the cloning of the bitfile, the token also includes a *unique FPGA identification number*.

In the Spartan-3A FPGA family, Xilinx implants a factory-set read-only unique number called *Device DNA* in each device [Xile]. This 57-bit number can be used to develop cores which allow execution only on specified FPGA devices.

A *physical unclonable function* (PUF) returns a unique value, which is extracted from physical properties of an object. *Silicon PUFs* (SPUFs) generate this device-dependent value from different manufacturing-related variations of timing and delay behaviors on nets of the silicon device [GCvDD02]. Related work of SPUFs can be categorized into approaches using *ring oscillators* and approaches using so-called *Arbiter-PUFs*. Gassend and others propose a method using ring oscillators in FPGAs for generating device-dependent values [GCvDD02]. The ring oscillators swing with a certain frequency, and the output is used to enable a counter, clocked with the operational clock. Lee and others [LLG+04] and Lim and others [LLG+05] present SPUFs, realized with an Arbiter-PUF at the IC fabric. An Arbiter-PUF, shown in Figure 2.2, consists of two identical designed delay lines, one for a data signal and

one for a clock signal which can be crossed with multiplexers. The challenge vector enables a route through the multiplexers for both signals. Edges are generated and propagate through the network according to the challenge vector. If the clock signal reaches the flip-flop, the current value of the data signal is registered. The result is a '1' if the clock signal is faster than the data signal; otherwise the result is '0'. Varying the challenge vector will cause different results, which can only be reproduced on the same device. Using another device, the achieved results are completely different. Mjzoobi and others [MKP09] show an implementation of an Arbiter-PUF for Virtex-5 FPGAs whereas Suh and Devadas [SD07] implement an Arbiter-PUF for Virtex-4. Holcomb and others [HBF07] and Guajardo and others [GKST07] present approaches where the *initial state* of SRAMs is used as a PUF. During the power up of SRAMs, some memory cells switch to a '1', others to a '0', depending on the process variations. Guajardo and others reported that *Block RAMs* of some FPGAs can be used for generating a unique key which might be used for design protection.

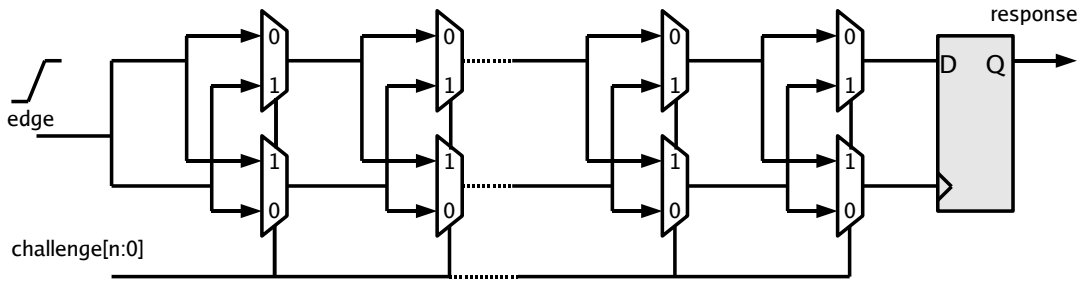


Figure 2.2: An Arbiter-PUF consists of a flip-flop and two delay lines the routing of which can be altered by different challenge values. An edge propagates through the multiplexer network to the flip-flop. The registered response is determined by which signal arrives first. The responses of different challenges are device dependent, hence to minimal uncontrollable path delay variations of different devices.

Simpson and Schaumont propose an authentication system for software, running in a soft core on an FPGA, by using a PUF [SS06]. Later, Guajardo and others enhanced this approach [GKST07].

2.2 Security: Defenses Against Code Injection Attacks

In this section, we show measures against different *code injection attacks*, as introduced in Section 1.4.1. A good overview of defenses against code injection attacks

is further given in [Rag06] and [Erl07]. The related work in this section is divided into six groups: Methods using an *additional return stack, software encryption, safe languages, code analyzer, anomaly detection*, as well as *compiler, library* and *operation system support*. *Control flow checking* methods, which combine security and reliability issues are discussed in Section 2.4.

2.2.1 Methods using an Additional Return Stack

Almost all code injection attacks manipulate the *memory-based return stack*. The return stack can be protected by an additional *hardware-based return stack*. Hardware-based return stacks are usually used for *indirect branch prediction* [KE91]. Xu and others propose a *secure return address stack* (SRAS) which redundantly stores a copy of the memory-based stack [XKPI02]. If the return address from the SRAS differs with the processor-calculated address from the memory return stack, an exception is raised which is handled by the operation system to determine whether it stems from a misprediction or from a code injection attack.

Lee and others propose a similar SRAS approach [LKMS04, MKSL03]. Additionally, scenarios are considered when the control flow of a return from subroutine does not comes back to the point the function was called from. Lee suggests either to prevent these situations or to introduce additional instructions which manipulate the SARS to manually resolve such situations.

Ozdoganoglu and others [OVB$^+$06] present another SRAS method called *Smash-Guard*. In some situations, the behavior of the correct control flow differs from the *last-in first-out* (LIFO) return stack scheme. In these situations which are often referred to as *setjmp* or *longjmp* calls, the control flow mostly returns to a previous call which is deeper in the stack. Ozdoganoglu resolves these situations by searching the target address of the currently executed return instruction in the complete stack.

Furthermore, Xu and others propose methods to divide the stack into a *control* and a *data stack* in [XKPI02]. Inside the control stack, the return addresses and stack pointers and inside the data stack variables, e.g., buffers are stored. This approach effectively solves the problem of buffer overflows. To achieve the stack split, Xu presents two different techniques, one modifies the compiler and the other is a hardware technique which modifies the processor.

2.2.2 Methods using Address Obfuscation and Software Encryption

Bhatkar and others [BDS03] and Xu and others [XKI03] propose methods for *address obfuscation*. To exploit buffer overflows and achieve the execution of malicious code, the attacker must know the *memory layout*. Due to address obfuscation, the achievement of such information about the memory structure is enormously complicated for

the attacker. In these methods, the program code is modified so that each time the code is executed, the virtual addresses of the code and data are randomized. These approaches randomize the base address of the stack and heap, the starting address of the dynamic linked library, as well as the location of static data. Also, the order of local and static variables as well as the functions are permuted. For objects which cannot be rearranged, Bhatkar inserts random gaps by padding, e.g., in stack frames.

Shao and others proposed hardware assisted protection against *function pointer* and *buffer smashing attacks* [SZHS03, SXZ+04]. The function pointers are XORed with a randomly assigned key for each process which is hard to be reconstructed by the attacker. This is a countermeasure for *function pointer clobbering* (see Section 1.4.1). Furthermore, Shao introduces a hardware-based *boundary checking method* to avoid stack smashing. On each memory write, it is checked if the write destination is outside the current stack frame and if so, an exception is raised.

If the software is loaded encrypted into the memory and decrypted in the fetch stage, code injection attacks are impossible, because the attacker needs to inject his code encrypted. The key for de- and encryption is different for each process, hence it is impossible for the attacker to encrypt his code properly. Injection of unencrypted code produces data garbage after decryption and results in a crash of the process. Barrantes and others propose a method which uses an *x86 processor* emulator for simulate the decryption in the fetch stage [BAFS05]. The process is encrypted at load time, whereas Kc and others present an approach where the executable is stored encrypted on the hard disk [KKP03]. The proper key is stored in the executable header which is loaded into a special register for decryption. However, the key is also easily extractable for an attacker which lowers the effectiveness of this approach.

2.2.3 Safe Languages

Many attacks can be launched due to the inherent security flaws which exist in the C and C++ language. Programming in C or C++ allows a lot of programming close to the hardware and the memory layout makes these languages very flexible. However, the programmer must consider many facts if he would like to produce invulnerable code. Safe languages, like Java, are capable of some implementation vulnerabilities which are discussed in Section 1.4.1. Nevertheless, C or C++ are preferred languages for low-level or even for high-level programming, especially in embedded systems which makes a safe implementation of these languages reasonable.

Cyclone is a C dialect which statically analyses given C code at compile-time and inserts dynamic checks at places where it cannot ensure that the code is safe [JMG+02, GHJM05]. Cyclone is designed to avoid buffer overflows, format string attacks, and memory management errors. However, the C syntax and semantics as well as the capability of low-level programming are preserved. Insecure constructs are refused to compile until more information is provided to make these constructs secure. However, Cyclone programs need existing libraries, like the GNU C library

libc which usually compiled with a standard C compiler [Rag06]. To secure library functions, the libraries should also be compiled with Cyclone.

Another approach is *CCured* which is a source to source translator for C [NCH+05]. The used techniques are similar to Cyclone, which includes static analysis and dynamic checks on these points where static analyses are not possible. CCured uses pointer and type analysis to make casts secure. However, these techniques make CCured programs incompatible with existing libraries. This obstacle can be solved by introducing *library wrappers*.

2.2.4 Code Analyzers

Code analyzers can be categorized into two groups: *static code analyzers* and *dynamic code analyzers*. Static code analyzer approaches check either the source code or the compiled object code for vulnerabilities without executing the program. It is impossible to detect buffer overflows statically. Therefore these tools use heuristics whose detection rates are never complete. The term *analyzer* corresponds to a wide area of automatic tools ranging from only considering the behavior of simple code statements to consider the complete source code. Some static code analyzer approaches need *annotations* to the source code whereas other approaches need no annotations.

For example, an annotated static code analyzer is *Splint* [EL02]. It is a lightweight tool which uses annotations to check properties of objects, e.g., the range of a variable. Dor and others introduce the *C String Static Verifier* (CSSV) which is able to detect string manipulation errors [DRS03]. This tool also uses annotations that have pre-, post-, and side-effect conditions. Furthermore, it analyzes pointer interaction and performs integer analysis. A non-annotated static code analyzer for detection of buffer overflows is described by Wagner and others in [WFBA00]. The analysis is done by formulating buffer overflows as an *integer range problem*. Another non-annotated analyzer approach is *PREfix* [BPS00] and its extension *PREfast* [LBD+04]. These methods build an execution model of the analyzed code which includes all possible execution paths of the program.

Other static approaches use *lexical analysis* which can be implemented as an *editor extension*. The written source code is compared to database entries of vulnerable code snippets. If such an entry is found, the tool might examine them further and report the security impact. Approaches using such lexical analysis are *ITS4* [VBKM00] and *Flawfinder* [Whe].

Dynamic code analyzers add further information to the source code and perform test runs in order to detect vulnerabilities. However, not all vulnerabilities might be detected, because the used input stimulus for the test runs might not cover all situations. *Purify* [HJ92] is a tool which tests software to detect memory errors like uninitialized memory access, buffer overflows, or improper freeing of memory as well as memory leakages. The tool is commercially available from IBM known as *Rational*

Purify [IBM]. Haugh and Bishop introduce a dynamic buffer overflow detection tool for C programs, called *STOBO* [HB03]. This tool instruments program code in order to keep track with memory buffers and checks function arguments. If a buffer overflow occurs in a test run, a warning is printed. Ghosh and O'Conner present the *Fault Injection Security Tool* (FIST) in [GO98]. This tool injects malicious strings in buffers and observes the application response to detect vulnerabilities.

2.2.5 Anomaly Detection

Anomaly detection refers to methods which compare the actual application behavior to a specified application profile. Any deviation from the profile will raise an exception which triggers further measures. The application profile can be user-specified or learned from past executions of the program. A disadvantage of these methods is the high false-positive rates due to the identification of any unusual behavior as an attack.

Hofmeyr and others and Forrest and others propose a method for monitoring system calls for UNIX processes [HFS98, FHSL96]. If the system call pattern deviates from the previous recorded pattern, subsequent actions like program terminations can be taken. A similar approach is presented from Wagner and Dean [WD01]. The occurrence of system calls is also monitored and checked with a *system call model*. The model is built statically from the *control flow graph* of a program, where the control flow graph is transformed into a *system call graph*, which models the sequence of the occurrence of system calls. Sekar and others also use a system call model for anomaly detection [SBDB01]. The model, however, is generated dynamically with system call recording in a learning phase. Furthermore, techniques using sliding windows which analyze the system calls inside the window are presented by Forrest and others [FHSL96] and Wagner and others [WD01]. Forrest uses a dynamic learning phase whereas Wagner uses static information derived from the control flow graph.

Feng and others use, besides the system call information, the return address from the stack for anomaly detection [FKF+03]. Like the other methods, the checks are done on system calls. During a learning phase, so-called *virtual paths* are recorded. A virtual path can be built with all return addresses, gathered from the stack, on a system call. These return addresses correspond to all unreturned functions. During the execution of the detection phase, the virtual path is checked on every system call to detect anomaly behavior.

Zhang and others present a hardware approach for detecting anomalies in the program behavior [ZZPL04]. In this approach, the detection is done on the *control flow instruction level* which has a finer granularity than the other system call-based approaches. Jump and branch information, like target addresses or favored conditional branch decisions, are stored additionally in the system memory. Fast memory access is assured through common cache structures of the processor. This method has some similarities with our method which will be introduced in Section 4. However, this approach does not store control flow graphs, rather each branch or jump is separately

looked up using a *context addressable memory* (CAM). Hereby, a hash of the branch or jump address is calculated which acts as index for the branch table. Although this approach needs more memory and has a higher latency as our approach, there is no need for synchronization with the control flow of the executed program. The aim of the method is to recognize attacks by detecting anomalous behavior. Therefore during a learning phase, the decisions of conditional branches are recorded and stored in the branch table. If the recorded decisions differ from the control flow behavior in the detection phase, a warning signal will be risen, whereas if the control flow diverges from the stored jump and branch information, a threat is signaled. The approach was extended with an anomalous path detection which compares sequences of branch decisions of the executed program with the decisions recorded in the learning phase [ZZPL05]. In the second approach, *general indirect jumps* (non returns from subroutine) are considered as well.

2.2.6 Compiler, Library, and Operating System Support

In this section we discuss countermeasures for code injection attacks through enhancement of compilers, libraries, or the operation system.

Compiler Support

Compilers are the most convenient place to insert countermeasures for code injection attacks without changing the programming language. Most attacks exploit buffer overflows to overwrite stack based content. Therefore, many approaches propose *stack frame protection* measures. In the stack, mainly the return address or frame pointers are in focus of attackers. Other items which can be protected with security enhanced compiler support are pointers in the program code. Buffer overflows occur, if there is more data written to the buffer, than its capacity can hold (see Section 1.4.1). Therefore, *boundary check methods* are in focus of this section.

There exist many different methods to protect the return address inside the stack, for example *StackGuard* [CPM+98, CBD+99], *Stack Shield* [Ven00], or *Return Address Defender* (RAD) [CH01]. StackGuard places a so-called *canary word*[1] between the return address and the local variables inside the stack. Before executing the return instruction, the canary word is checked and verified whether it is intact. By exploiting buffer overflows for return address alteration, the canary word is also overwritten which can be detected. Stack Shield uses a redundant return address which is copied in the data segment in the beginning of the function. Before leaving the function

[1] The term *canary word* corresponds to the miner's canary which was used in coal mines as an early warning system. If there were toxic gases in the mine, the birds died before the miners were affected. Canaries sing a lot, which made them very suitable for a visual and audible warning system. The last canaries in mines were phased out in 1986 in the UK [BBC05].

with the return jump, the return address is compared to the copy. If the addresses differ, the program will be terminated. A similar technique is used by RAD. However, the redundant copy of the return address is stored in an array in the data segment which is called *return address repository* (RAR). The RAR is further protected by so-called *mine zones* or *read only techniques*. Mine zones are the read only array boundaries, which protect the RAR from buffer overflow attacks. All these return address protection methods can be applied as a compiler patch for the *gcc compiler*. However, attacks described in [BK00] and [Ric02] are able to cancel the Stack Shield and StackGuard protection. Foremost, StackGuard is vulnerable if the attacker uses a pointer which directly points to the return address which allows the return address alteration without destroying the canary word.

Cowan and others introduce a compiler extension for *pointer protection*, known as *PointGuard* [CBJW03]. The technique protects pointers through encrypting them while they are in the memory. Additional en- and decryption operations are inserted in pointer read and write sequences at compile-time. For example, by accessing a pointer, the pointer is decrypted to a processor register which is safe against malicious overwriting. If a pointer is altered by overwriting the memory during a buffer overflow attack, the decrypted result points to a different location which prevents the access of malicious code.

Lhee and others propose a compiler extension which inserts additional buffer size checks to prevent buffer overflows at runtime [LC02]. The buffer size information is read out of a compilation with debugging information of the program. Using this information, additional checks are automatically inserted into the source code.

Erlingsson and others propose a fine-grained software-based memory access control technique called XFI [EAV+06, ABEL09]. This technique enriches the program code to grant access to an arbitrary number of memory regions. Furthermore, the entry and exit point of a program can be controlled using XFI. Budiu and others propose additional instructions to extend the instruction set architecture (ISA) for XFI hardware support [BEA06].

Jones and Kelly propose a method to identify *out-of-bound pointers* [JK97]. Every result of a pointer arithmetic must reference the same object as the original pointer. If not, the pointer is out-of-bounds. Such pointers can be identified dynamically by additional instructions which are included at compile-time and a new object table which is maintained during the execution. If a pointer is out-of-bounds, this pointer value is set to '-2'. The problem of this approach is that out-of-bound accesses are not allowed in ANSI C, however, such pointers are used in many programs. Therefore, Ruwase and Lam extend this approach with an *out-of-bound object* and call this approach *C Range Error Detector* (CRED) [RL04]. If a pointer becomes out-of-bounds, it is redirected to a special out-of-bound object which keeps the original pointer value and the referenced data. This approach prevents buffer overflows, because all data written over the bounds of the buffer are automatically redirected to other memory locations managed by the out-of-bound object.

Library Support

Many buffer overflows are caused by mishandling vulnerable *standard C library functions*. Particularly, string handling functions are vulnerable for buffer overflow attacks. Therefore, the obvious solution is to design safer libraries. Safe string function replacements are `strlcpy()` and `strlcat()` [MdR99] and *SafeStr* [MV05] which are immune to buffer overflows and format string vulnerabilities. *FormatGuard* is a patch for the glibc library to protect the `printf()` function from format string vulnerabilities [CBB+01].

Baratloo and others introduce two methods against buffer overflows which are completely transparent: *libsafe* and *libverify* [BST00]. Both approaches are implemented as dynamic link libraries under the Linux operating system. The library *libsafe* intercepts all calls to vulnerable functions of the glibc library and substitutes these calls with alternative functions which are not vulnerable to buffer overflow or format string attacks. The library *libverify* uses binary re-writing of the process memory to verify critical elements of the stack frame before they are used. The verification and protection against buffer overflows is similar to the StackGuard [CPM+98] approach, however the implementations differ. Whereas StackGuard is applied during the compilation, *libverify* embeds the verification code at the start of the process. The advantage is that the code does not have to be recompiled which makes this approach completely transparent to the user.

Robertson and others [RKMV03] and Krennmair [Kre03] propose countermeasures for *heap-based attacks*, described in Section 1.4.1. The allocation and deallocation routines of the standard C library are modified to protect the header of the heap segment. Robertson includes a padding mechanism and a checksum in the header on frame allocation and verifies these information, if the segment should be freed. Krennmairs technique, called *ContraPolice*, protects the heap pointer in the header of each heap segment by randomly generating canaries like the StackGuard approach for stack-based headers.

Operation System Support

Finally, the operating system can be enhanced to protect programs from code injection attacks. *Non-executable stack* prevents the execution of malicious code, injected into the stack. However, this approach prevents some allowed situations where code is executed in the stack. Examples are *functional programming languages* which generate code during runtime in the stack, *function trampolines* for nested functions used by the *gcc compiler*, or *stack-based signal handling* which is used by Linux. A patch for a non-executable stack for the Linux operating system was provided in [Des97] which also handles the above mentioned executions by disabling the protection in case of these situations. However, this approach is defeated by Wojtczuk [Woj98].

Lately, processor vendors have introduced hardware support to prevent the execution of code from the stack. With a new flag, the so called *NX* (No eXecute) bit, memory regions can be declared page-wise as non-executable areas which are excluded from execution by the hardware. Non-executable stack approaches for the Linux operating system, like *PaX* [PAX03] or *Exec Shield* [vdV04] are able to use this NX bit, or emulate it on processors which have no NX bit support. The technique can be combined with write protection to achieve that no memory location in the process can be marked as writable ('W') and executable ('X'). This so called *W⊕X protection* prevents attackers from injecting malicious code with subsequent execution. Nevertheless, Shacham demonstrated that it is not necessary to inject code in order to do malicious computations [Sha07] (see also Section 1.4.1).

StackGhost is an operation system-based approach to protect the stack frame for systems running on the *SPARC* architecture [FS01]. This method utilizes special SPARC features like the *windowed register file* (see Section 4.5.1) and provides a redundant copy of the return address. StackGhost is available as a patch for the *OpenBSD* kernel.

2.3 Reliability: Measures against Faults and Errors

In this section, measures against faults and errors in embedded systems are described. To achieve a fault tolerant system, errors must be detected and subsequently corrected. At RTL and system level, error detection and correction is done using *redundancies*. There exist three different types of redundancies: *Hardware, time*, and *information redundancy*. Beyond that, there exist hybrid approaches and approaches which are using different types of redundancies for error detection and correction. For example, the control flow checking methods, that will be described in Chapter 4, exploit hardware and information redundancies for error detection and time redundancies for error correction.

The following criteria are important for the evaluation of different error detection and correction methods. The *error coverage* denotes the degree of errors that can be detected by a method. Another criterion is the *detection latency* which denotes the time between the occurrence of a fault and its detection. This time is important to prevent a *system failure*. Only if an error is detected with a low latency, the error handling can react to transfer the system into a secure state or to trigger error correction measures. On the other hand, the measures against errors may cause overheads, for example, *area overhead*, *memory overhead*, or increase the delay of the processing core.

Error masking should also be considered. If an error is detected which does not affects any outputs, the error is masked. Correcting such errors is not necessary,

because the error has no effect on the system and therefore, the reliability is not increased by correcting this error. Moreover, if time redundancy is used to correct this error, the system performance is decreased. Weaver and others propose to record a detected error in microprocessor registers with a so-called π bit (*pi* for *possibly incorrect*) [WEMR04]. Later in the calculation, a hardware checker can decide by examining the π bit if this possible incorrect value affects the overall processor state. Furthermore, Weaver proposes to increase the error masking probabilities by *invalidate register states* in microprocessors during long delays, for example at pipeline stalls. The probability that during such long delays a soft error effects the state of valid registers and turns this fault into an error is high. By invalidating the register state, the occurrence of errors can be reduced at the same soft error rate. Nevertheless, the valid state must be restored after the delay by recalculation which impacts the processor performance.

The second part of this section describes *fault prevention* and *detection* approaches for single event effects at the process and device level. Fault detection schemes at device level can be combined with error correction methods using redundancies, e.g., the data path protection methods described in Section 4.5.5.

2.3.1 Hardware Redundancy Methods

A common method for error detection is the duplication of the circuit with subsequent comparison of the results. If the circuit is instantiated three times, the correct result can always be chosen without delay, assuming only single errors. This so-called *Triple Mode Redundancy* (TMR) technique [SS98] uses an additional *majority voter* which compares the results of the three different processing instances and returns the result which is calculated by at least two of them. Hardware redundancy can detect permanent and transient faults which lead to value or timing errors. However, TMR has an area overhead of over 200% (two redundant units plus the voter).

On the architecture level, the duplication of complete processing units or buses is most of the time too cost-intensive and thus prohibitive due to area overhead and power consumption. Hence, these approaches are only used in safety-critical systems with a high demand of reliability [Bar81, MAF+99, NMCB97].

Some approaches replicate only critical parts of the processing unit. These *partial redundancy* methods requires much less area and produces a much lower power overhead than the full redundancy methods. However, the error coverage per overhead (area, power) is higher than for the full redundancy methods.

Ragel and Parameswaran suggest to introduce a second instruction decoder for control flow instructions with its own instructions [RP06, Rag06]. The control flow instructions for this redundant decoder unit are stored inside the program code. The CPU registers and the program counter are duplicated to *shadow registers*. For SRAM caches, approaches have been developed which use redundant memory cells

to map out defective memory locations during fabrication [YP97] or to repair permanent memory faults at runtime [NAB03].

Austin introduces the *DIVA pipeline* [Aus99] which consists of additional checker pipeline stages. These additional stages are inserted before the commit stage and check the calculation of the previous pipeline stages by refetching the operands from the memory or registers and recalculating the operation. DIVA accepts only checked results for the further processing to the commit phase. The simple rudimentary checker pipeline is assumed to be fault-free which is not the reality in today's deep submicron designs. Austin suggests to implement the checker pipeline using larger transistors which makes it resistant to single event effects like radiation or other noise-related faults. The complexity of Diva is lower than in the case of fully redundant units, but it also reduces the performance, due to a longer pipeline. Weaver and Austin adapt this approach to an Alpha 21264 processor [WA01]. The additional checker requires less than 6% of the area and less than 1.5% of the power of the Alpha processor.

Bower and others extend the DIVA approach to detect *permanent degeneration faults* during runtime [BSOS04, BSO07]. In [BSOS04], permanent faults in memory structures inside the processor, like the *reordering buffer* or the *branch history table*, are detected by writing the same value to two locations with subsequent comparison. If a permanent fault is detected, the erroneous row is excluded from further usage. The handling of permanent faults in general processor logic is described in [BSO07]. High performance microprocessors have inherent redundancies to increase the system throughput, e.g., multiple ALUs or register files for *simultaneous multithreading* (SMT). By fault diagnosis, an erroneous unit is detected and switched off (deconfigured) whereas the complete system continues operation at a lower performance level. By using multiple DIVA checkers, permanent faults in these checkers can be corrected by switching of the corresponding unit as well.

2.3.2 Time Redundancy Methods

Time redundancy methods can be used for error detection and for error correction. The result of a computation is calculated twice or more on the same resources with a subsequent comparison. The advantage over hardware redundancy is the lower area overhead. Only the checker or voter and additional control logic causes area overhead. However, *permanent faults* and *design errors* cannot be detected with these methods. Error correction is usually done by *rollback* or *checkpointing methods*.

Error Detection by Multithreading

Rotenberg proposes a method called *AR-SMT* to execute a program twice on the same CPU using *simultaneous multi-threading* (SMT) [Rot99]. On SMT machines, the register and the processor state is separated for each thread for the seamless execution

of many threads. However, only one pipeline with the arithmetic units exits. The redundant program or thread, called *R-stream*, is started with a slight lag due to a feed back over a delay buffer of the leading thread, called *A-stream* (advanced stream). The executed instructions between the A-stream and the redundant thread are verified.

Reinhard and Mukherjee extend Rotenbergs approach by introducing a so-called *sphere of replication* [RM00]. Data that enters the sphere must be duplicated and results that leave the sphere must be checked. Inside the sphere, the redundant threads are processed independently on the SMT pipeline. Checking only results that leave the sphere reduces the verification overhead enormously compared to Rotenbergs original approach which verifies every instruction. Ray and others adapt the SMT-based reliability approach to *superscalar out-of-order* microarchitectures [RHF01]. The R-stream is produced by dynamic injection of redundant instructions and in case of an error, the program is reverted to a correct state. Other time redundant fault tolerance approaches using superscalar processors are [Fra95] and [RSR00].

The architecture of Rotenbergs approach is refined to a so-called *slipstream*[2] processor which is able to speed up single thread processing [SPR00]. The complexity of the A-stream is reduced by *speculative removing* of ineffectual computations, whereas the R-stream has the full complexity. Using this technique, the throughput of the (multi-)processor can be increased over the execution of the single thread alone, due to speeding up the A-stream by removing instructions, and on the same time speeding up the R-stream by transferring control flow information from the A-stream to the R-stream. This approach can be used either with an SMT or with a multiprocessor architecture. If both streams execute the same instructions, an error can be detected by time redundancy or time and space redundancy using multiprocessors. Nevertheless, if a fault or an error occurs in an instruction that is only executed by the R-stream, these error cannot be detected. This slipstream approach counts to the so-called *partial redundant threading* (PRT) methods. Other PRT methods are introduced in [GV05] and [WP06]. Reddy and others compare PRT methods and evaluate the performance impact and the fault coverage [RRP06]. Many other time redundant fault tolerance approaches exist for SMT processing and *chip multiprocessors* (CMP), e.g., [VPC02] and [GSVP03], respectively.

Oh and others present a method called *Error Detection by Diverse Data and Duplicated Instructions* (ED^{4}I) for redundant execution of a program [OMM02]. The processed data of the second redundant program is transformed. The result is that both programs implement the same functionality, but with different representations of the data. These data diversity enables the detection of permanent faults in the data path besides temporal faults.

[2]The name slipstream is an analogy to stock-car racing. Due to the aerodynamic resistance, the maximum speed of the cars is limited. However, if one car is driven in the slipstream of another car both cars can achieve a higher maximum speed. The second car has less air pressure in the front, whereas the front car has lesser air disturbances on the rear [SPR00].

Error Correction by Rollback

A common usage of time redundancy is error correction for temporal faults by doing *rollbacks* and *checkpointing*. Rollbacks restore a correct state which was previously recorded on so-called checkpoints. The recorded state coverage and the distance between the checkpoints are very important. Rollback techniques can be categorized by means of the number of checkpoints or the rollback distance. In other words, they can be categorized by how far they are able to jump back into the past.

On the one end, there are rollback methods which act at program level. If an error is detected, the program is terminated and restarted with the same inputs. In this case, the checkpoints are the entrance of the program and the covered state are the inputs. By using this technique, the input variables must be stored as long as the calculation is not finished. An example of this rollback technique is [WLG+89].

Approaches which have a finer granularity do checkpointing during the execution of the program. At a checkpoint, all relevant information for retrieving the execution of this point are stored. This may include the program counter, the register content and some variables in the system memory. Choosing the right checkpoints for state storing as well as the distance between the points are, for example, researched by Chandy [Cha75]. An early implementation of checkpointing is the STAR computer [AGM+71]. Note that checkpoints are called *rollback points* in this paper. For fine granular rollbacks at instruction level, Tamir and Tremblay extend the register file and the cache [TT90]. With these extensions, the processor is able to recover the state from a few cycles ago very fast. More about checkpointing can be found in [KK07].

If an error occurs and is detected fast enough so that register or memory contents are not overwritten, the corresponding instruction can be re-executed by a pipeline flush and subsequent refill. Due to the unaffected state of the register file and memory, only the program counter must be restored for this kind of rollback. In our control flow checking approach (see Chapter 4), we use a similar technique for error correction. However, we do not flush the pipeline. We set on the erroneous instructions an annul bit which prevents these instructions from being executed.

Finally, *micro-rollbacks* are rollbacks which can jump back one single clock cycle. This can be achieved by registering all registers in the pipeline with additional so-called *history registers* [BZS+06]. The history registers hold the state before the clock cycle. After each original pipeline register, a multiplexer is inserted which uses the value from the original register in normal operation and in case of a rollback the value from the history register. This approach is used for error correction of the data path protection technique, described in Section 4.5.5. With this technique, a detected error can be recovered in one clock cycle. However, the hardware overhead for storing the last state by duplication of the pipeline register is not small. In [PV01], Pflanz and Vierhaus describe a technique which uses a lower overhead by redundantly fetching every instruction twice. Using this double fetching approach, usually only the

first instruction is executed in the pipeline. However, if an error occurs, the processor switches back to the redundantly fetched instruction. Nevertheless, the performance is degraded by factor 2, which is unacceptable in most applications.

2.3.3 Information Redundancy Methods

Information redundancy methods are often counted to the system level methods, whereas hardware and time redundancy methods belong to the *register-transfer-level* (RTL). Therefore, information redundancy methods cause either additional hardware (area or memory) overhead or performance impact and can also be counted to hardware or time redundancy methods. One of the well-known information redundancy methods is the protection of the data using *error correcting codes* (ECC). If, for example, a bus communication should be protected, new signals (hardware redundancy) can be used to transmit the additional ECC bits which cause hardware overhead. However, the additional ECC bits can also be subsequently transmitted over the common data signals which causes a time overhead (time redundancy). Considering permanent faults, the underlying redundancy technique is important. A permanent broken data signal affects the recovery of data more if time redundancy methods are used than if hardware redundancy methods are used.

Protecting Storage Elements

As mentioned before, the most common form of information redundancy is *coding*. Protecting data with coding is usually used for data communication or storage. Storage includes the protection of the data in external or internal memories as well as on disks. An important property of codes is the *code distance*. The code distance is the minimal *hamming distance* between any two valid codewords. The hamming distance between two code words is the number of bit positions in which the two words differ. Furthermore, it must be distinguished between codes which have error detection and correction possibilities, the so-called ECC codes, and codes which only have error detection possibilities. To belong to the first category for single bit errors, the code distance of the code must be greater than 3 [Lal01]. Another important property of the code is the number of errors in one code word, which can be detected or corrected. There exist many different error correcting codes, like *parity codes, checksums, cyclic redundancy checks* (CRC), *M-of-N, Berger*, or *arithmetic codes* [KK07].

Applications for protecting storage elements are ECC protection of external or internal memories, and registers, as well as fault-tolerant state machines [Mey71]. Gaisler proposes the fault-tolerant *ERC32* and *Leon3* processors, which protect register files and the caches with additional parity or ECC bits [Gai94, Gai02, Gaib].

Protecting Combinatorial Logic

The methods described above can only be used for sequential storage elements. However, for combinatorial logic, like adders or multipliers, there exist so-called *concurrent error detection* (CED) techniques [SHB68, RF89, MM00]. In [MM00], Mitra and McCluskey compare CED techniques using hardware redundancy based on their area overhead and the protection they provide against multiple failures and common-mode failures.

The general principle is that a system realizes a function f and in response to an input sequence i produces the output $f(i)$, and another unit independently predicts the output $\widehat{f}(i)$ while a checker unit compares the outputs and produces an error signal in case of a mismatch (see Figure 2.3).

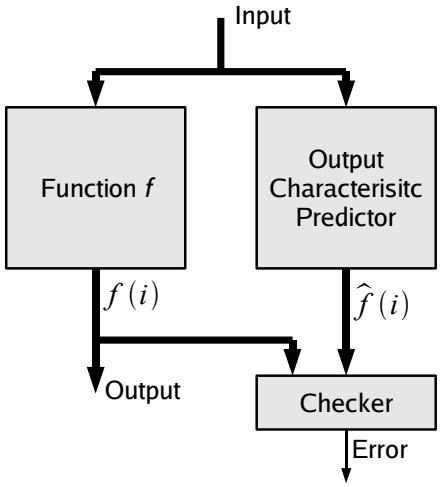


Figure 2.3: General architecture for concurrent error detection [MM00].

The simplest CED method is the *duplex system* (see Figure 2.4a). Here, two modules implement the same logic function while the implementations can be different. The outputs of the modules are checked by a comparator and if the results do not match an error is indicated. Unless identical errors are produced by the modules data, integrity is guaranteed if the comparator is fault-free (see also Section 2.3.1).

Another CED method is *parity prediction*. Figure 2.4b shows the basic architecture using a single parity bit. A general problem of parity methods is that one error may lead to multiple bit changes in the output. In this case, the method cannot reliably detect the error. One solution is to build cores in the way that the logic gates generating one output bit are not shared between the other output bits. In this case, a single error cannot affect more than one bit. A checker compares the parity to an independently predicted parity. A higher area overhead is the result, if any logic be-

tween the different output bits is not shared. Parity prediction methods are shown, for example, in [Nic93], [TM97] and [NDMF97].

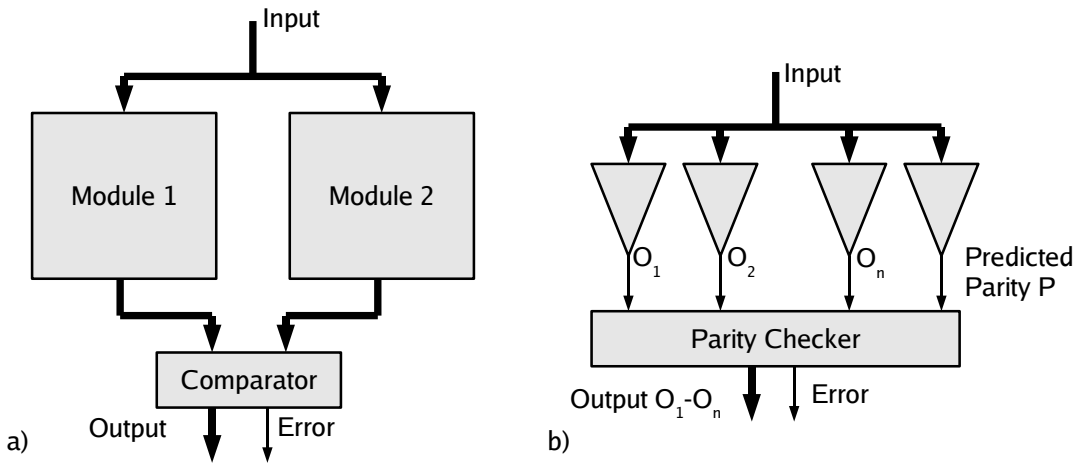


Figure 2.4: On the left side, the duplex system is shown. On the right side, a CED method using single parity bit prediction is shown. No logic sharing between the output bits is allowed [MM00].

Finally, there also exist methods which use *unidirectional error codes*. These methods assume that all errors are unidirectional which means that either '0's change to '1's or '1's change to '0's but not both. *Berger codes* and *Bose-Lin codes* are two unidirectional error detecting codes used for CED.

A Berger code-word is created by appending a binary string which includes the number of '0's (or the bit-wise complement of the number of '1's) in the given information word (see Figure 2.5a). It requires some extra space and detects all unidirectional errors. Usually, these codes are used on communication channels; the bit-wise complement of '1's in the information word is represented by the check bits. It is important to notice that a single error causes unidirectional errors at the outputs and therefore logic circuits are restricted to be synthesized to be inverter-free (except at the primary inputs). Lo and others introduced a Berger code protected ALU in [LTRN92]. Pflanz and Vierhaus extend this approach by further protecting registers and shifters [PV01].

Using Bose-Lin codes (see Figure 2.5b), it is possible to detect t-bit unidirectional errors in the code word. The circuit also has to be inverter-free. Additionally, there is a restriction on the amount of logic sharing since the code can only detect t unidirectional errors [MM00].

Mitra and McCluskey analyze the overhead and the protection against temporal and permanent faults by simulation [MM00]. The area overhead of techniques for unidirectional error detecting codes (Berger, Bose-Lin) is quite high. Using parity

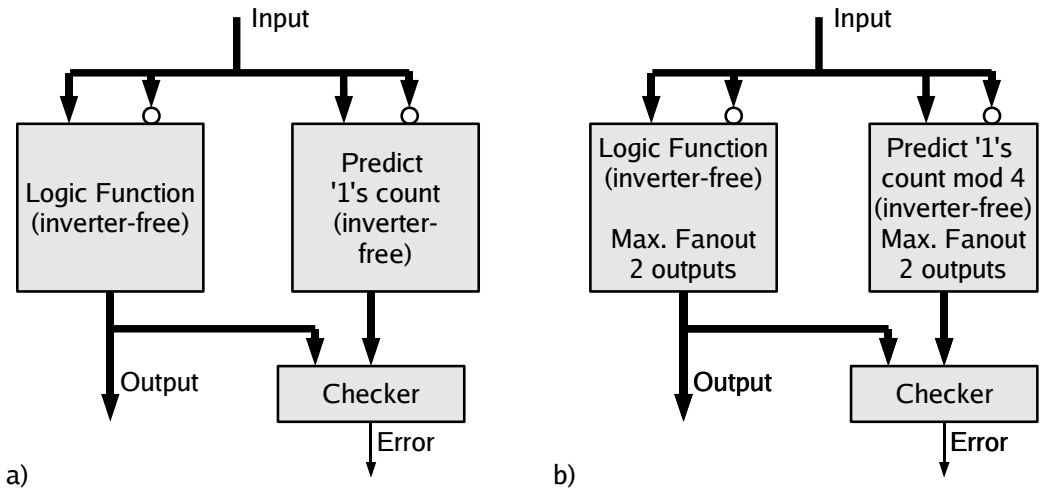


Figure 2.5: On the left side, a circuit which is protected by Berger codes is depicted. On the right side, a Bose-Lin code protected circuit with $t = 2$ is shown. For both methods, it is important that the combinatorial logic of the function and the code prediction circuit is inverter-free except of the primary inputs [MM00].

prediction, the area overhead is smaller than using duplication. Note that routing overhead has not been considered. Simulation showed that diverse duplication is the best way to detect temporal and permanent faults, better than identical duplication and parity prediction.

Joshi and others present a time redundancy based CED technique targeting *involutionary functions* in [JWSK06]. A function is involutional if $f(f(x)) = x, \forall x \in \mathcal{F}$. The proposed technique detects permanent and transient faults that affect the output. It can also detect permanent faults in a function even if the faults do not affect the output. This also enhances the security of the implementation since it is possible to detect attacks to the algorithm. The time overhead (after optimization) is less than 8%, the area-overhead is less than 18% and the fault coverage is almost 100%. This technique is primarily used in the domain of cryptography where a lot of involutionary functions appear.

Almukhaizim and others proposes a concurrent error detection technique for combinational and sequential logic [ADM04]. A state prediction of the next state of a control unit is calculated based on the current state and the inputs. The hardware overhead is typically smaller than in case of full redundancy on large state spaces. However, the method can only monitor the correct state transitions, but not the correct storage of the state.

2.3.4 Prevention and Detection of Single Event Effects

The prevention of single event effects can be done on the process level by applying *triple wells* or *buried layers* [MW04]. The goal is to drift the generated charges, produced by an ionized particle impact, away from the active region. By using buried junctions in a triple well architecture, an opposite electrical field with respect to the NMOS-depletion is created which causes the drift of the electron-hole pairs, generated by the particle impact, deep into the substrate. Furthermore, by replacing the radiating *BPSG dielectric* by purer mold components, the effect by thermal neutrons can be significantly lowered. The usage of *silicon-on-insulator* (SOI) can further increase the robustness against soft errors by reducing the charge depth.

At the device level, the critical charge Q_{crit} stored inside a register cell can be increased by increasing the capacitance or the voltage [MW04]. The capacitance can be increased by using larger transistors. Both, however, raise the area and the power overhead which restricts the applicability to selected nodes on critical parts of the circuit only. Memory cells which are insensitive to single event effects are introduced in [CNV96] and [HKW+03]. These cells use more transistors which also increase the area and power overhead. Mitra and others propose registers with *built-in soft error resilience* (BISER) for general logic in [MZS+08]. These registers are also insensitive to single event effects.

To detect single event transients, Nicolaidis introduces a shadow register concept in [Nic99]. A signal produced by combinatorial logic is sampled by multiple redundant registers with different clocks. The time lag δ between the rising edges of the clocks is greater than a typical single event effect. By using the original register and one shadow register, single event effects can be detected by comparing the registered values. By using three registers, one original and two shadows with three different clocks, the single event effect can be corrected by a majority voter. A similar approach is proposed by Mavis and Eaton [ME02].

Ernst and others use this technique with a main (original) and a shadow register to detect timing errors for *dynamic voltage scaling* (DVS) [EKD+03]. If timing errors occur, only the main register is affected due to the later sample point of the shadow register. Therefore, if a timing error is detected, the value of the shadow register can be fed back to the registers' input. The penalty is a double latency for the operation. These enhanced registers are called *Razor flip-flops*. The approach is extended to additionally detect and correct single event effects and variation-induced delay errors by Das and others which is now called *Razor-II* [DTP+09].

A similar result is presented by [BZS+06], where Nicolaidis flip-flops are extended with an additional history register which can be used for rollbacks to correct errors. This technique is later combined with our control flow checking methods (see Section 4 and in particular Section 4.5.5). The technique is extended with an ECC protection for detecting and correcting multiple errors [BZSH09]. Mehrara and others present an approach to detect single event effects by doubled sampled latches [MAS+07].

The correction is done by rollbacks to fixed checkpoints which separate the program into so called *epochs*.

Some volatile FPGAs have an internal protection of the configuration memory against single event upsets. A CRC value of the configuration memory is calculated periodically and compared to the CRC value stored in the configuration bitfile which is calculated at bitfile generation [Alta].

2.4 Reliability and Security: Control Flow Checking

Control flow checking (CFC) denotes the task of checking the control flow of a program according to a given specification. The specification is derived mostly statically at compile-time from the program. Control flow checking combines reliability and security issues and is a countermeasure against *single event effect*, *degeneration faults*, *code injection* and *invasive physical attacks*.

Related work on control flow checking can be divided into completely software-based approaches and approaches using an additional hardware *checker unit* or a *watchdog processor*. Usually in these approaches, the program code is first structured into basic blocks[3]. Other approaches achieve control flow checking by redundantly decoding the control flow instructions in an additional checker unit (e.g., [RLC+07]).

2.4.1 Software-Based Methods

Software-based control flow checking techniques belong to the so-called *software implemented hardware fault tolerance* (SIHFT) methods. A famous software-based CFC technique is called *Control Flow Checking using Assertions* (CCA) [KNKA96, MN98]. After the creation of the *control flow graph* (CFG, see also Section 4.3.2), special control instructions are inserted into the program code at the beginning and the end of a basic block (see Figure 2.6). The approach introduces two identifiers which are set and checked with these instructions. At the entrance of a basic block, a *basic block identifier* is assigned to a variable. Moreover, corresponding to the control flow graph, a special *control flow identifier* is checked and subsequently set for the next basic block. At the end of a basic block, both identifiers are verified, so erroneous jumps or branches from or in the middle of a basic block are detected. By checking the control flow identifier, the correct processing order of the basic blocks is ensured. The advantage is that no hardware modules are required, however this approach has

[3] A basic block is a sequence of code which is executed successively without any jumps or branches except, possibly, at the end. The basic block can only be left at the end of a block and can only be entered at the beginning. Only the last instruction can be a jump or branch and only the first instruction can be a jump or branch destination (see Section 4.3.2).

impact on the performance of the program code and the erroneous jumps can only be checked at the transitions of the basic blocks.

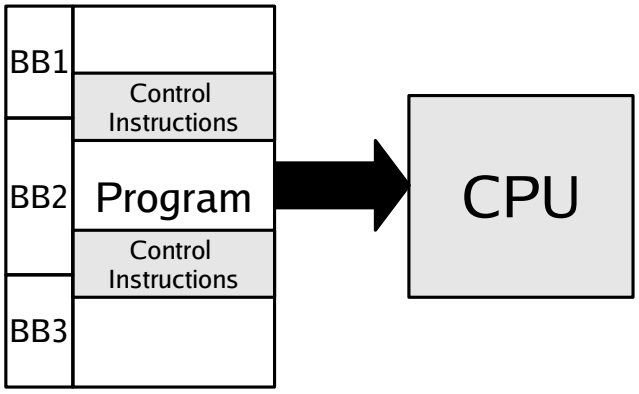


Figure 2.6: Control instructions are usually inserted before and after a basic block (BB1-BB3) in software-based control flow checking approaches. The additional control flow instructions check the executed control flow according to the control flow graph.

The approach is enhanced for real-time distributed systems to achieve a lower performance overhead and faster error detection in [ANKA99]. The additional instructions are inserted at an intermediate level of the compiler which makes this technique, called *Enhanced Control Flow Checking with Assertions* (ECCA), language independent.

Another software-based control flow checking approach called *Block Signature Self Checking* (BSSC) is introduced by Miremadi and others [MKGT92]. The code is also structured into basic blocks and additional instructions are added at the entry and at the end of each block. Checking is done by storing a signature, e.g., the current address, into a variable at the basic block entrance. Before leaving the basic block, this signature is verified. The method can verify the subsequent linear execution of the basic block. However, the processing of the correct basic block order cannot be verified.

Oh and others introduce a technique called *Control Flow Checking by Software Signatures* (CFCSS) [OSM02]. A unique signature is assigned to each basic block and the signature is embedded with the signature difference to the predecessor block in the code. During the execution, a runtime signature is calculated and stored in a general purpose register. The signature of the last block and the stored signature difference are used to calculate and verify the current runtime signature at each basic block entrance. In other words, this approach checks if the correct predecessor of the current basic block, according to the CFG, was processed. However, if a basic

block has more than two predecessors, the method is not applicable. In this case, an additional runtime variable is introduced to resolve the problem. Borin and others propose error classification for control flow checking and analyze the error coverage of the most existing software-based approaches [BWWA06]. Furthermore, they introduce two methods which are enhancements to the original CFCSS method. The first method is called *Edge Control Flow Checking* (EdgCF) and the second is called *Region Based Control Flow* (RCF) technique.

Other similar approaches are SWIFT [RCV+05] and YACCA [GRRV03, GRRV05]. All these method insert control instructions at the basic block borders into the program code, as depicted in Figure 2.6.

Bagchi and others introduce the *Preemptive Control Signature* (PECOS) checking, which is able to prevent jumps or branches in case of an error [BLW+01, BKIL03]. The program code is equipped with additional checker instructions before each control flow instruction. The runtime target address is determined and verified with the valid target addresses, extracted from the compiled code. The list of valid target addresses is stored inside the code, whereas the runtime target address is determined by loading the control flow instruction into a register and decode it by software. If the runtime target address is not in the list with valid addresses, an exception is triggered which prevents the execution of the erroneous jump or branch. The drawback of this approach is that only the integrity of the control flow instruction in the memory is checked. Transient faults, such as single event effects in the control path cannot be detected by this method.

Abadi and others introduce a software-based CFC technique for security issues called *Control Flow Integrity* (CFI) [ABEL05, ABEL09]. This method focuses on indirect calls and returns. The destination of indirect calls and returns are determined at compile-time, and each of these jump destination are labeled with a unique identifier in the code. Instructions to check the identifier of the destination are added to the program code before an indirect jump. Only if the identifier is correct, the jump is executed. Budiu and others present an *instruction set architecture* (ISA) extension which introduces new instructions for CFI hardware support [BEA06].

2.4.2 Methods using Watchdog Processors

A *watchdog processor* is a simple coprocessor which is able to monitor the behavior of a main processor in order to detect errors [Lu82, MM88]. The predecessor of the watchdog processor is the *watchdog timer* [CPW74, OCK+75]. A watchdog timer is reset by the program running on the main processor at certain intervals. If the monitored program hangs, the timer is not reset anymore. Therefore, the timer produces an overflow which generates an interrupt. Inside the *interrupt service routine*, countermeasures can be started, e.g., the erroneous program can be terminated.

A watchdog processor is initialized with information about the main processor or the process which should be monitored. At runtime, the watchdog processor concur-

rently collects information about the execution of the program in the main processor and compares the gathered with initial information to detect errors. The information may include the *memory access behavior*, the *control flow*, the *control signals*, or the *reasonability of results*. Mahmood and McCluskey give a survey over error detection with watchdog processors in [MM88]. Traditionally, a watchdog processor is coupled with the main processor via a system bus. However, other approaches exist where the watchdog processor is directly attached by dedicated signals.

The advantages of watchdog processors are the lower overhead than the duplication of the main processor, the possibility of concurrently checking the execution which results in none or only little performance degeneration and the detection of common or related design errors of the program or the processor due to the diversity of the processors architectures.

Watchdog processors, when used for control flow checking, have a *watchdog program* which is derived from the control flow of the checked program. The control flow of a program can be represented in a graph whose nodes represent sequences of code, e.g., basic blocks or whole functions, and the edges between the nodes represent the control flow. An identifier, often called signature which is known by the watchdog program is attached to each node. The signatures can be assigned at random or they can be derived form the instructions inside a node. Techniques using the arbitrarily assigned signatures are called *assigned-signature control flow checking* and techniques using the derived signature are called *derived-signature control flow checking* [MM88]. The different watchdog processor approaches can be further categorized by the storage of the watchdog signatures. Therefore, the methods can be divided into two groups, called *Embedded Signature Monitoring* (ESM) and *Autonomous Signature Monitoring* (ASM). ESM methods embed the watchdog signatures into the code of the checked program. To verify a signature, the corresponding signature must first be transferred to the watchdog or to the main processor, depending on the comparison location. The watchdog processors for the ASM methods have their own memory to store the signatures. Therefore, the watchdog must be initialized with all watchdog signatures before the program execution. In summary, there exist four categories for control flow checking with watchdog processors (see Table 2.1).

Watchdog-processor-based CFC approaches can be further categorized according to their error detection capability. The first category checks that the nodes are processed in an allowed sequence whereas approaches of the second category verify the instruction sequence inside a node. The third category includes schemes which do both [MM88].

Assigned-Signature Control Flow Checking

During the execution, the arbitrarily assigned signatures used for assigned-signature CFC are transferred to the watchdog processor for verification. Usually, the trans-

	signature storage location	
	ESM	ASM
Assigned-Signature CFC	SEIS [PMHH93]	SIC [Lu82], ESIC [MH91]
Derived-Signature CFC	PSA [Nam82], SIS [SS87]	Cerberus-16 [Nam83], RMP [ES84]

Table 2.1: Four different categories for control flow checking using watchdog processors with some example references. The methods are categorized by different watchdog signature storage locations (embedded into the program: ESM; in additional memories for the watchdog processor: ASM) and the different type of signatures (derived, assigned) according to [MHPS96].

ferred signatures are compared by the watchdog processor to the watchdog signatures, stored in a separate watchdog memory (ASM method). The advantages of these methods are the ease of implementation and the possibility to perform runtime checks asynchronously. However, the drawbacks are the performance impact, due to the program-based transfer of the signatures to the watchdog with additional control flow instructions, and the low error coverage since only the sequence of the signatures is checked.

The first known method is introduced by Yau and Chen [YC80] which assigns prime numbers to loop-free intervals which are checked at runtime. Lu proposes a method called *Structural Integrity Checking* (SIC) [Lu82]. The method assigns labels to high-level control flow structures which are verified by the watchdog processor. The approach is enhanced by Majzik and Hohl which is called *Extended Structural Integrity Checking* (ESIC) [MH91].

An embedded signature monitoring approach for assigned-signature CFC is introduced by Pataricza and others, called *Signature Encoded Instruction Stream* (SEIS) [PMHH93, MHPS96]. In this approach, each basic block is assigned a unique signature which further encodes the successor basic block. The signatures are transferred to the watchdog processor during the execution which verifies the control flow of the program only using the information encoded in the signatures. Therefore, the watchdog processor needs no signature storage memory and initialization phase.

Derived-Signature Control Flow Checking

Derived-signature CFC uses a signature calculated from the properties of the executed instructions inside a node. To check all instructions, a signature, e.g., an XOR, hash or CRC value, of all instructions of a basic block is calculated offline (at compile-time). At runtime, a checker unit calculates the signature of the executed

instruction in a basic block. When leaving a basic block, both signatures can be compared and errors inside the basic block can be detected. The derived-signature CFC methods can also be categorized by the storage of the precalculated (golden) signature in ESM and ASM methods.

Embedded Signature Monitoring *Derived-signature ESM methods* store the offline calculated signature (golden signature) in the program code with additionally inserted instructions at the end or the beginning of each basic block. During runtime, the calculated signatures from the watchdog processor are compared to these embedded signatures. The advantage of these methods is that all instructions can be checked and a new program already contains the corresponding signature (see Figure 2.7). The disadvantages are the performance impact and that a fault can only be detected at the end of a basic block which may be too late. Also, a single event upset during the execution of the additionally inserted instructions can lead to a false detection or spoofing of an error.

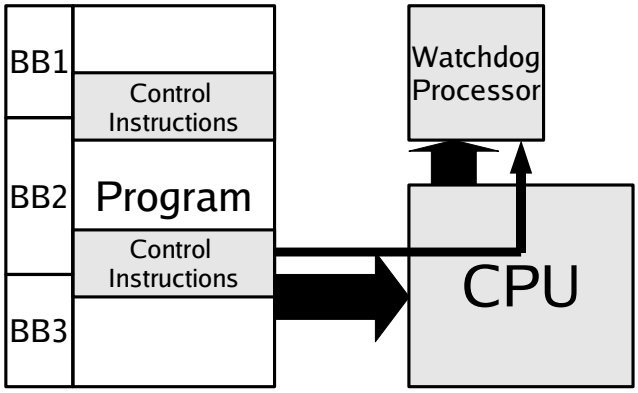


Figure 2.7: In the *derived-signature ESM CFC* approaches, a signature is calculated from the executed instructions by a watchdog processor. The golden signatures are calculated at compile-time and embedded with control instructions in the code. The additional control instructions read out the runtime calculated signature from the watchdog processor and compare it to the golden signature.

According to [MM88], the first derived-signature CFC method is introduced by Namjoo [Nam82] and is called *Basic Path Signature Analysis* (Basic PSA). The signatures are calculated by XORing over all instructions inside each basic block. After the calculation of the signatures at compile-time, the signatures are stored at the beginning of each basic block. The watchdog processor monitors the instruction stream and identifies the loading of the signature from the instruction memory. During the

execution of the basic block, the watchdog processor calculates the runtime signature by XORing the processed instructions and, at the end of a basic block, the signatures are compared. A very similar technique is proposed by Sidhar and Thatte in [ST82].

Other approaches use *linear feedback shift registers* (LFSR) [DS90, DS91] or *checksums* [SM90] as signatures or try to lower the number of used signatures by using larger blocks which include multiple branches [SS87, WS90]. Gaisler enhanced his *ERC32* processor with an ESM CFC technique where the signatures are embedded into NOP instructions [Gai94]. Meixner and others store the signatures for the *Argus-1 checker* into unused instruction bits of the SPARC ISA to reduce the performance and memory overhead of their ESM method [MBS07, MBS08]. If insufficient unused bits are available, they also embed the signature into NOP instructions.

Upadhyaya and Ramamurth propose a derived-signature CFC technique using *m*-of-*n* codes [UR94]. An *m*-of-*n* code is an *n*-bit code whose bit values have *m* ones. At compile-time, the signature of a basic block is calculated, for example, by XORing the instructions. If the intermediate result is an *m*-of-*n* code, then this instruction is tagged. At runtime, the watchdog calculates the signature, recognizes the tagged instructions and verifies on the tagged instructions if the runtime signature is an *m*-of-*n* code. At the basic block borders, an additional byte is inserted which adjusts the current signature to an *m*-of-*n* code in order to force a check. The advantage is that the signature must not be stored in the program code. However, one additional byte per branch is necessary to force a check in order to restart the runtime signature calculation at a new basic block. A similar approach is presented by Ohlsson and Rimen called *Implicit Signature Checking* (ISC) [OR95]. The implicit signatures are the current start addresses of the basic blocks. This can be achieved by using additional justified signatures embedded into the code.

Autonomous Signature Monitoring The golden (compile-time calculated) signatures of *derived-signature ASM methods* are stored in a separate memory for the watchdog processor. The comparison between the golden and the runtime calculated signature is implemented in hardware (see Figure 2.8). If the control flow graph is mapped into the instruction memory of the watchdog processor, the jumps and branch destinations can also be checked. The advantages are that the program code does not need to be altered and that there is no performance impact. Also, all instructions can be monitored. The disadvantages are that extra memory is required for the checker unit and the synchronization between the CPU and the checker unit is difficult. Therefore, interrupts, multi threading, and indirect jumps cannot be covered completely.

One of the first approaches using the ASM scheme is the *Cerberus-16* watchdog processor [MM88, Nam83]. The control flow graph and the corresponding signatures are mapped in the microinstructions which are stored into the watchdog processor memory. The Cerberus-16 processor only has control flow instructions with

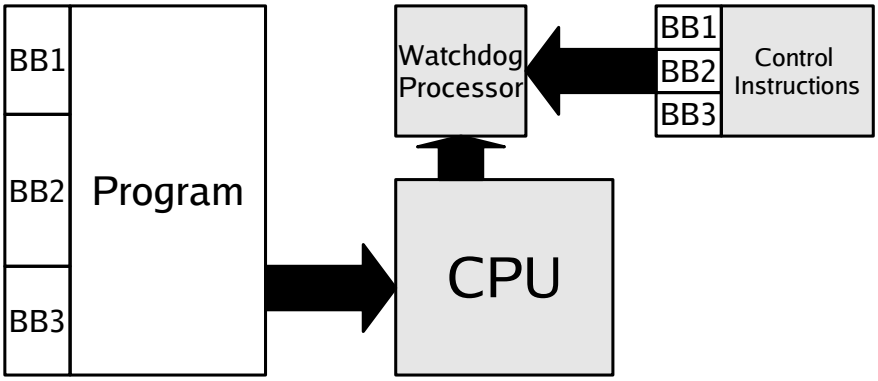


Figure 2.8: In the derived-signature ASM approach, the watchdog processor has a separate memory for storing the control instructions. The execution must be synchronized between the CPU and the watchdog processor.

encoded signatures and instructions for the communication with the main processor. The processing of the control flow of the main and the watchdog processor are completely synchronized. The approach is extended by Michel and others by a *branch address hashing* (BAH) technique, now called *Watchdog Direct Processing* (WDP) which reduces the memory overhead for the watchdog processor memory [MLS91].

An asynchronous ASM approach is presented by Eifert and Shen [ES84, ST87] which is called *Roving Monitor Processor* (RMP). At compile-time, the control flow of the program is extracted and the signatures (CRC values) are calculated. During the execution, the runtime signatures are calculated with an additional signature generation unit and, at the end of a block, the calculated signature is sent to the watchdog processor. The watchdog compares the received signature to the signatures achieved from the control flow graph and stores then in the watchdog memory. At branches, the received signature is compared with the two successor signatures of the current node in order to determine the next signature. This approach can also be used to check multi-processor systems where each processor has its own signature generation unit and sends the signature via a signature queue to the watchdog processor which is responsible for the whole system. The approach is extended by Madeira and Silva who introduce the *Online Signature Learning and Checking* (OSLC) technique. In this approach, the golden signature is generated during a learning phase [MS91]. The learned signatures are stored in the watchdog memory of an RMP-like watchdog processor.

Arora and others describe an ASM approach for security applications in [ARRJ06]. This hardware approach consists of three parts: the *Inter-Procedural CFC*, the *Intra-Procedural CFC*, and the *Instruction Stream Integrity Checker*. The Inter-Procedural CFC verifies the function calls and returns by implementing the function call graph

in hardware using *content addressable memories* (CAMs) and an FSM. The Intra-Procedural CFC checks the basic blocks by a compile-time generated control flow graph, implemented in checker memories. Finally, the Instruction Stream Integrity Checker is similar to those of other ASM methods, however, they use hash functions to generate and verify the signatures.

Our new control flow checking approach introduced in Section 4 belongs to the class of *derived-signature ASM methods*. We propose a term *checker unit* for the watchdog processor, because our unit is very simple with only few hardware overhead. However, like in the Cerberus-16 or the WDP approach, the control flow graph is mapped into microinstructions which are stored in a separate memory. Further advantages of our control flow checker are the fast error detection due to the tight integration into the processor, the error recovery possibility, and the expandability with modules which support more control flow instructions or detect more errors. Moreover, like the other ASM-methods we have no performance impact on the error-free case and the program must not be altered.

2.5 Summary

This chapter gives an overview of methods and techniques to increase the security and reliability of embedded systems. The focus of this survey is on IP protection and control flow checking which correlates with the following chapters. In addition to the introduced methods, there exist security and reliability frameworks which combine many of these techniques, for example, the RSE framework [NKIX04] or the Argus checker [MBS08]. Nevertheless, this survey is far away from being complete, rather its purpose is to give an impression of the different research areas of this topic. The relevance increases more and more with the ongoing growing of complexity due to the advances in technology.

3 IP Core Watermarking and Identification

In this chapter, methods for IP core watermarking and identification as well as their implementations are presented. First, we give an introduction to the topic and state the goals of the proposed methods. The following section presents a theoretical model for watermarking and identification of IP cores. We proceed by introducing methods for extracting a watermark and verifying the authorship by IP core identification using an FPGA bitfile. The subsequent section explains the extraction of a watermark by analyzing the power consumption of the FPGA. After that, we provide experimental results for all introduced methods. In conclusion, the contributions will be summarized.

3.1 Introduction

The ongoing miniaturization of on-chip structures allows us to implement very complex designs which require very careful engineering and an enormous effort for debugging and verification. Indeed, complexity has risen to such enormous measures that it is no longer possible to keep up with productivity demands if all parts of a design must be developed from scratch. In addition, the very lively market for embedded systems with its demand for very short product cycles intensifies this problem significantly. A popular solution to close this so called *productivity gap* is to reuse design components that are available in-house or that have been acquired from other companies. The constantly growing demand for ready to use design components,

also known as IP cores, has created a very lucrative and flourishing market which is very likely to continue its current path not only into the near future.

One problem of IP cores is the lack of protection mechanisms against *unlicensed usage*. A possible solution is to hide a unique *signature* (*watermark*) inside the core. However, there also exist techniques where an IP core can be identified without an additional signature. *Identification* methods are based on the extraction of unique characteristics of the IP core, e.g., lookup table contents for FPGA IP cores. With these techniques, the author of the core can be identified and an unlicensed usage can be proven. In this chapter, watermarking as well as identification techniques for IP cores will be presented.

Our vision is that unlicensed IP cores, embedded in a complete SoC design which could be further embedded into a product, can be detected solely by using the given product and information from the IP core developer. Information of the accused SoC developer or product manufacturer should not be necessary and no extra information should be required from the accused company. Obviously, such concepts need advanced verification techniques to detect a signature or certain IP core characteristics, present in one of many IP cores inside a system. Furthermore, the embedded author identification should be preserved even when the IP cores pass through different design flow steps. On the one hand, we must deal with the problem that design tools might remove the signature or the characteristics during synthesis and optimization. On the other hand, we must also secure the signature or characteristics against the removal by pirates which do not want the IP core be identifiable. Licensing IP cores is a market where a lot of money is involved and if a company decides to use unlicensed cores, e.g., to lower the production costs, they usually have very high skilled employees who might try to remove or bypass the watermark or identifying techniques.

In Figure 3.1, a possible watermarking flow is depicted. An IP core developer embeds a signature inside his core using a *watermark embedder* and sells the protected IP core. A third-party company may obtain an unlicensed copy of the protected IP core and use it in one of their products. If the IP core developer becomes suspicious that his core might have been used in a certain product without proper licensing, he can simply acquire the product and check for the presence of his signature. If this attempt is successful and his signature presents a strong enough proof of authorship, the original core developer may decide to accuse the product manufacturer of IP fraud and press legal charges.

An extensive survey regarding existing watermark techniques was already given in Section 2.1. The major drawback of these approaches is the limitation of the verification possibilities of the watermarked core. Our verification strategy solely requires the fishy product and no additional information from the producer.

IP cores exist for all design flow levels, from *HDL cores* at RTL to *bitfile cores* for FPGAs or *layout cores* for ASIC designs at device level (see Section 1.2.7). In the future, IP core companies will concentrate more and more on the flexible *HDL*

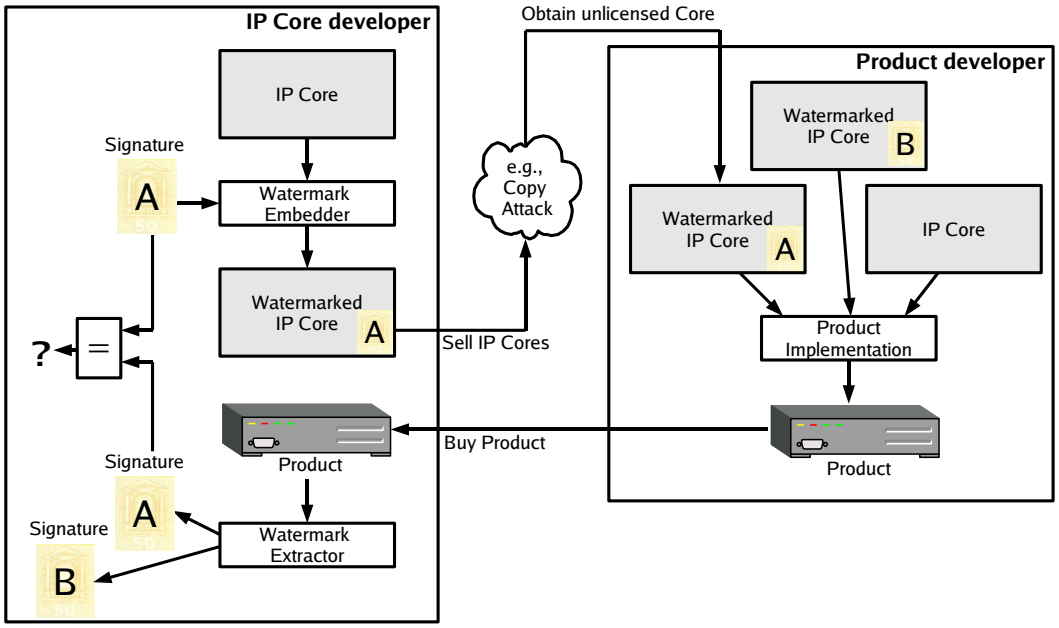


Figure 3.1: This figure shows a typical watermarking flow: An IP core developer embeds a watermark A inside his core. If a product developer obtains an unlicensed core and embeds this core in his product, the IP core developer can buy this product and extract the watermarks of all used IP cores. Now, he is able to compare his signature with the extracted signatures.

and *netlist cores*. One reason for this development is that these cores can be easily adapted to new technologies and different FPGA devices.

This chapter focuses on watermarking and identification methods for IP cores implemented on FPGAs. These have a huge market segment and the inhibition threshold for using unlicensed cores is lower than in the ASIC market where products are produced in high volumes and millions for mask production are spent. Nevertheless, some of these methods can be adapted for the ASIC design flow as well. Moreover, we concentrate on flexible IP cores which are delivered at RTL or logic level in HDL or netlist format. The advantage is that these cores can be used in different FPGA devices and can be combined with other cores to provide a complete SoC solution. Most of the existing watermarking techniques do not cover the area of HDL or netlist cores, or are not able to easily extract the signature from a heterogeneous SoC implementation in a given product.

The problem of applying watermarking techniques to FPGA designs is not the coding and insertion of a watermark, rather the verification with an FPGA embed-

ded in a system. Hence, our methods concentrate particularly on the verification of watermarks. In general, there are five potential sources of information:

- Bitfile,
- Ports,
- Power,
- Electromagnetic (EM) radiation, and
- Temperature.

The bitfile of an FPGA can be extracted by wire tapping the communication between the PROM and the FPGA. But some FPGA manufactures provide an option to encrypt the bitstream. The bitfile is stored in the PROM in encrypted form and will be decrypted inside the FPGA. Monitoring the communication between PROM and FPGA in this case is useless, because only the encrypted file will be transmitted. The bitfile is a proprietary format which is not documented by the FPGA manufacturers. However, it seems to be possible to read out some parts of the bitfile such as information stored in RAMs or lookup tables. This makes it possible to evaluate the lookup tables of a design. In Section 3.3, we introduce bitfile watermarking methods. Note that we use the term "*bitfile watermarking*" for extracting the watermark of the bitfile. The insertion of the watermark and the delivery of the core to the customer can also be done at the RTL or logic level. If we mean the insertion of a watermark at bitfile level, we use the term "*watermarking for bitfile cores*".

Another popular approach for retrieving a signature from an FPGA is to employ unused ports. Although this method works for top-level designs, it is impractical for IP cores, since these are mostly components that will be embedded into a design so that the ports will not be accessible any more. Due to these restrictions, we do not discuss the extraction of watermarks over output ports.

In this thesis, we present a new and unique technique to read out the signature by analyzing the power of an FPGA. We show that the clock frequency and toggling logic can be extracted from the power spectrum. The basic idea behind these techniques is thus to force a certain toggle pattern and extract this *signature* from the FPGA's power spectrum. We present the basics of these new methods with many extensions in Section 3.4. We use the term *"Power Watermarking"* to refer to collecting the signature from the FPGA's power consumption.

Detecting a signature using the *electromagnetic* (EM) spectrum uses almost the same strategy. This technique has the further advantage that a raster scan of an FPGA surface with an EM sensor can also use the location information to extract and verify the watermark. Unfortunately, more and more FPGAs are delivered in a metal chip package which absorbs the EM radiation. Nevertheless, this is an interesting alternative technique for extracting watermarks and invites for future research.

Finally, a watermark might also be read out by monitoring the temperature radiation. The concept is similar to the power or EM-field watermarking approach, however, the transmission speed is drastically reduced. Interestingly, this is the only watermarking approach which is commercially available [KMM08]. Here, reading the watermark embedded into an FPGA design may need up to 10 minutes.

3.2 Theoretical Watermark Model

In this section, we propose a *theoretical watermarking model*. With this model, different threats and attack scenarios can be described and evaluated. In general, watermarking techniques must deal with an uncontrolled area, where the watermarked work is further processed. This is true for multimedia watermarking, where, e.g., watermarked images are processed to enhance the image quality by filters or for IP core watermarking where the core is combined with other cores and traverses other design flow steps. However, the watermarked work may be exposed further to attacks in this uncontrolled area that may destroy or negate the watermark and thus the proof of authorship as well. This uncontrolled area is difficult to describe in a precise way and therefore, the security goals and issues for watermarking are often described in natural language which results in an imprecise description. This natural description makes an assessment of the security very difficult, particularly if the attackers are intelligent and creative.

Introducing a defined theoretical watermarking model with attackers and threats allows us to assess the security of general watermarking techniques. However, it should be noted that the model has to cover all possible attack scenarios and represent all aspects of the real world behavior to allow for a meaningful assessment of the security. In this section, we present a general watermark model introduced by Li et al. [LMS06] which will be enhanced with aspects of IP core watermarking.

3.2.1 General Watermark Model

Watermarking intellectual property can be specified precisely by characterizing the involved actions using a security model. We use the standard definitions from security theory, which defines security goals, threats and attacks. Security goals represent certain abilities of a scheme, which are important to protect in order to keep its functionality in tact. These abilities may be violated by threats which are realized by attacks. Regarding watermarking, the overall security goal is to be able to present a *proof of authorship* that is strong enough to hold in front of a court. The security goal of a watermark scheme is violated if the original author cannot produce a strong enough proof of authorship, so that a dispute with another party will lead to an *ownership deadlock*, but also in the occasion, that another party is able to present a more convincing proof of authorship than the original author, resulting in *counterfeit*

ownership. Another violation of the proof of authorship occurs if the watermark of a credible author is forged by another author and is used to convince a third party, that a work was created by someone who did not.

An attacker can realize an ownership deadlock, if he can present a watermark in the work, that is at least as convincing as the original authors watermark. If such an *ambiguity attack* is successful, the real ownership cannot be decided and the original cannot prove his authorship. If, in addition, the ambiguity attack results in the pirate being able to present an even more convincing proof of authorship than the creator of the work, the pirate can counterfeit the ownership. Another way to take over the ownership of a piece of IP is to be able to remove the original authors watermark by means of a *removal attack*. Forged authorship can be achieved by a *key copy attack* which simply duplicates the means of creating a credible authors watermark. One last violation of the security goal does not directly involve the author, but requires him to not take part in a dispute over *theft*. The theft of a work resulting in counterfeit ownership can be realized by a *copy attack*. The realized threat is only successful until the original author realizes the violation. An overview of the introduced terms can be observed in Figure 3.2.

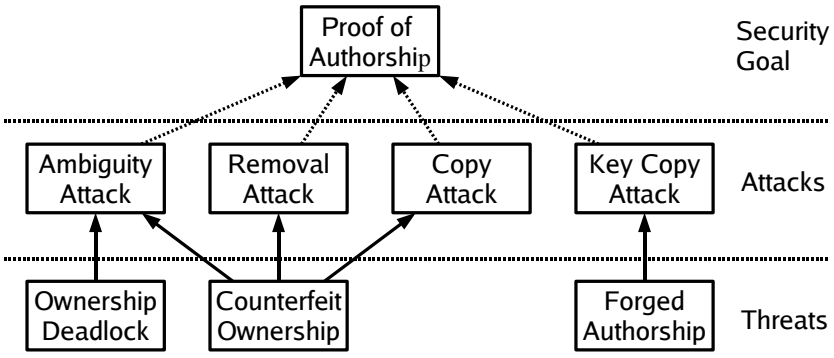


Figure 3.2: An overview of threats, attacks and the watermarking security goal of the proof of authorship. The different threats are realized by attacks which violate the security goal.

To explain all threats and attacks in detail, some definitions have to be made first [LMS06].

Definitions

A work I is defined as a vector $I = (x_1, x_2, \ldots, x_n)$, where each element $x_i \in \mathcal{I}$ resides in a universe $\mathcal{I}$. The universe $\mathcal{I}$ depends of the kind of the work. Let $Dist(\cdot, \cdot)$ be a distance function which is able to measure the differences of two works. A watermark

W is a vector $W = (w_1, w_2, \ldots, w_l)$, where each element $w_i \in \mathcal{W}$. The universe $\mathcal{W}$ is dependent on the universe of the work $\mathcal{I}$ and the watermark generation process. A key K is a sequence of m binary bits ($K = \{0,\ 1\}^m$).

In the general watermark model, there exist three algorithms: the *watermark generator* $\mathcal{G}$, the *watermark embedder* $\mathcal{E}$, and the *watermark detector* $\mathcal{D}$. The watermark generator $\mathcal{G}$ is able to generate a watermark W in the universe $\mathcal{W}$ for the key K: $W = \mathcal{G}(K)$. The watermark embedder $\mathcal{E}$ embeds this watermark W into the work I. The output is the watermarked work $\widetilde{I} = \mathcal{E}(I, W)$. The watermark in $\widetilde{I}$ should obviously not be visible. Therefore, the difference between I and $\widetilde{I}$ should be small. With the distance function, this can be expressed as $Dist(I, \widetilde{I}) < t_I$, where t_I is a threshold value upon the difference is noticeable. Using the watermark detector $\mathcal{D}$, the existence of the watermark W in the work $\widetilde{I}$ can be proven, if $\mathcal{D}(\widetilde{I}, W) = true$ or negated if $\mathcal{D}(\widetilde{I}, W) = false$.

This watermarking model may exist in different variations. The detection may be achieved with a watermarking extractor $\mathcal{X}$ instead of the watermark detector $\mathcal{D}$. The watermark extractor $\mathcal{X}$ uses the watermarked work $\widetilde{I}$ as input and the extracted key K_X as output: $K_X = \mathcal{X}(\widetilde{I})$. If no watermark is present in the work $\widetilde{I}$, the watermark extractor indicates this conveniently. Another extractor variant needs the watermarked and the original work as input: $K_X = \mathcal{X}(I, \widetilde{I})$. The extracted key K_X can now be compared with the authors key K to establish the ownership.

Threat Model

The different threats against the security goal to establish the authorship of a work can be explained using attack examples. Let an author create a work I_A and a watermark W_A using his key K_A with the watermark generator $\mathcal{G}$: $W_A = \mathcal{G}(K_A)$. The author's work is now watermarked with the watermark embedder $\mathcal{E}$: $\widetilde{I_A} = \mathcal{E}(I_A, W_A)$. The watermarked work $\widetilde{I_A}$ is published. The work can be modified by third parties, e.g., by filtering a watermarked image, so several variants $\widehat{I}_{A_x}$ of $\widetilde{I_A}$ may exist. For all variants, it is necessary that the difference to $\widetilde{I_A}$ is smaller than a specified threshold t_I: $Dist(\widehat{I}_{A_x}, \widetilde{I_A}) < t_I, \forall x$. This condition is necessary, because if a work is completely or exhaustively altered, the relation to $\widetilde{I_A}$ is weak, and the work can be counted as a new work. Let $\widehat{I}$ be a published work where the authorship is to be proven. We propose different scenarios where an attacker or pirate tries to sabotage the proof of authorship (see also Figure 3.2):

The first scenario is **counterfeit ownership** using a **removal attack**. The pirate obtains the watermarked work $\widetilde{I_A}$ from the author or any variant of it ($\widehat{I}_{A_x}$) and manages to remove the watermark and transforms the obtained worked into a work without or with a disabled watermark $\widehat{I}$. If the author of the work tries to proof his ownership of the work $\widehat{I}$, he fails:

$$Owner: \quad Dist(\, I_A\, ,\, \widehat{I}\,) < t_I \quad \mathcal{D}(\widehat{I}, W_A) = false$$

Furthermore, the pirate can embed his watermark W_P into the work to take over the ownership: $\widetilde{I_P} = \mathcal{E}(\widehat{I}, W_P)$. For the proof, he presents the work with the removed watermark $\widehat{I}$ as original work.

$$Pirate: \quad Dist(\,\widehat{I}\,,\,\widetilde{I_P}\,) < t_I \quad \mathcal{D}(\widetilde{I_P}, W_P) = true$$

The next scenario is **counterfeit ownership** by using a **copy attack** where a work is copied without the permission from the author. To realize a counterfeit ownership with a simple copy attack, the original author must be passive, either if he publishes a work which is not watermarked, or publishes a watermarked work, but does not take not part in the dispute. In both cases, the pirate can embed his watermark W_P into the work $\widehat{I}$.

$$Pirate: \quad \widetilde{I_P} = \mathcal{E}(\widehat{I}, W_P) \quad Dist(\,\widehat{I}\,,\,\widetilde{I_P}\,) < t_I \quad \mathcal{D}(\widetilde{I_P}, W_P) = true$$

If the author had not watermarked the work, he has no possibility to proof his authorship, whereas otherwise, by detecting his watermark in $\widehat{I}$ and $\widetilde{I_P}$, he can establish the authorship. Here, the author has shown that his watermark W_A is present in the fake original work from the pirate $\widehat{I}$, whereas the pirate's watermark W_P is not in the original work I_A of the author. This proves that the pirate has taken a watermarked work from the author and embedded his watermark in it.

$$\begin{aligned} Owner: &\quad Dist(\,I_A\,,\,\widetilde{I_P}\,) < t_I \quad \mathcal{D}(\widetilde{I_P}, W_A) = true \quad \mathcal{D}(\widehat{I}, W_A) = true \\ Pirate: &\quad Dist(\,\widehat{I}\,,\,\widetilde{I_P}\,) < t_I \quad \mathcal{D}(\widetilde{I_P}, W_P) = true \quad \mathcal{D}(I_A, W_P) = false \end{aligned}$$

In the next scenarios, the pirate analyzes the work $\widehat{I}$ in order to find a fake watermark W_P and a corresponding fake original work I_P. This leads to an **ownership deadlock** by using an **ambiguity attack**. Both parties can show that the work $\widehat{I}$ is watermarked with their corresponding watermarks and furthermore, the own watermark is present in the original work of the opponent. This leads to an ownership deadlock, because the rightful owner cannot be chosen.

$$\begin{aligned} Owner: &\quad Dist(\,I_A\,,\widehat{I}\,) < t_I \quad \mathcal{D}(\widehat{I}, W_A) = true \quad \mathcal{D}(I_P, W_A) = true \\ Pirate: &\quad Dist(\,I_P\,,\widehat{I}\,) < t_I \quad \mathcal{D}(\widehat{I}, W_P) = true \quad \mathcal{D}(I_A, W_P) = true \end{aligned}$$

Moreover, if the pirate can achieve that in his fake original work I_P the watermark of the author W_A cannot be found, he can take over the ownership. This **ambiguity attack** leads to a **counterfeit ownership**.

$$\begin{aligned} Owner: &\quad Dist(\,I_A\,,\widehat{I}\,) < t_I \quad \mathcal{D}(\widehat{I}, W_A) = true \quad \mathcal{D}(I_P, W_A) = false \\ Pirate: &\quad Dist(\,I_P\,,\widehat{I}\,) < t_I \quad \mathcal{D}(\widehat{I}, W_P) = true \quad \mathcal{D}(I_A, W_P) = true \end{aligned}$$

Forged authorship can be achieved by **key copy attacks**. Here, the key K_A of a credible author is stolen, and a fake watermark W'_A is generated. With this watermark,

a work with lower quality I_P is marked. The pirates goal is that everyone believes that this poor work is from the credible author which might raise the value of his own work.

$$Pirate: \quad W'_A = \mathcal{G}(K_A) \quad \widetilde{I_A}' = \mathcal{E}(I_P, W'_A) \quad \mathcal{D}(\widetilde{I_A}', W'_A) = true$$

In the following, some security properties of a watermark scheme will be analyzed with respect to the attacks and threats introduced above. Let attacker $\mathcal{A}$ be any algorithm of polynomial complexity.

Definition 3.1 *A watermark scheme is t_I-resistant to removal attacks if for any attacker $\mathcal{A}$ and given any work $\widetilde{I}$ watermarked by W, it is computationally infeasible for $\mathcal{A}$ to compute any work I' such that $Dist(\widetilde{I}, I') < t_I$ and $\mathcal{D}(I', W) = false$ [LMS06].*

The term *t_I-resistant* means that the watermark scheme is resistant against removal attacks with respect to the threshold value t_I. If the distance exceeds t_I, the works cannot be counted as identical. For example, if the attacker creates a completely new work, the watermark is also removed, but the works are not the same. The phrase *computationally infeasible* follows the standard definition from cryptography. Something is computationally infeasible if the costs (e.g., memory, runtime, area) is finite but impossibly large [DH76]. Here, this is true if the probability $Pr[\mathcal{A}(\widetilde{I}) = I']$ is negligible with respect to the problem size n. A quantity X is negligible with respect to n if and only if for all sufficiently large n and any fixed polynomial $q(\cdot)$ (the attacker $\mathcal{A}$ is defined as an algorithm of polynomial complexity), we have $X < 1/q(n)$ [LMS06].

In other words, with a sufficiently large problem size of watermarked work $\widetilde{I}$, resistance against removal attacks means that the attacker is unable to remove the watermark as the problem size is beyond the computational capability of the attacker, unless the resulting work is perceptually different from the original work.

If we consider ambiguity attacks where a possible attacker finds a fake watermark inside a watermarked work, we come to the following definition:

Definition 3.2 *A watermark scheme is resistant to ambiguity attacks if for any attacker $\mathcal{A}$ and any work $\widetilde{I}$, it is computationally infeasible for $\mathcal{A}$ to compute a valid watermark W_P such that $\mathcal{D}(\widetilde{I}, W_P) = true$ [LMS06].*

A watermark scheme cannot be resistant against copy attacks, because the watermark scheme cannot distinguish if the work which is used as original work is really the work of the person which is identified by the key.

In case of key copy attacks, the key of a credible author is used to watermark a work with lower quality. In general, it should be impossible for an attacker to create a work I' which is distinguishable from any work of another author where the key or watermark of a credible author can be found.

Definition 3.3 *A watermarking scheme is t_I-resistant to key copy attacks if for any attacker $\mathcal{A}$ and any work $\widetilde{I} = \mathcal{E}(I,W)$ for some original I and the watermark W, it is computationally infeasible for $\mathcal{A}$ to compute a work I' such that $Dist(I,I') > t_I$, yet $\mathcal{D}(I',W) = true$ [LMS06].*

To prevent key copy attacks, a private/public key algorithm, like RSA [RSA78] can be used. RSA is an asymmetrical cryptography method. It is based on factorization of a number into prime numbers. The author encrypts a message which clearly identifies the author and the work with his private key. The work can be identified by a hash value over the original work. This encrypted message is now used for generating the watermark and embedded inside the work. Stealing this watermark is useless, because everyone can decrypt the message with the public key, whereas no one can alter this message.

3.2.2 IP Core Watermark Model

Watermarking IP cores in electronic design automation is in some aspects different from multimedia watermarking (image, audio, etc.). An essential difference is that watermarking should preserve the functionality of the core. Another difference is that IP cores can be distributed at different abstraction levels which have completely different properties for the watermark security against attacks. We define different design steps as different technology or abstraction levels a work or IP core can be specified on (see Section 1.2.7). On higher abstraction levels, e.g., the architecture level or RTL, the functionality is described by an algorithm which should be implemented. At these levels, mainly the behavior is described and the representation is optimized for easy reading and understanding the algorithm. During the course of the design flow, more and more information is added. For example, at the device level also placement information is included in the representation of the core. Extracting only the relevant information about the behavior of the algorithm is much harder than at higher abstraction levels. Furthermore, the information at lower abstraction levels is usually interpreted by tools rather than humans. The representation of this information is therefore optimized for tools and not for human readability. For example, consider an FPGA design flow. Here, three different abstraction levels exist: *RTL, logic*, and *device level*. An algorithm, specified at the register-transfer-level (RTL) in an HDL core is easier to understand than a synthesized algorithm at the logic level, represented by a netlist. In summary, we can say that the behavior of an algorithm is easier to understand on higher abstraction levels than on the lower ones.

Transformations from a higher to a lower abstraction level are usually done by *design tools*. For example, a synthesis tool is able to transform a HDL core specified at the register-transfer-level (RTL) into its representation on the logic level. Transformations from a lower to a higher level can be achieved by reverse engineering. Here, usually no common tools are available. One exception is the Java library *JBits* from

Xilinx [Xilc] which is able to interpret the bitfiles of Virtex-II device types. Thus, it is possible to transfer a bitfile core into a netlist at the logic level by using JBits. However, in general, reverse engineering must be considered as very challenging task which may cause high costs.

A watermark can be embedded at every abstraction level. Furthermore, the watermarked core can be published and distributed also at every abstraction level which must not necessarily be the same level at which the watermark was embedded. However, the extraction of the watermark is usually done in the lowest abstraction level, because this representation is implemented into the end product.

Hiding a watermark at a lower abstraction level is easier, because first, there are more possibilities of how and where to hide the watermark. Second, the information stored at these abstraction levels is usually outside the reception area of the human developer.

Definitions

Our new definitions for the IP core watermarking model [SZT08] are derived from the general watermarking model introduced in the beginning of this section. A work or IP core that is specified at abstraction level Y is denoted by $I_Y = (x_{Y_1}, x_{Y_2}, \ldots, x_{Y_m})$, where each $x_{Y_i} \in \mathcal{I}_Y$ is an element of the work, and $\mathcal{I}_Y$ is a universe, inherent to the abstraction level Y. For example, an FPGA design at the device abstraction level might be represented by a bitfile which can be characterized as a work $I_B = (x_{B_1}, \ldots, x_{B_m})$, whose elements reside in the universe *Bit* ($\mathcal{I}_B$). Hence, a bitfile I_B with $|I_B| = m$ can also be considered as a binary sequence $I_B = \{0,1\}^m$.

Let $\mathcal{T}(\cdot)$ be a transformation, which transforms a work on a specific abstraction level into a work of another abstraction level. A transformation from the higher level Y to the lower abstraction level Z is denoted $\mathcal{T}_{Y \to Z}(\cdot)$, whereas a transformation from a lower to a higher level is denoted $\mathcal{T}_{Y \leftarrow Z}(\cdot)$.

The distance function $Dist_Y(\cdot,\cdot)$ in the context of IP core watermarking is only able to compare two works of the same abstraction level. If the distance of two IP cores I_Y and I'_Y of the same abstraction level Y is smaller than a threshold value t_I ($Dist_Y(I_Y, I'_Y) < t_I$), the two works may be considered similar.

Furthermore, the watermark generator $\mathcal{G}$, embedder $\mathcal{E}$, and detector $\mathcal{D}$ must be restricted in a way that they cannot cross the abstraction level boundaries. In detail, a specific watermark generator $\mathcal{G}_X(\cdot)$ is able to generate a watermark W_X for the abstraction level X from a key K: $W_X = \mathcal{G}_X(K)$. The input of the watermark embedder or detector must be in the same abstraction level. For example, to watermark an IP core I_X at abstraction level X, also the watermark W_X must be generated for this abstraction level. So to obtain a watermarked work on the abstraction level X, it is necessary to also use a watermarked and an embedding algorithm suitable for the same abstraction level, i.e., $\widetilde{I_X} = \mathcal{E}_X(I_X, W_X)$.

In order to achieve full transparency of the watermarking process towards design tools, it is an essential requirement that a work, marked on any abstraction level, will retain the watermark if transformed to a lower abstraction level. Hence, if $\mathcal{D}_Y(\widetilde{I_Y}, W_Y) = true$, so should also $\mathcal{D}(\widetilde{I_Z}, W_Z) = true$, if $\widetilde{I_Z} = \mathcal{T}_{Y \to Z}(\widetilde{I_Y})$, and W_Z is a representation of W_Y on abstraction level Z.

However, considering reverse engineering, the watermark information is may be removed by the reverse engineering transformation $\mathcal{T}_{Y \leftarrow Z}(\cdot)$, or the detection and removal of the watermark may be easier on the higher abstraction level. For example, consider an FPGA bitfile IP core watermarking technique for which the watermark is stored in some placement information inside the bitfile. The watermark is generated for bitfiles at device level: $W_B = \mathcal{G}_B(K)$. The watermark is embedded in a bitfile core I_B to create the watermarked bitfile: $\widetilde{I_B} = \mathcal{E}_B(I_B, W_B)$. If an attacker is able to reverse engineer the bitfile and reconstruct a netlist at logic level the placement information will get lost, since there is no representation for this on the logic level. This implies, of course, that the watermark is lost, as well: $\widetilde{I_L} = \mathcal{T}_{L \leftarrow B}(\widetilde{I_B})$, $\mathcal{D}_L(\widetilde{I_L}, W_L) = false$. Another problem of reverse engineering may be that an embedded watermark might become obviously readable at the higher abstraction level and can be removed easily.

Figure 3.3 shows an example of the IP core watermark model considering different abstraction levels.

Threat Model

The threat and attack model for general watermarking, as depicted in Figure 3.2, may remain almost the same for the IP core watermarking model. However, the resistance definitions against different attacks must be redefined. Sometimes, it might be easier for an attacker to redevelop an IP core than to remove a watermark. The question to purchase or to redevelop a core is a pure matter of cost. An uprising economical question is whether the development of an attack is an option. For many cases, the redevelopment from scratch might be cheaper than obtaining an unlicensed core and develop an attack in order to remove the watermark. On the other hand, there are designs involving such cunning cleverness and creativity that trying to redevelop a work of equivalent economic value would exceed the costs of developing an appropriate attack by several orders of magnitude.

In the general watermarking model, it should be computationally infeasible to remove the watermark without changing the properties of the work. For the introduced IP core watermarking model, this requirement does not necessarily hold. We may consider a watermarking technique secure, if the cost for obtaining an unlicensed IP core and developing a removal attack is higher than to purchase the IP core.

Let the attacker $\mathcal{A}_Y$ be an algorithm which is able to transform a watermarked IP core $\widetilde{I_Y}$ at the abstraction level Y into an IP core with removed or disabled watermark $I'_Y = \mathcal{A}_Y(\widetilde{I_Y})$. Let $C(\cdot)$ be a cost function. Furthermore, let $C_D(\cdot)$ denote the development cost of a specified IP core or attack and $C_P(\cdot)$ the purchase cost of an

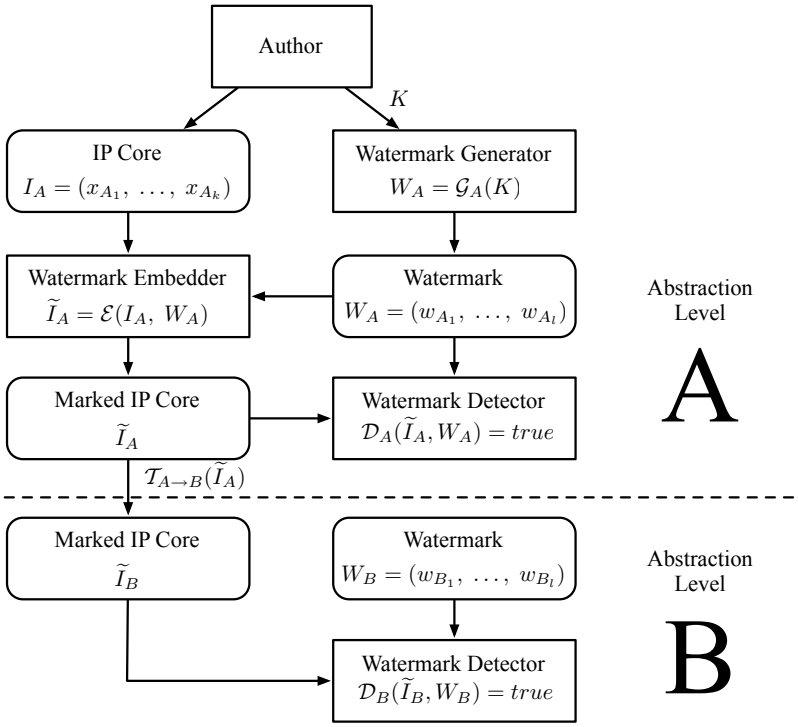


Figure 3.3: An example of a watermarking procedure characterized in the IP core watermark model with different abstraction levels. At abstraction level *A*, the watermark is generated and embedded. A transformation to the abstraction level *B* retains the watermark [SZT08].

IP core. Let $C_O(\cdot)$ denote the cost to obtain an (unlicensed) IP core. Note that this cost may vary between the costs for copying the core from an arbitrary source and those for purchase it. We define a watermarked core $\widetilde{I_Y}$ to be secure against attacks if attacks produce higher costs than the legal use of the core. Instead of requiring computational infeasibility, it is enough to fulfill:

$$C_P(\widetilde{I_Y}) < C_D(I_Y) \leq (C_O(\widetilde{I_Y}) + C_D(\mathcal{A}_Y(\widetilde{I_Y})). \tag{3.1}$$

Furthermore, a reverse engineering step to a higher abstraction level and the development of an attacker algorithm on this level might be cheaper than the development of an attacker algorithm on the lower abstraction level. Therefore, we must also consider the usage of reverse engineering:

$$C_P(\widetilde{I_Y}) < C_D(I_Y) \leq (C_O(\widetilde{I_Y}) + C(\mathcal{T}_{X\leftarrow Y}(\widetilde{I_Y})) + C_D(\mathcal{A}_X(\widetilde{I_X})) + C(\mathcal{T}_{X\to Y}(I'_X)). \tag{3.2}$$

Definition 3.4 *An IP core watermarking scheme is called t_I-resistant to removal attacks if for any attacker $\mathcal{A}$ and any IP core $\widetilde{I_Y}$ of a given abstraction level Y and watermarked by W_Y, it is either computationally infeasible to compute $I'_Y = \mathcal{A}(\widetilde{I_Y})$*

with $Dist_Y(\widetilde{I}_Y, I'_Y) < t_I$ *and* $\mathcal{D}_Y(I'_Y, W_Y) = false$ *or produces higher costs than its legal use.*

For ambiguity attacks where an attacker tries to counterfeit the ownership or to achieve an ownership deadlock, the attacker searches for a fake watermark inside the IP core. This can be done by analyzing the IP core and searching for a structural or statistical feature which might be suitable to interpret as a fake watermark. However, the published IP core may be delivered in different target technology versions, e.g., for an ASIC design flow or different FPGA target devices. This fake watermark should also be present in any other distributed sample to guarantee the attacker's authentic evidence of ownership. Furthermore, the attacker must present a fake original work, and the evidence of a comprehensible watermark generation from a unique key, which clearly identifies the attacker. These are all reasons, why ambiguity attacks are very difficult in the area of IP core watermarking.

Definition 3.5 *An IP core watermarking scheme is called resistant to ambiguity attacks if for any attacker* $\mathcal{A}$ *and any given IP core* $\widetilde{I}_Y$ *of a certain abstraction level* Y *and watermarked by* W_Y*, it is either computationally infeasible to compute a valid watermark* W'_Y *such that* $\mathcal{D}_Y(\widetilde{I}_Y, W'_Y) = true$ *or produces more costs than its legal use.*

For copy attacks and key copy attacks, the threat model proposed for the general watermarking model remains the same for the IP core watermarking model. Hence, these attacks are independent of the actual watermarking scheme. We have seen for the general watermarking model that cryptographic methods are suitable to prevent key copy attacks. Many different approaches exist to increase the confidence of a signature or watermark by using cryptographic methods. In the following an example taken from [QP03] is given for the usage of cryptographic methods for IP core watermarking.

Example Cryptographic Preparation of a Watermark

A signature may be a short ASCII-text which identifies the owner of the core, for example: "Hardware-Software-Co-Design, CS 12, University of Erlangen-Nuremberg". First, we hash the string to get a white spectrum of the probabilities of occurrence of each letter (see Figure 3.4). A usual one-way-hash function is MD5 [Riv92a]. MD5 generates a 128 Bit hash value with non-linear functions from any long message. The next step is to encrypt the hashed string. This may be done using a private/public key algorithm like RSA [RSA78]. To watermark a design, we need information where the watermark will be inserted and about the content of the watermark. For the watermarking methods according to [QP03], this information comes from a pseudo random sequence produced from a seed which is the encrypted and hashed signature with a pseudo random generator like RC4 [Riv92b]. RC4 gener-

ates a pseudo random sequence of an initial key. The key length is variable. The interpretation of this sequence is up to the used watermarking algorithm.

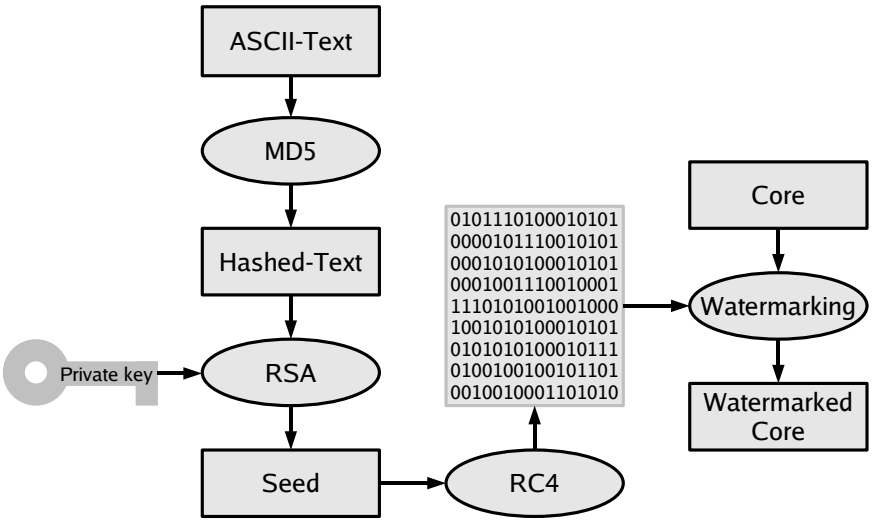


Figure 3.4: The preparation process of a watermark with cryptographic methods according to [QP03]. An ASCII text which clearly identify the author is hashed and encrypted with an public/private cryptographic method. The encrypted key is used as a seed for a pseudo random generator which result is used for watermarking the core.

To verify the signature, we extract the watermark from the design and rebuild the pseudo random sequence. We produce a second random sequence from our original seed, the encrypted and hashed signature. Now, we have to prove that the two sequences are identical. After this is done, we must also check if the seed corresponds to our signature. The seed is decrypted by our public key and we get the hash value. If this hash is equal to the hash of our signature, the verification of the signature is successful.

The application of cryptographic methods for watermarking is not a must but can make it resistant against tampering and key copy attacks.

3.2.3 IP Core Identification Model

Another possibility to establish the ownership of an IP core is by identifying it in a complete design. In this approach, no signature or watermark is necessary. Instead, let an IP core I_{X_1} for a specific abstraction level X be registered by the author and then published. The registration can be made at a trusted third party institution. The IP core customers can purchase the core I_{X_1} and combine it with other self-written

or purchased IP cores $I_{X_2}, I_{X_3}, \ldots, I_{X_m}$ to a complete design which traverses further design steps to lower abstraction levels: $I_Y = \mathcal{T}_{X \to Y}(I_{X_1} \circ I_{X_2} \circ I_{X_3} \circ \cdots \circ I_{X_m})$. The $\circ$ operator combines different cores to one design. If a company is suspected to use an unlicensed core, their design can be analyzed in a way that the unlicensed IP core usage can be detected. This can be done by an IP core detector $\mathcal{D}$ which is able to detect the published IP core in a complete design: $\mathcal{D}(I_Y, I_{X_1}) = true/false$.

The threat and attack model against proof of authorship using IP core identification methods differ slightly from the threat and attack model when using watermarked IP cores. An attacker can try to achieve an *ownership deadlock* by using an ambiguity attack. He can claim that an identified IP core I_{X_1} belongs to him. To prove his claim, he can further present the original IP core I_{X_1} which he used to build his complete design. This core can be obtained or purchased from the IP core developer or any other source (e.g., legal customers of the core which copy the core). Since the IP core I_{X_1} has no signature, the real author cannot be determined. Only by using a trusted third party institute, the real owner of the IP core I_{X_1} can be determined. Here, it is of utmost importance that the real owner of the core registered the IP core first. Usually, this should happen before the IP core is published. Furthermore, the attacker can take over the ownership of the core, if the original author forgot the registration. If the same IP core is registered at different trusted third party institutes, the date of filing should solve the dispute.

Furthermore, an attacker can *counterfeit the ownership* if he embeds a watermark inside a copied core which identifies him as the legal author. This copy attack can be prevented if the core was previously registered at a trusted institution by the rightful author. On the other hand, if an attacker registers a copied watermarked core at a trusted third party institution at which the work was not registered before, the establishment of the ownership can be difficult. Because of the registered core carries already the watermark, a judge would properly decide for the author of the watermark. This shows us that registration of an IP core at a trusted third party institute, even if it is watermarked correctly, elevates the chances of the rightful proof of authorship.

In summary, it can be said that the clear proof of authorship using any identification method can only be done if the IP core is registered. For watermarking methods, this is not mandatory. However, registration can increase the trust into the ownership proof significantly.

3.3 Bitfile Watermarking and Identification

This section explains methods where the verification is done by *extracting an FPGA bitfile*. The bitfile can be analyzed to detect structures that can carry a watermark or that can be used to identify an IP core. In this work, *lookup table contents* are used which are excellently suitable for watermarking and IP core identification.

First of all, the extraction of the lookup table contents from an FPGA bitfile is discussed. This step is necessary for all following IP core identification and bitfile watermarking methods. Afterwards, methods to identify a registered IP core in an FPGA bitfile are proposed. These include identifying netlist cores as well as HDL cores. Following, watermarking methods for netlist and bitfile cores are proposed. The focus of these watermarking methods lies on the usage of *functional lookup tables* in order to increase the robustness against removal attacks. The term *functional lookup table* refers to lookup tables which are already used in a given (non-watermarked) IP core and represent a part of the functional logic of the core which may not be removed by an attacker in order to retain the correctness of the core.

3.3.1 Lookup Table Content Extraction

In this section, we discuss which possibilities exist to get information about the contents of lookup tables from a product. First, we need to extract the *configuration bitfile* of the FPGA from the product. On some devices, it is possible to read back the bitfile. This is the easiest way, but it is not always possible, because not all FPGA devices support this feature, or it might have been disabled by the creator. In SRAM based FPGAs, the bitfile is stored into a PROM and during the startup phase, the FPGA is configured by loading this bitfile. The communication between the FPGA and the PROM can be recorded by wire tapping and so, the bitfile can be obtained.

The extraction of the lookup table contents from the configuration bitfile depends on the FPGA device and the FPGA vendor. Xilinx provide the Java library JBits [Xilc] for Virtex-II FPGA configuration bitfiles. Although, all other Xilinx devices must be reversed engineered in order to obtain any information of their configuration bitfiles.

To read out the LUT content directly from the bitfile, it must be known at which position in the bitfile the lookup table content is stored and now these values must be interpreted.

We have applied standard black-box reverse engineering procedures to Xilinx Virtex-II and Virtex-II Pro bitfiles. These bitfiles are structured in packets, where each packet consists of one or more configuration words for a certain configuration register which controls the configuration process [Xil05]. One large packet contains the configuration data for the circuit, which should be instantiated on the FPGA. This packet is divided into frames which are the smallest configuration unit. The length of one frame and the number of frames is device-dependent. There exist six frame types: IOB, IOI, CLB, BRAM, BRAM Interconnect, and GCLK. In *IOB (Input/Output Blocks) frames*, the information about the right and left Input and Output Blocks is stored. The information about the upper and lower IOBs is stored in the *CLB (Configurable Logic Block) frames*. On the left and on the right side of the FPGA exists one column of *IOB frames*; each IOB column consists of four frames. The *IOI (Input/Output Interconnect) frames* are next to the IOB frames. Here, the routing

information of the interconnect network for the IOBs is stored. There are also 22 IOI frames on each side of the FPGA. In the CLB frames, information about the CLBs, routing and the upper and lower IOBs are stored. The FPGA consists of a regular pattern of CLB columns. Each CLB column consists of 22 frames. In two of these frames, the contents of the lookup tables are stored. In *BRAM (Block Ram)* and the *BRAM Interconnect frames*, the content and the configuration of the block rams as well as the configuration of the hardware multiplier and the interconnect network is deposited. Each BRAM column consists of 64 frames and each BRAM Interconnect column consists of 22 frames. Each FPGA has one *GCLK (Global Clock)* column, consisting of four frames. Here, the information of DCMs (Digital Clock Manager), clock buffers and most of the clock routing is stored. More information can be found in [Xil05].

Each frames of the configuration memory in the FPGA can be accessed by an individual address. The address word contains a block address and a major and a minor address. The IOB, IOI, GCLK and CLB frames are stored in the block with address 0, whereas the BRAM frames are stored in the block with address 1. Block 2 consists of the BRAM Interconnect frames. The major address directs the columns whereas the minor address directs the frames in a column. Figure 3.5 shows the configuration map of a Xilinx Virtex-II or Virtex-II Pro FPGA. This is also the order in which the frames are stored in the configuration packet in the bitfile.

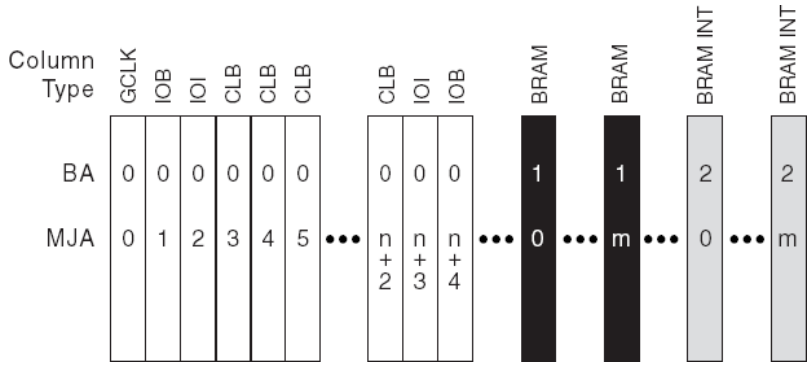


Figure 3.5: The map of the configuration memory from Xilinx Virtex-II and Virtex-II Pro FPGAs [Xil05].

A Xilinx Virtex-II CLB consists of four slices, two horizontal and two vertical. Each slice has two lookup tables, the *G*- and the *F-LUT*. The content of the lookup tables for one slice column is stored in one frame, so two frames are used for the lookup table content in a CLB column. In the second frame of a CLB column, the lookup table content of the left slice column is stored whereas the third frame carries the content of the right slice column.

16 bits are stored in a four input lookup table. These bits are stored together in two bytes but with bit inverted values. The F and G lookup tables in a slice are separated

by one byte which is not used for storing lookup table content information. So, the lookup table content packet for one slice consists of 5 bytes. First, the G-LUT is stored, but in reverse bit order. Then, the separate byte and the F-LUT is stored. The bit order of the F-LUT is not reversed.

These packets are stored successively in the frame for the slices in one column, beginning with the slice with the highest Y coordinate and ending with the slice with the 0 coordinate (see Figure 3.6). For the upper and lower input/output blocks which are also stored in the CLB frames, the first and the last 12 bytes in the lookup table content frame are reserved. The frame length FL is:

$$FL = CY \cdot 2\ Slices \cdot 5\ Bytes + 2 \cdot 12\ IOB\ Bytes \tag{3.3}$$

where CY is the number of CLB rows in the FPGA.

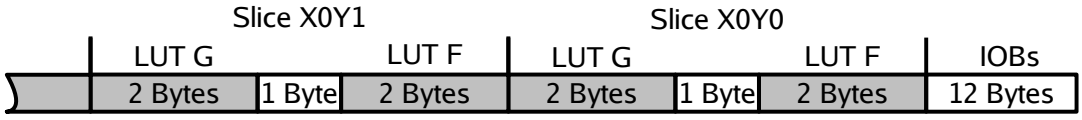


Figure 3.6: The positions of the F and G lookup table content in a frame. The gray cells denotes the lookup table contents. Note that the coordinate of the slices are in reversed order whereas the rear part of the frame is displayed.

To calculate the frame address of the lookup table frames, we can use the following formula:

$$F_{cy0} = 1 \cdot F_{GCLK} + 1 \cdot F_{IOB} + 1 \cdot F_{IOI} \tag{3.4}$$

$$f_{lf}(x_S) = F_{cy0} + CF_{lut0} + \lfloor x_S/2 \rfloor \cdot F_{CLB} + x_S\ mod\ 2 \tag{3.5}$$

where F_{cy0} denotes the frame number of the first CLB frame and F_{GCLK}, F_{IOB}, F_{IOI} and F_{CLB} identify the number of frames of a GCLK, IOB, IOI or CLB column. CF_{lut0} denotes the first lookup table content frame in a CLB. x_S and y_S denote the slice coordinates, and Y_S the number of slice rows in the FPGA. $f_{lf}(x_S)$ denotes the frame which stores the lookup table information of slice column x_S. All addresses begin with zero.

The values of F_{GCLK}, F_{IOB}, F_{IOI}, F_{CLB} and CF_{lut0} are device-independent so to provide:

$$F_{cy0} = 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 22 \tag{3.6}$$

$$f_{lf}(x_S) = F_{cy0} + 1 + \lfloor x_S/2 \rfloor \cdot 22 + x_S\ mod\ 2 \tag{3.7}$$

$$f_{lf}(x_S) = 31 + \lfloor x_S/2 \rfloor \cdot 22 + x_S\ mod\ 2 \tag{3.8}$$

To calculate the byte addresses of the lookup tables in the configuration packet of the bitfile, we can use the following formula:

$$Adr_{LUTG}(x_S, y_S) = f_{lf}(x_S) \cdot FL + 12 + ((Y_S - 1) - y_S) \cdot 5 \tag{3.9}$$

$$Adr_{LUTF}(x_S, y_S) = f_{lf}(x_S) \cdot FL + 12 + ((Y_S - 1) - y_S) \cdot 5 + 3 \tag{3.10}$$

Lookup tables in unused slices have the value `0x0000`, whereas unused lookup tables in used slices have the value `0xFFFF`. With this information, we are able to extract and decode the lookup table content and the position from used lookup tables in a Xilinx Virtex-II and Virtex-II Pro FPGA.

For generalizing this approach, we define a lookup table extractor function $\mathcal{L}_X(\cdot)$ for the abstraction level X. The extractor function is able to extract the lookup table content of a work I_X as follows: $\mathcal{L}_X(I_X) = \{x_{X_1}, x_{X_2}, \ldots, x_{X_m}\}$, whereas x_{X_i} is a lookup table content element of the abstraction level X, and m is the number of used lookup tables.

The following extraction function can be applied to extract the lookup table contents of a design I_B of the bitfile at abstraction level B: $\mathcal{L}_B(I_B) = \{x_{B_1}, x_{B_2}, \ldots, x_{B_q}\}$. Each element x_{B_i} consists of the lookup table content as well as the (x_S, y_S) coordinates of the corresponding lookup table.

3.3.2 Identification of Netlist Cores by Analysis of LUT Contents

In this approach, we do not add any signature or watermark. The core itself remains unchanged, so the functional correctness is given and no additional resources are used. We compare the content of the used lookup tables from the registered core I_{L_1} with the used lookup tables in an FPGA design I_B from the product of the accused company. If a high percentage of identical content is detected, the probability that the registered core is used is very high.

The synthesis tool maps the combinatorial logic of an FPGA core to lookup tables and writes these values into a netlist. After the synthesis step, the content of the lookup tables of a core is known, so we can protect netlist cores which are delivered at the logic level. The protection of bitfile cores at the device level is also possible.

After the core I_{L_1} is purchased, the customer can combine this core with other cores: $I_B = \mathcal{T}_{L \to B}(I_{L_1} \circ I_{L_2} \circ I_{L_3} \circ \ldots)$. In the following CLB mapping step, it is possible that lookup tables are merged across the core boundaries or are removed by an optimizing transformation. This happens when different cores share logic or when outputs of the core are not used. These lookup tables cannot be found in the FPGA bitfile I_B, but experimental results (see Section 3.5.1) show that the percentage of these lookup tables compared to the number of all lookup tables in the core is typically low for the used mapping tool (Xilinx *map*).

If a company is accused of using unlicensed cores in a product, the bitfile of the used FPGA can be extracted (see Section 3.3.1). After reading out the content and the positions of the lookup tables from the bitfile and comparing them with the lookup table contents from the original core, the ownership of the core can be proven by evaluating a detector function $\mathcal{D}(I_B, I_{L_1})$.

Identifying the Core

After the extraction of the content of lookup tables from a bitfile, we can compare the obtained values with the information in the netlist. The extraction of all lookup table contents from a bitfile is done as described in Section 3.3.1: $\mathcal{L}_B(I_B) = \{x_{B_1}, x_{B_2}, \ldots, x_{B_q}\}$. The content of the lookup tables can easily be read out from a netlist file: $\mathcal{L}_L(I_{L_1}) = \{x_{L_1}, x_{L_2}, \ldots, x_{L_r}\}$. For example, in an EDIF netlist for Xilinx FPGA devices, the lookup table contents appear after the INIT property for the lookup table instances. Unfortunately, the mapping tools do not necessarily adopt these values. The mapping tool may merge lookup tables from different cores together, convert one, two or three input lookup tables to four input lookup tables and permute the inputs to achieve a better routing.

All lookup tables of an FPGA have n_l inputs. On most FPGA architectures, lookup tables have $n_l = 4$ inputs. In a core netlist, also lookup tables with less than n_l inputs may exist. These lookup tables must be mapped onto n_l input lookup tables. If one input is unused, only half of the memory is needed to store the function and the remaining space must be filled. In the case that a function uses less inputs than the underlying technology of the FPGA provides, it is desirable to turn the unused inputs into don't cares. Intuitively, this can be achieved rather easily by replicating the function table as it is demonstrated in Figure 3.7.

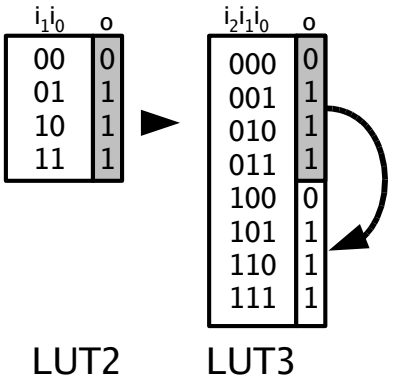


Figure 3.7: Converting a two input lookup table into a three input lookup table with unused input i_2.

The mapping tool can permute the inputs of the lookup tables, for example, to achieve a better routing. In most FPGA architectures, the routing resources for lookup table inputs are not equal, and so a permutation of the lookup table inputs can lower the amount of used routing resources. Permutation of the inputs significantly alters the content of a lookup table. For n_l inputs, $n_l!$ permutations exist and thus up to $n_l!$ different lookup table values for one so-called *unique function*. To compare the contents of the lookup table from the netlist and the bitfile, it must be checked if one of these possible different lookup table values for one unique function is equal to the value of the lookup table in the bitfile. This is done by creating a table with all possible values of lookup tables for all unique functions (see Figure 3.8).

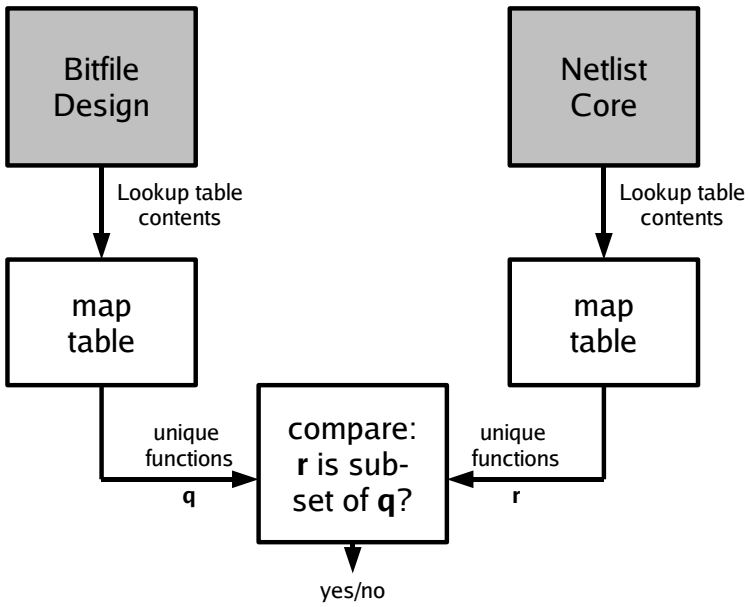


Figure 3.8: Before the lookup table contents of the bitfile and the netlist are compared, they are mapped into unique functions.

Robustness Analysis

For robustness analysis against false positive detection, it is necessary to know how many different functions can be realized by an n_l input lookup table, if it is allowed to permute the inputs. This topic is related to the problems of Boolean matching and equivalence classes [Har65, Hüt03, ZV96, DS99]. Boolean matching is a technique to check if two Boolean function are equal with respect to transformations (e.g., input permutations, input negotiation, output negotiation). Functions which are equivalent

with respect to these transformations are in the same equivalence class. Our interest is on the equivalence class P which allows to permute the inputs of a function.

The number of different values which can be encoded in an n_l-input lookup table is $2^{2^{n_l}}$. The number of equivalence classes P is smaller. Table 3.1 shows the number of equivalence classes of functions with different amounts of inputs. These values were obtained by experimentation. Functions which have a constant output do not appear in a netlist, but the mapping tool can create such functions later to generate a power or ground signal. Nevertheless, we define the set of unique functions as the set of equivalence classes without the two constant functions.

Inputs n_l	equivalence classes P	unique functions f_u
1	4	2
2	12	10
3	80	78
4	3984	3982
5	37333248 [Har65]	37333246

Table 3.1: The number of different equivalence classes P and unique functions.

In order to decide whether a certain core is instantiated inside an FPGA, we define two vectors $\mathbf{q}$ and $\mathbf{r}$ that contain all quantities of each unique function appearing in the core I_{L_1} ($\mathbf{r}$) and the whole design ($\mathbf{q}$) that may consist of multiple different cores. Also, the number of different unique functions, denoted with f_u, is of interest (see Table 3.1).

The unique function i is included r_i times in the core I_{L_1} and thus q_i times in the design I_B. Moreover, r is the number of all functions or lookup tables in the core, and q the number of lookup tables in the whole design. The vector $\mathbf{r}$ can be obtained from the set of different unique functions of the netlist of the core I_{L_1} and the vector $\mathbf{q}$ can be created from the information of the lookup table content extraction from the bitfile I_B. Let

$$\mathbf{r} = (r_1, r_2, r_3, \cdots, r_{f_u}), \tag{3.11}$$

$$\mathbf{q} = (q_1, q_2, q_3, \cdots, q_{f_u}), \tag{3.12}$$

and

$$r = r_1 + r_2 + r_3 + \cdots + r_{f_u}, \tag{3.13}$$

$$q = q_1 + q_2 + q_3 + \cdots + q_{f_u}. \tag{3.14}$$

Now, we can define a necessary criterion for a core to exist in a design: If the number of each different unique function q_i in $\mathbf{q}$ is greater or equal to r_i in $\mathbf{r}$, the core is possibly included in the design:

$$\forall i \in \{1, \cdots, f_u\}: \quad q_i \geq r_i \tag{3.15}$$

Also, if a high percentage of unique functions q_i in $\mathbf{q}$ is lower than r_i in $\mathbf{r}$, we can come to the conclusion that the core is most likely not included.

An important value is the probability p_c for the false-positive detection that a core is found in a design by chance but which has this core not included. This value should be very low to obtain a high robustness of this method. The probability depends on q, r, f_u, the core $\mathbf{r}$ and the probability distribution $\mathbf{p}_q$ of the unique functions of the whole design.

$$\mathbf{p}_q = (p_1, p_2, p_3, \cdots, p_{f_u}), \tag{3.16}$$

$$p_1 + p_2 + p_3 + \cdots + p_{f_u} = 1, \tag{3.17}$$

where p_1 is the appearance probability of the function 1 in the design $\mathbf{q}$.

First, we assume that $q = r$. This means that we calculate the probability that the number of each unique function of $\mathbf{q}$ and $\mathbf{r}$ is equal. All lookup tables of $\mathbf{r}$ must be in $\mathbf{q}$ with the probability distribution $\mathbf{p}_q$. This can be calculated with the multinomial distribution:

$$p_c = \binom{r}{r_1, r_2, r_3, \cdots, r_{f_u}} \cdot p_1^{r_1} \cdot p_2^{r_2} \cdot p_3^{r_3} \cdot \ldots \cdot p_{f_u}^{r_{f_u}}. \tag{3.18}$$

If $q > r$, the core I_{L_1} ($\mathbf{r}$) is combined with other cores or logic in the design I_B ($\mathbf{q}$). For each possible combination from $\mathbf{r}$ with other cores, the appearance probability must be calculated and summed up. The first question is how many combinations of other cores (s) exist. This can be calculated with the formula of combination with repetitions:

$$s = \frac{(f_u + q - r - 1)!}{(q-r)! \cdot (f_u - 1)!} \tag{3.19}$$

Now, all possible combinations of functions in these cores are calculated and stored in a matrix A.

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1f_u} \\ a_{21} & a_{22} & \ldots & a_{2f_u} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \ldots & a_{sf_u} \end{pmatrix}, \tag{3.20}$$

$$\forall i \in \{1, \cdots, s\}: \quad a_{i1} + a_{i2} + \cdots + a_{if_u} = q - r \tag{3.21}$$

For example, for $f_u = 2$ and $q - r = 3$:

$$A = \begin{pmatrix} 0 & 3 \\ 1 & 2 \\ 2 & 1 \\ 3 & 0 \end{pmatrix} \tag{3.22}$$

The probability that a core is detected in $\mathbf{q}$ with the probability distribution $\mathbf{p}_q$ is:

$$p_c = \sum_{i=1}^{s} \binom{q}{r_1 + a_{i1}, r_2 + a_{i2}, \cdots, r_{f_u} + a_{if_u}} \cdot p_1^{r_1 + a_{i1}} \cdot p_2^{r_2 + a_{i2}} \cdot \ldots \cdot p_{f_u}^{r_f + a_{if_u}}. \quad (3.23)$$

If we assume that all unique functions in $\mathbf{q}$ are uniformly distributed, we can simplify this formula to:

$$p_c = \sum_{i=1}^{s} \binom{q}{r_1 + a_{i1}, r_2 + a_{i2}, \cdots, r_{f_u} + a_{if_u}} \cdot \frac{1}{f_u^q} \quad (3.24)$$

Unfortunately, this probability can be calculated only for small values of q, r, and f_u (see Figure 3.9 and 3.10), because of the exponential computation complexity. For values of q, r, and f_u which can appear in realistic applications, we can only refer to the experimental results in Section 3.5.1.

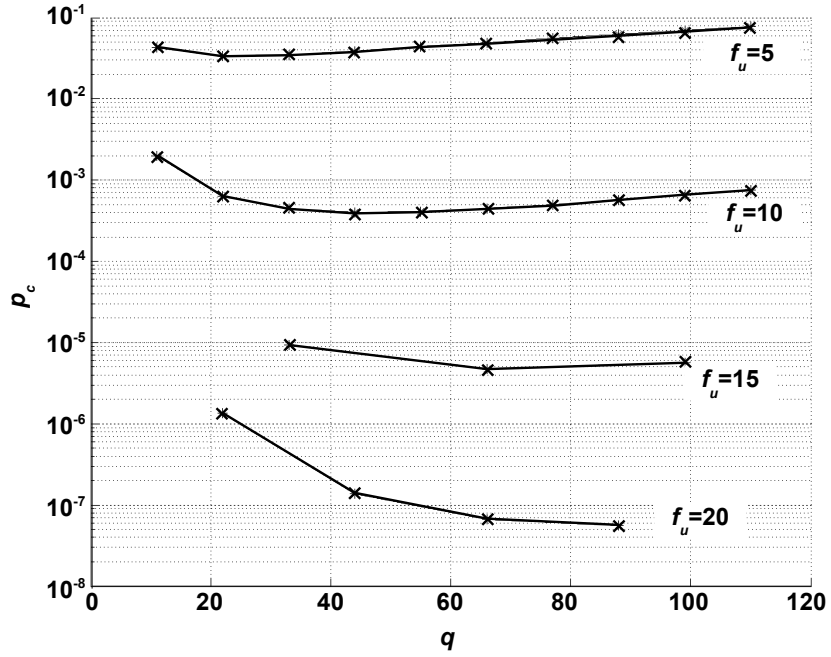


Figure 3.9: p_c for different f_u and q, where $r = 0.91 \cdot q$ and $\mathbf{r}$ is uniformly distributed.

In order to increase the robustness of our method and lower the risk of a false detection of a core, the positions of the lookup table in an FPGA can be used. The position of a lookup table can be extracted from the bitfile as well as the content (see Section 3.3.1). We assume that the lookup tables of a core are placed together, because elements with direct connection are likely to be placed in close proximity to

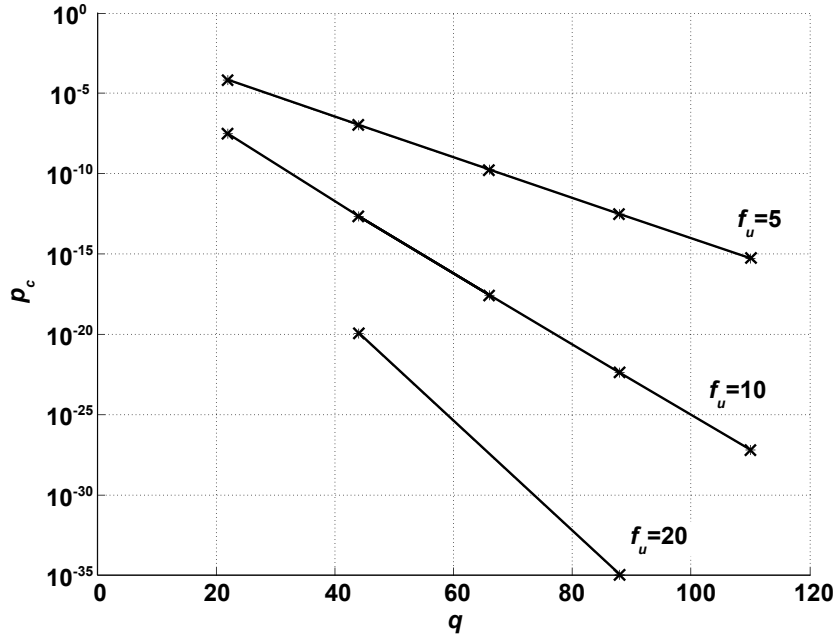


Figure 3.10: p_c for different f_u and q. $r = 0.91 \cdot q$ and $\mathbf{r}$ is distributed as follows: $p_{r1} = \frac{1}{2} + \frac{1}{2f_u}$ and $\forall i \in \{2, \cdots, f_u\} : p_{ri} = \frac{1}{2f_u}$.

obtain an optimal routing. The assumption is that lookup tables of a core have more connections with elements inside the core than with external elements.

If the mean distance between the found lookup tables in the bitfile is small the probability that these lookup tables are part of the searched core is higher than if the mean distance is high. In order to calculate this mean distance, we must know which lookup tables in the bitfile belong to the core.

First, we search for those functions which only appear in the core I_{L_1} and not in the other cores $(I_{L_2}, I_{L_2}, \dots)$, i.e.

$$\{i \in \{1, \dots, f_u\} \mid q_i = r_i\} \tag{3.25}$$

Lookup tables which implement these functions are surely inside the core. From these lookup tables, we can calculate an estimate of the geometrical position of the core center. If the number of these functions is zero, we take all lookup tables which implement functions from the core to calculate the core center. In this case, the core center is inaccurate because also lookup tables which do not belong to the core are considered. For functions which appear inside and outside the core, i.e.,

$$\{i \in \{1, \dots, f_u\} \mid q_i > r_i\} \tag{3.26}$$

we take the r_i lookup tables which are nearest to the calculated core center. Now, the distance to the core center of all lookup tables of the core can be calculated.

In the formula for p_c, the difference between q and r is important. If $q-r$ is small, then also the probability of a false detection p_c is low. We can decrease $q-r$ if we define a bounding box around the core and only consider lookup tables inside the box. The dimensions of the box can be calculated from the positions of the lookup tables from the core inside the bitfile. However, the value of the probability p_c must be multiplied with the number of all possible positions of the bounding box in the design.

Summary

We have presented a new method to identify IP cores in FPGA bitfiles. Possible transformations of the mapping tools and the effect of the robustness of the method were discussed. The experimental results in Section 3.5.1 show that it is possible to identify a core in the design with a high probability. The identification process is based on two parameters, namely the number of found lookup tables of the core in the design and the mean distance to the core center. However, it must be taken into account that lookup tables of the core are removed by optimization tools, if parts of the core are not used because outputs are unused or constant values are applied to inputs.

3.3.3 Identification of HDL Cores by Analysis of LUT Contents

In the last section we have shown that is possible to identify an IP core, distributed as a netlist, in an FPGA design by analyzing the LUT contents of the configuration bitfile. However, many IP cores are published at the RTL abstraction level as HDL core.

To identify HDL cores, the lookup table contents can be used as well. However, the lookup table content is generated by the synthesis step, which is executed after the publication of the HDL cores. Therefore, a pirate who can obtain an unlicensed HDL core, controls the complete design flow from the RTL to the device level. It is up to the pirate to decide which synthesis tool is used to synthesize the core and therefore, create the lookup table contents. Different synthesis tools might create different lookup table contents. To prove or disprove this assumption, we analyzed common synthesis tools with respect to the generation of lookup table contents.

The first step is to analyze different netlist cores to find out whether they were generated from the same HDL core. The goal is to find different netlist cores which can be assigned to a corresponding HDL source even if they were synthesized with different tools and different synthesis parameters.

Criterion Value Method

To compare two netlist cores, we define the criterion value C. If this value is greater than 1, then it is likely that both netlist cores were generated from the same HDL source. C is based on the lookup table content of both netlist cores. Therefore, the first step is to extract the lookup table contents from the netlist cores and map these to unique functions. For one core, this step is the same procedure as in netlist core identification, described in Section 3.3.2. The procedure is done for both cores and the results are two vectors which store the number of each unique function for the corresponding core. $\mathbf{r}_a$ is the vector of the first core I_{LA}, and $\mathbf{r}_b$ is the vector of the second core I_{LB}:

$$\mathbf{r}_a = (r_{a1}, r_{a2}, r_{a3}, \cdots, r_{af_u}), \quad r_a = \sum_{i=1}^{f_u} r_{ai}, \tag{3.27}$$

$$\mathbf{r}_b = (r_{b1}, r_{b2}, r_{b3}, \cdots, r_{bf_u}), \quad r_b = \sum_{i=1}^{f_u} r_{bi}. \tag{3.28}$$

The number of lookup tables that can be found in both cores and, which implement the same unique function can be calculated if we take the minimum for all elements of both vectors:

$$r_{min,i} = min(r_{ai}, r_{bi}), \quad \forall i \in 1 \ldots f_u. \tag{3.29}$$

The value $P_{a\ in\ b}$ denotes the percentage of unique functions of I_{LA} that can be found in I_{LB}. The value $P_{b\ in\ a}$ is calculated vice versa:

$$P_{a\ in\ b} = \sum_{i=1}^{f_u} \frac{r_{min,i}}{r_b}, \tag{3.30}$$

$$P_{b\ in\ a} = \sum_{i=1}^{f_u} \frac{r_{min,i}}{r_a}. \tag{3.31}$$

Note that both values are not necessarily the same, because of the possibly different number of used lookup tables or functions r_a and r_b. For example, if we compare a core with a large number of lookup tables with a core which has only few lookup tables, it is likely that the content of the few lookup tables of the second core is also found in the large core. Vice versa, however, the percentage of the found lookup tables should be very low.

If the number of used lookup tables differs extremely between the two cores, then the probability that both cores were generated from the same source is low. We introduce a parameter I_P which indicates that the difference of the number of used lookup tables of both cores is more than the average of used lookup tables:

$$r_{avg} = \frac{r_a + r_b}{2}, \tag{3.32}$$

$$r_{diff} = |r_a - r_b|, \tag{3.33}$$

$$I_P = \begin{cases} 0, & \text{if } r_{diff} > r_{avg} \\ 1, & \text{if } r_{diff} \leq r_{avg} \end{cases} \tag{3.34}$$

Now, we can calculate the criterion value C:

$$C = 25 \cdot I_P \left(\frac{r_{avg}}{r_{diff}} \right)^2 \cdot \left(P_{a\,in\,b} - \frac{P_{a\,in\,b} + P_{b\,in\,a}}{2} \right)^2 + \left(P_{b\,in\,a} - \frac{P_{a\,in\,b} + P_{b\,in\,a}}{2} \right)^2 \tag{3.35}$$

This formula is derived from the experimental results in Section 3.5.2. If $C > 1$ then the probability that both cores were generated from the same source is high, if $C < 1$ it is rather unlikely that both cores are related.

Identification of Synthesis Tools

After the identification of netlist cores that have been generated from the same HDL source, it is also of interest to identify the synthesis tool that was used to generate the netlist. If it is possible to identify the used synthesis tool from the lookup table contents, the HDL core can be synthesized with the same tool. A comparison of the generated netlist core with the bitfile can possibly provide better identification result. The fact, that only a few synthesis tools exist for a certain technology simplifies the identification of the synthesis tool as well as the used core.

The first step is to analyze netlists that were generated by different synthesis tools. We identify characteristic usage of special primitive cells, which can be found only in netlists which were synthesized by a certain tool. For example, in our analysis for Virtex-II synthesis tools we found different implementations of a simple inverter function. On *precision synthesis* from *Mentor Graphics* an inverter is always implemented into a LUT1 or LUT1_L primitive cell, whereas the *Xilinx Synthesis Technology (XST)* synthesis tool implements an inverter always as an INV primitive cell. *Synplify Pro* from *Synopsis* uses both implementations for an inverter. Furthermore, LUT_D primitive cells which use the local and the normal lookup table output (see [Xilh]) exist only in netlists which were generated by the XST synthesis tool.

However, during the implementation of these cores into a bitfile, an INV primitive cell might also be implemented into a one-input lookup table, which removes this special characteristic. Future research might find better characteristics for synthesis tools which might still be present in the bitfile. Ideally, such characteristics should be found in lookup tables, which can be easily extracted.

Conclusions and Future Work

We have shown the first steps towards an identification of HDL cores in bitfiles. We concentrated on the synthesis step between the RTL and logic abstraction level and have shown that it is possible to identify netlist cores that were generated from the same HDL source. Furthermore, we have shown that it might be possible to identify the synthesis tool a netlist was generated by. However, to build the complete chain for identification of HDL cores from bitfiles some links are missing. Being able to identify the synthesis tool helps for the identification, but even if the same tool is used, the resulting netlists may be different due to varying synthesis parameters. Identifying HDL cores in netlists has an inherent uncertainness comparable to the identification of netlist cores in bitfiles. By combining both techniques the uncertainness can be too high to give a trustful result. Nevertheless, this is an interesting topic for future research.

3.3.4 Watermarks in LUTs for Bitfile Cores

In this section, we introduce our first watermarking technique for IP cores. The easiest way to watermark an FPGA design is to place the watermarks into the bitfiles. Bitfiles are very inflexible because they were specifically generated for a certain FPGA device type, however, it makes sense to sell bitfile IP cores for common development platforms which carry the same FPGA type. Usually, a bitfile core is a whole design which is completely placed and routed and therefore ready to use. There also exist partial bitfiles which carry only one core. These partial bitfile cores can be combined into one FPGA which increases the flexibility of these cores and therefore may increase the trade possibilities.

In this approach, we hide our signature inside unused lookup tables. It is very unlikely that a design or bitfile core uses all available lookup tables in an FPGA. Before a design reaches this limit, the routing resources are exhausted and the timing degenerates rapidly. Therefore, many unused lookup tables exist in usual designs. On the other hand, lookup table content extraction is not difficult (see Section 3.3.1). Using lookup tables for hiding a watermark which are far away from the used ones, makes it easier for an attacker to identify and remove them. Even if an attacker is able to extract all lookup tables from a bitfile core, the lookup tables which carry the watermark should not be suspicious.

In Xilinx devices, lookup tables are grouped together with *flip-flops* into slices. A slice usually consists more than one lookup table, e.g., the Virtex-II and Virtex-II Pro devices which are discussed in Section 3.3.1, have two lookup tables in one slice. It is not unusual that only one lookup table of a slice is used and the other remains unused. Hiding a watermark in the unused lookup table of a used slice is less obvious than using lookup tables in unused slices. Even if the attacker is able to extract the lookup table content and coordinates, the watermarks are hard to detect.

The extraction and verification of the watermark is rather easy. First of all, the content and the coordinates of all used lookup table of the core are extracted. For the verification there exist two approaches: a *blind approach* and a *non-blind approach*. In the blind approach, the watermarks are searched in all extracted lookup table contents, whereas in the non-blind approach the location of the watermarks are known. Having the right coordinates, the watermarked lookup table content can be directly compared to the watermarks of the core developer. The locations of the watermarks delivered from the core developer, however, should be kept secret, because otherwise it is very easy for an attacker to remove the marks.

Concept

In the following, the watermark approach is described in detail. For watermarking a bitfile core, the watermarks which should be embedded into the unused lookup tables must be generated. This is done by the watermark generator function: $\mathcal{G}_B(K) = W_B$. This watermark generator can be, for example, an algorithm similar to Figure 3.4 in Section 3.2.2. The generator needs a unique key K which identifies the author as well as the core and the authors private key as input. The output is a set of watermarks $W_B = (w_{B_1}, w_{B_2}, \cdots, w_{B_m})$. Each element w_{B_i} must fit into a single lookup table. For Xilinx Virtex-II and II Pro FPGAs, which use 4-input lookup tables, the size is 16 bit.

Additionally, the number of usable lookup tables which can carry a watermark must be determined. This can be done by extracting all lookup table contents and coordinates: $\mathcal{L}_B(I_B) = \{x_{B_1}, x_{B_2}, \ldots, x_{B_q}\}$. The next step is to find suitable location candidates which can carry a watermark. For Xilinx Virtex-II and II Pro FPGAs, possible candidates are unused lookup table in a used slice. Such candidates can be easily determined, because they carry the initialization value `0xFFFF`, whereas unused lookup tables in unused slices have `0x0000` as initialization value (see Section 3.3.1). The higher the number of location candidates and therefore watermarked lookup tables is, the more reliable is the proof of authorship. For example, if only one lookup table candidate was found, only $2^4 = 16$ different watermark values overall exist, which makes the proof of authorship contradictable.

The content of the chosen locations of the bitfile core I_B can be replaced by the watermarks W_B with the embedder $\widetilde{I_B} = \mathcal{E}_B(I_B, W_B)$ (see Figure 3.11). The result is the watermarked bitfile core $\widetilde{I_B}$. The distance $Dist_B(I_B, \widetilde{I_B})$ between the watermarked and original core is low, because of the functional correctness and all electrical properties of the core are preserved. Furthermore, if the watermarks are near to the functional lookup tables, the watermarks cannot be easily distinguished from the functional lookup tables.

For extracting the watermarks, we need the bitfile $\widetilde{I_B}$ from the accused company, and the locations of the watermarks (see Figure 3.11). The first step is to extract the content and coordinates from all lookup table in $\widetilde{I_B}$: $\mathcal{L}_B(\widetilde{I_B}) = \{\widetilde{x}_{B_1}, \widetilde{x}_{B_2}, \ldots, \widetilde{x}_{B_q}\}$.

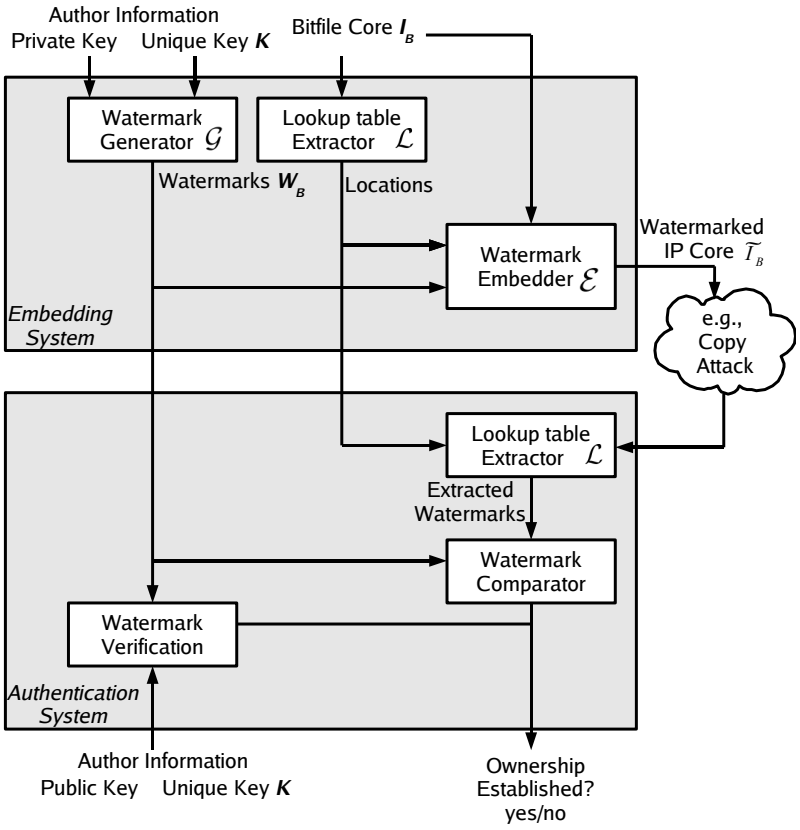


Figure 3.11: The watermarking bitfile IP core approach consists of an embedding system and an authentication system. The embedder needs the author information and the bitfile core and the result is the watermarked core $\widetilde{I_B}$ which can be published. The authorship of the core can be established by extracting and comparing the watermark and verifying the authentication of the watermark with the author information.

Using the locations from the core developer, the watermarks $\widetilde{W_B}$ can be identified. By comparing these watermarks to the watermarks W_B of the core developer, the detection process $\mathcal{D}_B(\widetilde{I_B}, W_B) = true/false$ can be finished.

Finally the authentication of the watermarks W_B must be proofed. If the algorithm depicted in Figure 3.4 is used for watermark generation, the watermarks W_B can be verified as follows: The author must present the seed for the verification process. From the seed, the pseudo random sequence which is also the watermarking information is generated by the RC4 algorithm. If this watermark information is equal to W_B, this seed was used for watermarking the core. Furthermore, the seed can be decrypted using the public key of the author. The key K which identifies the author and the core uniquely should be the result. Hereby, the verification process and the proof of authorship is finished.

Robustness Analysis

Attacks against the watermarking scheme are, according to Section 3.2.1, *ambiguity, removal*, and *key copy attacks*. The prevention of the *copy attack*, where an attacker watermarks an IP core which he illegally obtained with his own signature, is almost impossible. A possible solution of this dilemma is to watermark all published works or register the core on a trusted third party institute.

In case of *removal attacks*, the attacker tries to remove the watermarks. If he knows the location of the watermarks this task is easy. Therefore, it is utmost important that the locations of the watermarked are kept secret. If the attacker does not know the locations, he can try to analyze the bitfile. If he is only able to extract the lookup table content and the locations of the lookup tables, it is almost impossible to detect the watermark, because the locations are near the functional lookup tables and the content is not distinguishable from the other lookup tables. However, if the attacker is able to reverse engineer the bitfile core to the logic level ($\widetilde{I_L} = \mathcal{T}_{L \leftarrow B}(\widetilde{I_B})$), the watermarks are easy to detect and can be removed. This task is, however, very expensive if no reverse engineering tool is available. For Virtex-II devices the Xilinx "reverse engineering" tool *JBits* [Xilc] is available, which is in fact able to remove the watermarks.

The attacker may analyze the bitfile core and search for lookup table content which he can present as his own watermark in case of *ambiguity attacks*. He can use the inserted watermarks and assert that these watermarks belong to him. To be successful with such an attack, he must also present the procedure to generate the watermarks. Hereby, the attacker must generate a signatures or key which identifies him as the author and fits to the watermarks inside the core. This is very hard to achieve due to the usage of one way cryptographic functions. Furthermore, the attacker can present some functional lookup tables as his watermarks. This should also be nearly impossible due to the characteristics of one way cryptographic functions. Another possibility to check this attack, is to remove the watermarks from the bitfile core. The correct watermarks are inserted after the implementation of the core and therefore the core should keep the functional correctness. Whereas the removal of the wrong watermarks which are functional lookup table contents, destroys the core.

Using asynchronous public/private key cryptographic functions for the watermark generation and verification and further storing information about the core into the unique key successfully prevents *key copy attacks*.

3.3.5 Watermarks in Functional LUTs for Netlist Cores

After watermarking bitfile cores, we now watermark netlist cores. Netlist IP cores consist of *primitive cells* (e.g., LUT4, DFF, XORCY) of a certain FPGA family which covers many different FPGA devices. For example, the whole Xilinx Virtex-4, or Altera Stratix-II family with all different FPGA sizes. This means, that one netlist core can be deployed for the whole family without changing the file. Once again,

we are using the Virtex-II and II Pro family to demonstrate this approach. However, using other FPGA families should also be possible by adapting the methods to their primitive cells. Another big advantage from netlist cores over bitfile cores is that the bitfile creator (e.g., product developer) can combine different cores.

As mentioned before in Section 3.3.2, FPGAs usually consist of the same type of lookup tables with regard to the number of inputs. For example, the Xilinx Virtex-II uses lookup tables with four inputs whereas the Virtex-5 has lookup tables with six inputs. However, in common netlist cores many logical lookup tables exist, which have less inputs than the FPGA type. These lookup tables are mapped to the physical lookup tables of the FPGA. If the logical lookup table of the netlist cores has fewer inputs than the physical one, the memory space which cannot be addressed remains unused. We use this memory space to embed a watermark into functional lookup tables.

One problem of watermarking netlist cores is that the core further traverses the design flow which includes different optimization steps. Additive watermarking methods which use redundant structures or logic as watermark have the problem that the global optimization steps may detect and remove this redundancy. Todays design tools are very sophisticated to find redundant logic in a design. Even if a special redundant logic which can be used as watermark is not removed by today's tools, it is not granted that future versions or other tools may not detect and remove this logic. The challenge is to find an element or component which can be used as watermark and is not altered by design tools. For Xilinx FPGAs such elements are shift registers and memories which are implemented in lookup tables.

In some FPGA architectures (e.g., all Xilinx Virtex architectures), the lookup tables (LUTs) can also be used as a *shift register* or *distrubuted memory* [Xilf]. For example, a 4-input lookup table can be further used as a 16-bit shift register (see Figure 3.12). The content of such a shift register can be further addressed by the lookup table input ports. So, the shift register can also be used as a functional lookup table. If the lookup table is used as a LUT primitive cell, the content is interpreted as logic by the design tools and is in focus of optimization. However, if the same content is used as a shift register or memory primitive cell, the design tools do not touch the content. Using the unused memory space of functional lookup tables for watermarking without converting the lookup table either to a shift register or distributed memory turns out to be not applicable, because design flow tools identify the watermark as redundant and remove the content due to optimization. Converting the watermarked functional lookup table into shift registers or memory cells, prevents the watermark from deletion due to optimization.

Embedding of the Watermark

In this approach we are use Virtex-II Pro FPGAs and convert LUT1, LUT2, or LUT3 primitive cell which can be found in netlists of IP cores into the shift register primitive

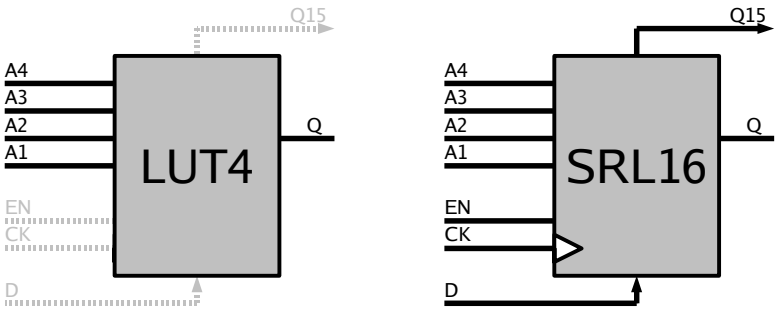


Figure 3.12: In the Xilinx Virtex architecture, a lookup table (LUT4) can also be configured as a 16-bit shift register (SRL16).

cell SRLC16E. Note that LUT1 has one input, LUT2 two and so on. LUT4 has four input and uses the whole lookup table memory for its function which make this type uninteresting for our approach. The content of the physical 4-input lookup table in an FPGA stores 16 bits. A LUT3 primitive cell uses only 8 bits, a LUT2 4 bits, and LUT1 only 2 bits out of the 16 bits. The Xilinx mapping tool *map* duplicates the used memory area to the unused area if not all inputs are needed (see Section 3.3.2). Therefore, to use the unused memory space for embedding a watermark, we must restrict the memory reachability of the function by clamping the unused inputs to constant values. In Figure 3.13, we demonstrate this idea for an AND-function, implemented by a LUT2. By clamping input A3 and A4 to zero, we can free 12 bits which can be used for carrying a watermark.

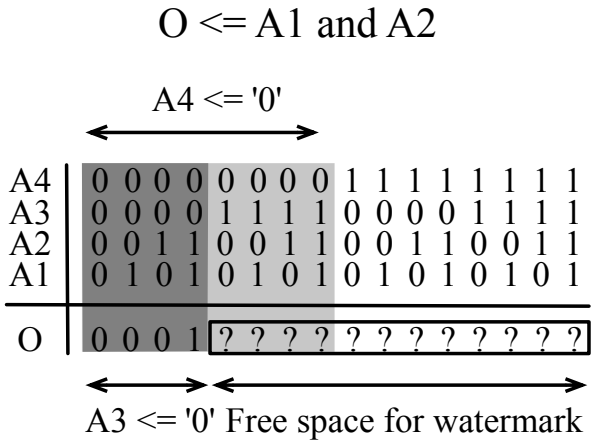


Figure 3.13: Example of implementing a two input AND gate using a four input lookup table. Addressable storage is restricted by connecting the unused inputs to zero [SZT08].

Another problem of watermarking netlist cores is that the published core is combined with other cores and undergoes further design flow steps, like the placement of the lookup table in the FPGA. Therefore, at the extraction of the watermark, we do not know the locations of the watermarks. To reduce the effort for identifying the watermarks after the extraction, we can cascade the watermarks over the *shift in* (D) and *shift out* (Q15) ports of the shift register cell. We assume, that design tools place these chains of watermarks close together which extremely simplifies the extraction of the watermarks. Furthermore, for rebuilding the watermark from the individual extracted watermarked lookup tables, the sequence is important. To bring the different watermarks, which have further different sizes according to the used original functional lookup table cell, into the right order, we concatenate the watermark bits with a counter. Due to limited space, only few counter bits can be used, which results in a repetition of the counter values. Nevertheless, this method further simplifies the detection and extraction of the watermark during the verification process.

The first step of embedding a watermark is to extract all lookup tables from a given netlist core I_L: $\mathcal{L}_L(I_L) = \{lut_{L_1}, lut_{L_2}, \ldots, lut_{L_r}\}$, where L denotes the logic abstraction level used for netlist cores (see Figure 3.14). Each element lut_{L_i} denotes a lookup table primitive cell in the netlist (e.g. for Virtex-II devices, LUT1, LUT2, LUT3, or LUT4). A watermark generator $\mathcal{G}_L(\cdot,\cdot)$ must know the different lookup table cells with the functional content as well as the unique key K to generate the watermarks: $\mathcal{G}_L(K, \mathcal{L}_L(I_L)) = W_L$.

From the unique key K a secure pseudo random sequence is generated, for example with an algorithm depicted in Figure 3.4 in Section 3.2.2. Some or all of the extracted lookup table primitive cells are chosen to carry a watermark. Usually a core which is worth to be watermarked consists of many markable lookup tables. Transforming all of these lookup tables into shift registers restricts the optimization degree of the tools and results in non-optimal timing behavior. Therefore, only a small subset of all suitable lookup table are chosen. Note that the shift registers must never be shifted, because this alters the functional part of it. Nevertheless, we connect the clock input with the clock, but the shift enable input to ground. Now, the transformed shift registers are ordered and the first 4 bits of the free space are used for the counter value. The other bits are initialized according to the position with values from the pseudo random stream, generated from the key K. Note that the number of bits which can be used for the random stream depends on the original functional lookup table type.

The generated watermark W_L consists of the transformed shift registers: $W_L = \{srl_{L_1}, srl_{L_2}, \ldots, srl_{L_k}\}$ with $k \leq r$. The watermark embedder $\mathcal{E}_L$ inserts the watermarks into the netlist core I_L by replacing the corresponding original functional lookup tables with the shift registers: $\mathcal{E}_L(I_L, W_L) = \widetilde{I_L}$. The watermarked work $\widetilde{I_L}$ can now be published and sold.

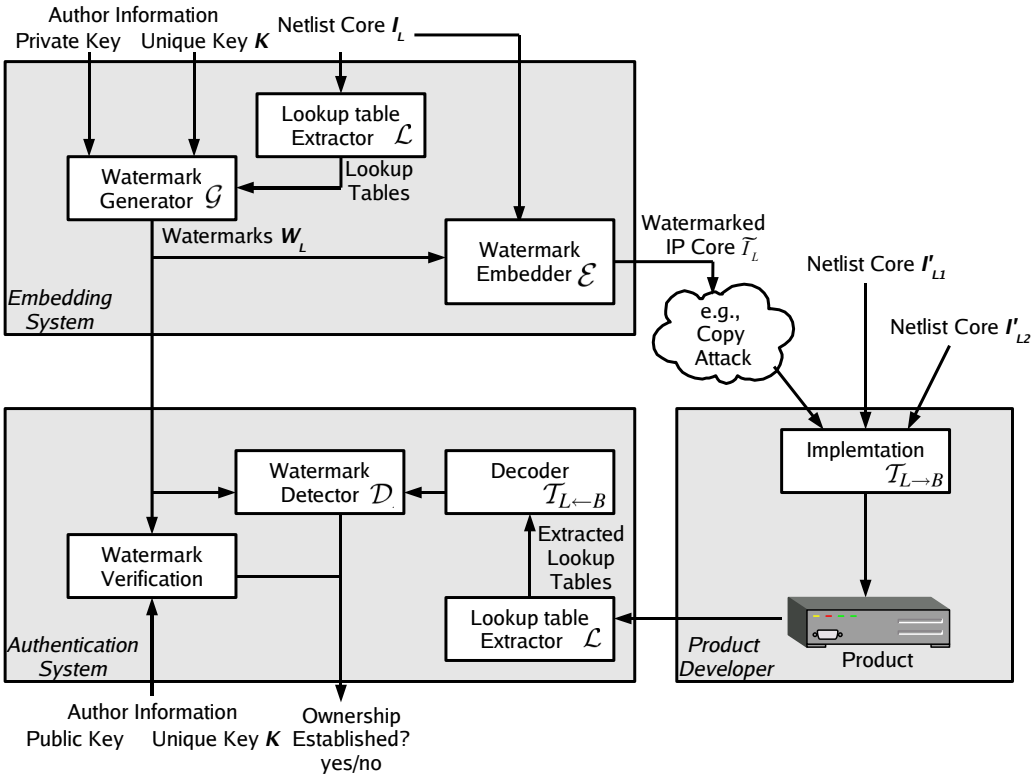


Figure 3.14: The watermarking netlist core system. In the embedding system the lookup tables are extracted from the netlist core and the watermark generator select suitable lookup table, transform it to shift register and add the watermark. The embedder insert the watermark. A product developer may obtain this watermarked netlist core an combine it with other cores into a product. The lookup tables from the product can be extracted and transformed so, that the detector can decide if the watermark is present or not.

Extraction of the Watermark

The purchased core $\widetilde{I_L}$ can now be combined by a product developer with other purchased or self developed cores and implemented into an FPGA bitfile: $\widehat{I_B} = \mathcal{T}_{L\to B}(\widetilde{I_L} \circ I'_{L_1} \circ I'_{L_2} \circ \dots)$ (see Figure 3.14). An FPGA which is programmed with this bitfile $\widehat{I_B}$ may be part of a product. If the product developer is accused of using an unlicensed core, the product can be purchased and the bitfile can be read out, e.g., by wire tapping. The lookup table content and the content of the shift registers can be extracted from the bitfile: $\mathcal{L}_B(\widehat{I_B}) = \{\widehat{x}_{B_1}, \widehat{x}_{B_2}, \dots, \widehat{x}_{B_q}\}$.

The lookup table or shift register elements x_{B_i} belong to the device abstraction level B. The representation can differ from the representation of the same content in the

logic abstraction level L. For example, in Xilinx Virtex-II FPGAs the encoding of the shift register differs from the encoding of lookup tables. For shift registers the bit order is reversed compared to the lookup table encodings. Therefore, the bitfile elements must be transferred to the logic level by the corresponding decoding. This can be done by the *reverse engineering operator*: $\mathcal{T}_{L \leftarrow B}(\mathcal{L}_B(\widehat{I}_B)) = \{\widehat{x}_{L_1}, \widehat{x}_{L_2}, \ldots, \widehat{x}_{L_q}\}$. Reverse engineering lookup table or shift register content is however very simple compared to reverse engineering the whole bitfile. Now, the lookup table or shift register content can be used for the watermark detector $\mathcal{D}_L$ which can decide if the watermark W_L is embedded in the work or not: $\mathcal{D}_L(W_L, \{\widehat{x}_{L_1}, \widehat{x}_{L_2}, \ldots, \widehat{x}_{L_q}\}) = true/false$.

The detector $\mathcal{D}_L$ searches the content of the watermarked shift register W_L in the extracted lookup table contents from the bitfile. It might occur that certain watermarks will be found in more than one locations, because more of the same watermarks exist with an identical content, or a complete functional lookup table has, by chance, the value of a watermarked one. To simplify the extraction, the watermarks are chained together by the shift in and out ports. It is likely that these watermarks are placed close together. From the bitfile lookup table extraction $\mathcal{L}_B$, we also have the locations of the possible watermarks. Using these locations we can in most cases identify the right watermark, if duplicates exist. Note that this chaining approach is not mandatory, but elevates the robustness of the approach against ambiguity attacks.

After the detection of the watermark W_L inside the bitfile $\widehat{I}_B$, the watermark must be verified similar to the watermarking approach for bitfile cores proposed in Section 3.3.4.

Robustness Analysis

To recall Section 3.2.1, the most important attacks are *removal, ambiguity, key copy*, and *copy attacks*. As stated for the watermarking method for bitfile cores in the previous section, a possible protection against copy attacks does not exist and key copy attacks can be prevented by using an asymmetric cryptographic method, like RSA.

Removal attacks most likely occur on the logic level, after obtaining the unlicensed core and before the integration with other cores. The first step of a removal attack is the detection of the watermarks. The appearance of the shift register primitive cells (here SRL16) in a netlist core is not suspicious because shift registers appear also in unwatermarked cores. However, the cumulative appearance may be suspicious, which may alert an attacker. In contrast to bitfiles, the signal nets can be easily read out from a netlist core. An attacker may analyze the net structures of shift registers in order to detect the watermarked cells. This might be successful, however, we can better hide the watermark if we alter the encoding of the watermark and, therefore, the connections to the watermark cell. The reachable functional part of the shift register can be shifted to other positions by using other functional inputs and clamping the remaining inputs to different values. If a watermark cell is detected by an attacker,

he cannot easily remove the cell, because the cell also has a functional part. By removing the cell, the functional part is removed and the core is damaged. Therefore, after the detection of the watermark, the attacker must either decode the content of the watermarked shift register to extract the functional part and insert a new lookup table, or overwrite the watermarked part of the cell with other values, so the watermark is not detectable any more. The different encodings of the functional part of the shift register content complicates the analysis and the extraction of it. Furthermore, even if some watermarks are removed, the establishment of the right ownership of the core is still possible, because we need not all watermarked cells for a successful detection of the signature.

In case of *ambiguity attacks*, an attacker analyzes the bitfile or the netlist to find shift register or lookup table contents which may be suitable to build a fake watermark. Like the watermarking method for bitfile cores, the attacker must also present the insertion procedure to achieve a meaningful result. Due to the usage of secure one way cryptographic functions for generating the watermark, the probability of a success is very low. Furthermore, the attacker can use a self-written netlist core which he watermarked with his signatures and combine it with the obtained unlicensed core. The result is, that the watermarks of the authors of both cores are found in the bitfile, which are both trustful. Inside the unique key K, not only the author information should be included but also information of the core, e.g., a hash value over the netlist core file without the watermark. Of course, the attacker can use the identification of the obtained unlicensed core for watermarking his core. However, to generate a hash value of the obtained core without watermarks, he must first remove the marks. In general, attacks against this approach are possible, but they need a high amount of effort. To increase the security against ambiguity attacks, the core may be registered at a trusted third party.

Summary

In this section we presented a watermark method, which is able to watermark netlist cores. The netlist cores can be combined with other cores to assemble a product in an uncontrolled area from the core developer's point of view. Nevertheless, the watermark can be extracted from a bitfile which is recovered from a product from an accused company. We have shown that the approach is resistant against removal and ambiguity attacks.

The disadvantage of this approach is that we might have some timing overhead due to the conversion of lookup tables to shift registers. As mentioned before, design tools can permute inputs to achieve a better timing behavior. By converting the lookup tables into shift registers we take this optimization degree away from the tools, which may results in a worse timing. Experimental results of the timing overhead can be found in Section 3.5.4.

3.4 Power Watermarking

This section introduces watermarking techniques, where a signature is verified over the *power consumption pattern* of an FPGA. These techniques may also be suitable for ASIC designs, however, we concentrate on FPGA designs and develop several enhancements which are exclusively related to the FPGA technology. The presented idea is new and differs from [KJJ99] and [AARR03] where the goal of using power analysis techniques is the detection of cryptographic keys and other security issues.

For power watermarking methods, the term *signature* refers to the part of the watermark which can be extracted and is needed for the detection and verification of the watermark. The signature is usually a bit sequence which is derived from the unique key for author and core identification.

First of all, a short introduction is given and the communication channel between the generation and the detection of the watermark is discussed. Next, the basis method is presented and afterwards, several enhanced methods which increase the robustness of decoding the watermark in case of external or internal disturbances are introduced. Finally, multiplex methods are discussed which enable the detection of more than one watermark if multiple watermarked cores are present in the design.

3.4.1 Verification over Power Consumption

There is no way to measure the relative power consumption of an FPGA directly, only through measuring the relative supply voltage or current. We have decided to measure the voltage of the core as close as possible to the voltage supply pins such that the smoothing from the plane and block capacities are minimal and no shunt is required. Most FPGAs have *ball grid array* (BGA) packages and the majority of them have vias to the back of the PCB for the supply voltage pins. So, the voltage can be measured on the rear side of the PCB using an oscilloscope. The voltage can be sampled using a standard oscilloscope, and analyzed and decoded using a program developed to run on a PC. The decoded signature can be compared with the original signature and thus, the watermark can be verified. This method has the advantage of being non-destructive and requires no further information or aids than the given product (see Figure 3.15).

The consumed power of an FPGA can be divided into two parts, namely the static and the dynamic power. The static power consumption is caused by the leakage current from CMOS transistors and does not change over time if the temperature stays constant. The dynamic power consists of the power related to short circuit currents and the power required of reloading the capacities of transistors and wires. The short circuit current occurs when the *PMOS* and the *NMOS* transistors are both in conducting state for a short time during the switching activity. As shown in [SKB02], the main part of an FPGA's dynamic power results from capacity reloading. Both parts of the dynamic power consumption depend on the switching frequency [CSB92].

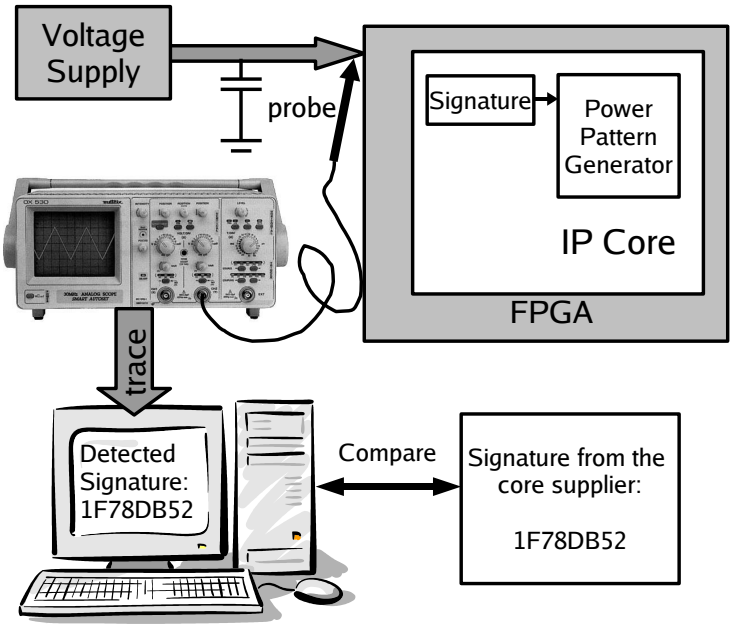


Figure 3.15: Watermark verification using power signature analysis: From a signature (watermark), a power pattern inside the core will be generated that can be probed at the voltage supply pins of the FPGA. From the trace, a detection algorithm verifies the existence of the watermark.

What happens to the core voltage, if many switching activities occur at the same time, at the rising edge of a clock signal? It is interesting to observe that the core supply voltage drops and rises (see Figure 3.16). In the frequency domain, the clock frequency with harmonics and even integer divisions are present (see Figure 3.17). The real behavior of the core voltage depends on the individual FPGA, the individual printed circuit board and the individual voltage supply circuits.

In the following, we seek for techniques to encode a watermark such that the core voltage is subject to change once the watermark is processed within a core. In the first method, the frequency of the voltage drops shall be influenced, in the second, the amplitude of the voltage drops shall be manipulated.

In the first case, a watermark can be identified if we produce another frequency line in the spectrum of the core voltage which is not an integral multiple or a rational fraction of the clock frequency. For achieving this, we need a circuit that consumes a considerable amount of power and generates a signature-specific power pattern, and a clock which can be identified in the spectrum. The power consumer can be, for example, an additional shift register. If we would derive the clock source from

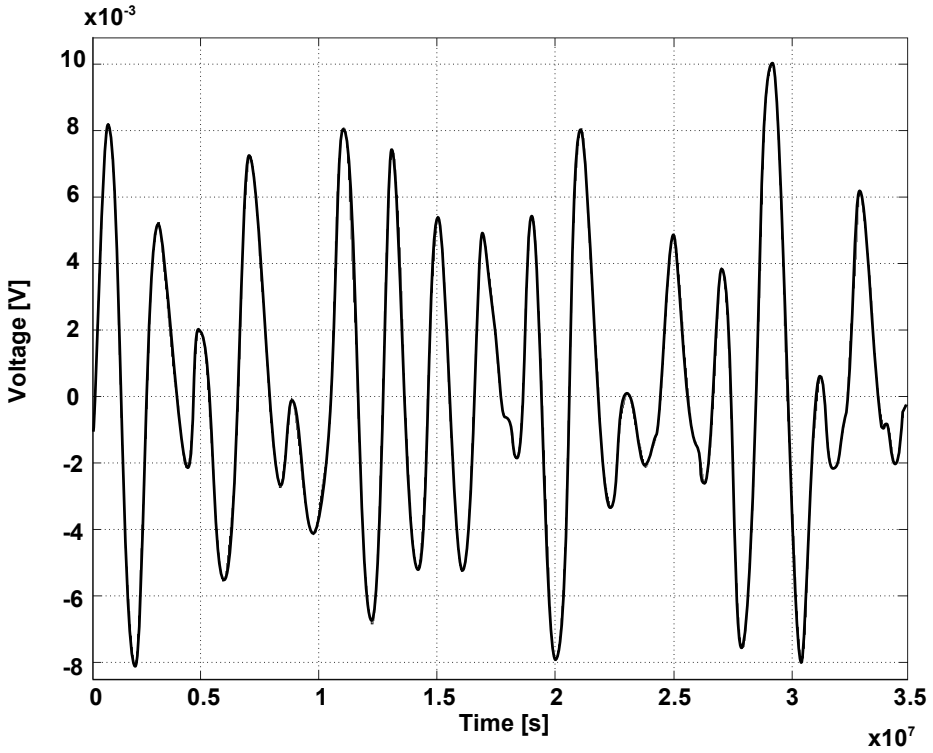


Figure 3.16: A measured voltage signal at the voltage supply pin of an FPGA. The core supply voltage drops and rises. Note that the DC component is filtered out.

the operational clock, we would not be able to distinguish the frequency line in the spectrum from operational logic. Another opportunity is to generate a clock using combinatorial logic. This clock could be identified as a watermark, but the jitter of a combinatorial clock source might be very high, and no clean frequency line could be seen in the spectrum. This means that we need a higher additional power consumer to make the watermark readable. Another drawback is that we have only limited possibilities to encode a signature reliably in these frequency lines.

In the following approaches, we alter the amplitude of the interferences in the core voltage. The basic idea is to add a power pattern generator (e.g., a set of shift registers), and clock it either with the operational clock or an integer division thereof. Further, we control these power pattern generators according to the characteristics of the data sequence which should be sent, respectively detected. A logical '1' lets the power consumer operate one cycle (e.g., perform a shift), a '0' causes no operation. We detect higher amplitudes in the voltage profile over time corresponding to the ones and smaller amplitudes according to the zeros. Note that the amplitude for the

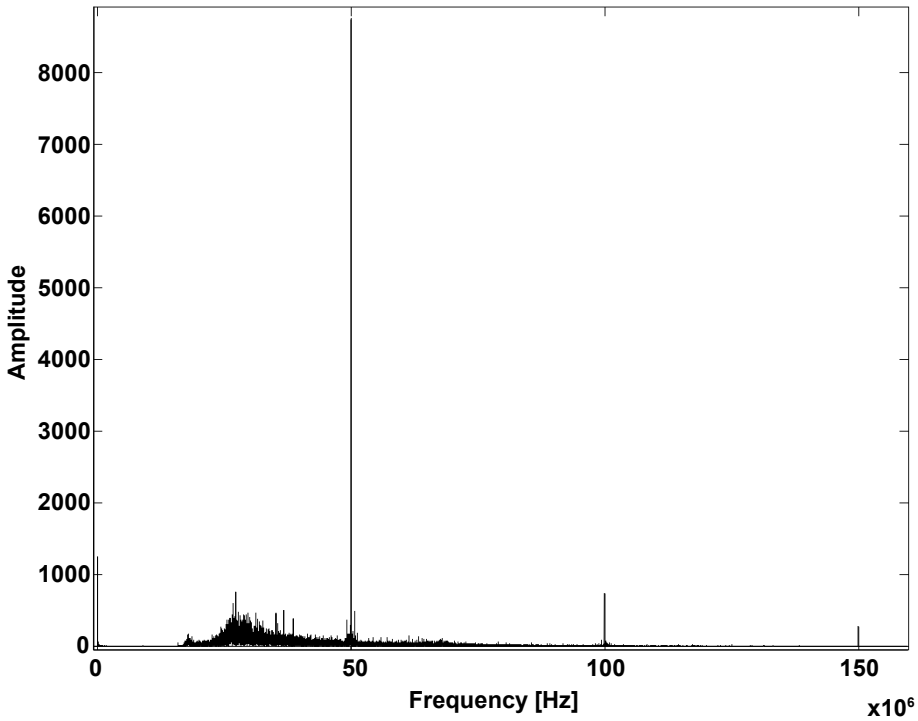


Figure 3.17: The spectrum of the measured signal in Figure 3.16. The clock frequency of 50 MHz and harmonics can be seen. Also, a peak at the half of the clock frequency is visible which is caused by switching activities from the logic.

no-operation state is not zero, because the operational logic and the clock tree is still active.

The advantage of power watermarking methods is that the signature can easily be read out from a given device. Only the core voltage of the FPGA must be measured and recorded. No bitfile is required which needs to be reverse-engineered. Furthermore, these methods work also for encrypted bitfiles whereas methods where the signature is extracted from the bitfile fail. Moreover, we are able to sign netlist cores, because our watermarking algorithm does not need any placement information. So, also cores at this level can be protectedly watermarked.

3.4.2 Communication Channel

The transmission of a signal, generated from a source of data, over the supply voltage to a testing point outside of the FPGA which can be accessed by an oscilloscope presents an *unidirectional communication system*. The source is the power pattern

generator inside the core, and the sink is a device that decodes the signal after it has been measured, digitized and recorded by an oscilloscope. This communication channel transforms the signal and adds noise. If we know how to characterize the channel, we are able to build better en- and decoding systems. Therefore, the behavior of the communication channel must be approximated in a channel model. This topic is related to communication engineering, therefore a short introduction for building a channel model is given.

The basic elements of a communication system are the *source*, the *source encoding*, the *channel encoding* and *modulation*, the *communication channel*, the *channel demodulation* and *decoding*, the *source decoding* and the *sink* (see Figure 3.18). In current digital communication systems as well as in our power watermarking technique, the source encoding is usually a binary encoding of the data to be transmitted. From the source encoded data, the transmission sequence is generated by channel encoding. The digital modulation then translates the encoded data sequence into single, successively transmitted *symbols*.

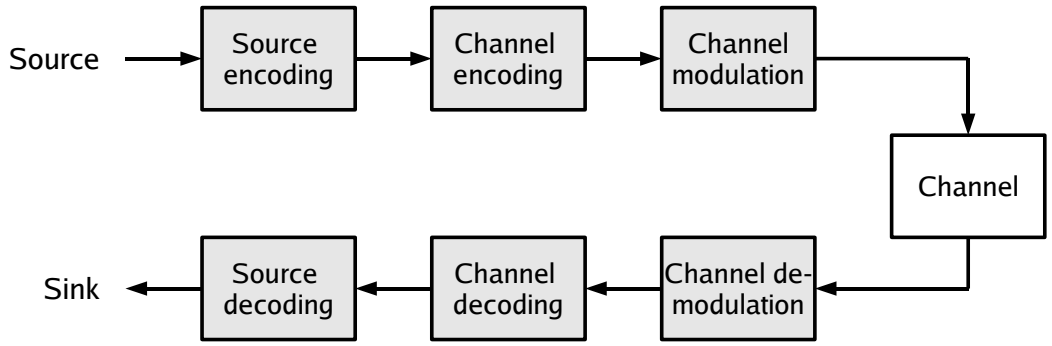


Figure 3.18: The basic elements of a digital communication system. The source which is encoded by source encoding is first transferred to the channel encoder and then modulated on the channel. The receiving part demodulates the signal and decodes the channel encoded data.

A symbol σ is the smallest unit which carry information, however a symbol can carry more than one bit of data. For example, by using a *Quadrature Phase Shift Keying* (QPSK) [PS95] modulation, multiple bits can be represented by a single symbol. The sequence of different symbols generated by modulation is called *signal*. The signal is transmitted and altered by the communication channel. After receiving, the signal is demodulated and decoded to the source coding.

On a real communication channel, the signal is disturbed by interferences, noises, line losses, delays, etc., which complicates its decoding. With a certain probability $P_E > 0$, the symbol cannot be correctly decoded. For example, when using a binary modulation where the bit '1' corresponds to the symbol σ_1 and a '0' corresponds

to the symbol σ_0, the decoder must first calculate the probability that the currently received symbol was sent as σ_0 or σ_1. After that, the decoder can decide in favor of one symbol. To lower P_E, the channel code can add redundancy which increases the probability of correct decoding. The decision which channel code and modulation can be used depends on the communication channel as well as the sender and receiver characteristics. Thus, it is necessary to develop a channel model which adapts the actual communication and disturbance as close as possible.

For using the power channel for communication, the mapping of a bit sequence $s_s = \{0,1\}^m$ into a sequence of symbols $\{\sigma_0, \sigma_1\}^m$ is called encoding: $\{0,1\}^m \rightarrow \mathcal{Z}^m, m \geq 0$ with the alphabet $\mathcal{Z} = \{\sigma_0, \sigma_1\}$. Here each bit $\{0,1\}$ is assigned to a symbol. Each symbol σ_i is a triple $(e_i, \delta_i, \omega_i)$, with the *event* $e_i \in \{\gamma, \bar{\gamma}\}$, the *period length* $\delta_i > 0$, and the *number of repetitions* $\omega_i > 0$. The event γ is *power consumption through a shift operation* and the inverse event $\bar{\gamma}$ is *no power consumption*. The period length is given in terms of a number of clock cycles. For example, the encoding through 32 shifts with the period length 1 (one shift operation per cycle) if the data bit '1' should be sent, and 32 cycles without a shift operation for the data bit '0' is defined by an alphabet $\mathcal{Z} = \{(\gamma, 1, 32), (\bar{\gamma}, 1, 32)\}$.

Impulse Response

Every linear, time invariant system or channel can be characterized by its *impulse response*. The impulse response $h(t)$ is the reaction of the system to an infinitely brief signal with an infinitely high amplitude – an impulse. Let $x(t)$ be the *input signal* and $y(t)$ the *output signal* of the system, then the output signal can be calculated by the *convolution* of $x(t)$ with the impulse response $h(t)$:

$$y(t) = \int_{-\infty}^{\infty} h(\tau) \cdot x(t-\tau)\, d\tau. \tag{3.36}$$

With known impulse response, the original signal $x(t)$ can be reconstructed from the received signal, if the signal/noise ratio is not too high. To determinate $h(t)$, an impulse must be generated on the input and the output must be measured. A *Dirac-pulse* $\delta(t)$ is defined as an infinitely short signal with an infinitely high amplitude [GSS64]:

$$\delta(t) = \begin{cases} \infty, & \text{if } t = 0 \\ 0, & \text{if } t \neq 0 \end{cases}, \tag{3.37}$$

$$\int_{-\infty}^{\infty} \delta(t)\, dt = 1. \tag{3.38}$$

To measure the impulse response in a real system, an approximation of the Dirac-pulse must be used. A single short and high *rectangular signal* is suitable in most cases.

For measuring the power watermarking communication channel, such a single short pulse is used. The power consumption, measured in terms of the voltage swing outside of the FPGA, should be used for this watermarking scheme. For this purpose, we use several power consumers which are all active only for a short time. A set of *shift registers* with common clock input is suitable for the generation of the impulse. To obtain a high amplitude, the toggle rate in each shift register must be maximized. This can be done by initializing the shift registers with the alternating bit sequence "01010101..." and feeding back the output to the input to perform a cyclic shift. If all shift registers are enabled for exactly one clock cycle, this causes a maximum power consumption in a minimal amount of time.

The resulting signal is recorded and digitized outside the FPGA using a *digital oscilloscope*. To reduce the noise and disturbances in the result, several responses of different impulses are recorded and then averaged. The resulting signal is the impulse response for this board and FPGA. Note that the impulse response may be different for each combination of FPGA and board, because the power switching characteristic depends mainly on different capacities inside the FPGA as well as on the board. Furthermore, the power supply circuit has also a high influence on the impulse response. Figure 3.19 shows a typical example impulse response for the Digilent Spartan-3 starter board [Dig].

Experimental results (see Section 3.5.5) show that the impulse responses of different boards look similar, but are not completely the same. Using *mathematical approximation*, the impulse response can be generalized to find a function which can be parameterized over as few parameters as possible and which hopefully covers all possible FPGA and board combinations. After transformation of the measured impulse responses into the frequency domain, only few independent frequency components can be identified in the spectrum. Each of those components is approximated through a *basic frequency component* and an *envelope function*. A good starting point for approximating the envelope is the *probability density* function of the χ^2_n*-distribution* [Bau08]. The χ^2_n-distribution is a common *continuous probability distribution* of a sum of squares of n independent random variables. The density function of the χ^2_n-distribution $P(n,t)$ is especially suitable for approximation, because it has only one extra parameter n besides the time t:

$$P(n,t) = \frac{t^{\frac{n}{2}-1} \cdot e^{\frac{-t}{n}}}{2^{\frac{n}{2}} \cdot \Gamma(\frac{n}{2})} \quad \forall t > 0, \tag{3.39}$$

where the gamma function $\Gamma(x)$ is defined as:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt. \tag{3.40}$$

For $x > 0$, the function can be written recursively:

$$\Gamma(x) = x \cdot \Gamma(x-1). \tag{3.41}$$

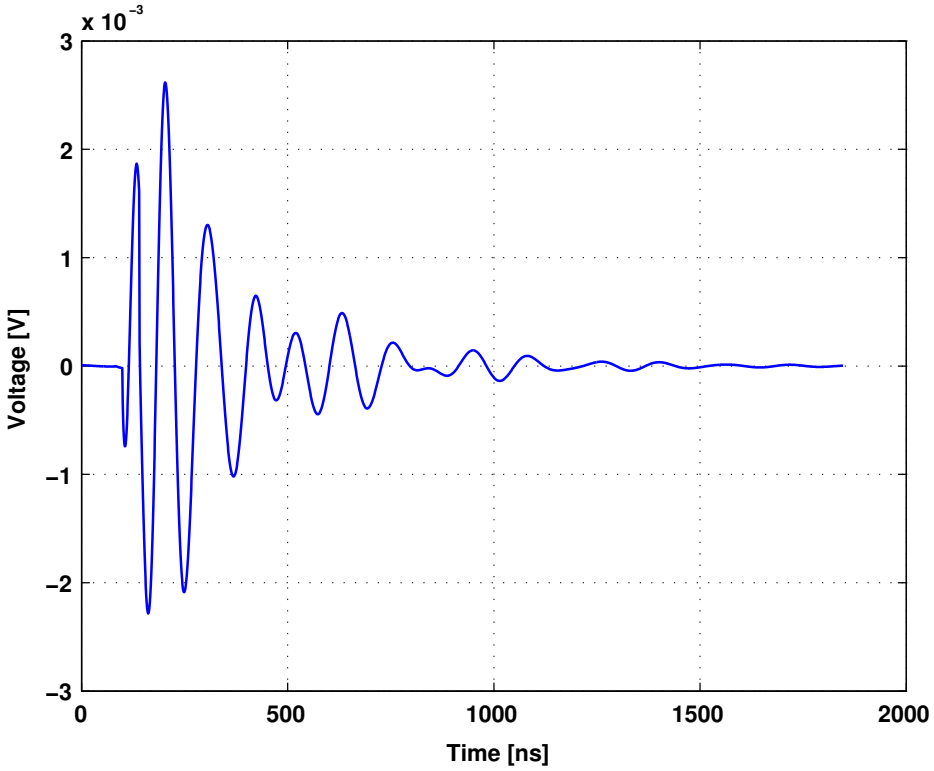


Figure 3.19: The impulse response obtained by a shift of a huge shift register, implemented using 128 SRL16 primitive cells in the Spartan-3 FPGA on the Digilent Spartan-3 starter board [Dig] with the experimental setup described in Section 3.5.5.

A frequency component $h_i(t)$ can now be approximated by:

$$h_i(t) = P(n_i,t) \cdot \alpha_i \cdot sin(2\pi f_i t + \phi_i), \tag{3.42}$$

with four parameters that need to be adapted: n_i, α_i, f_i, and ϕ_i. The approximated impulse response consists of the combination of l different frequency components:

$$h(t) = \sum_{i=1}^{l} h_i(t) \tag{3.43}$$

As our experimental results in Section 3.5.5 show, only two or three different frequency components are necessary to get a good approximation. The different parameters can be adapted using a *genetic algorithm* [Gor96] on the measured impulse response from a given FPGA and board combination. We used the minimization of the average square error as optimization goal. Note that the number of used shift registers may influence the impulse response as well. However, we show later that

the usage of more or less shift registers only influences the amplitude α of the signal which can be adapted by a constant factor.

Synchronization

For measuring the impulse response as well as for the decoding of the transmitted data later, several repeated equal data sequences can be accumulated to reduce the noise and disturbances which are not related to the sequence. For this approach, it is important that the individual sequences to be accumulated are exactly aligned over the time axis. This can be done by synchronizing the *oscilloscope sample clock* with the *shift clock* inside the FPGA.

Another way to get exact, noise-reduced data is by multiple oversampling inside the oscilloscope for recording the measurements. Here, a subsequent resynchronization on the measured data might be necessary.

To check this, the different sequences are plotted on top of each other. If the sequences match exactly, no resynchronization is necessary. If there is a shift over time, a resynchronization must be done. It is only necessary to look for the positions of the maxima if a sequence consists of only one event. A resynchronization is necessary if the positions of the maxima drift away over time and can be achieved by inserting or removing samples for compensating the deviation.

Channel Approximation

The measured signal $y(t)$ can be approximated with the help of the impulse response:

$$y(t) = c \cdot \left(\int_{-\infty}^{\infty} h(\tau) \cdot x(t-\tau)\, d\tau \right) + n(t). \tag{3.44}$$

Here, $x(t)$ is the input signal, consisting of Dirac-pulses which encode the signature. For each Dirac-pulse, a scaled impulse response is inserted. The *scaling factor* c depends on the amplitude of the pulse which therefore corresponds to the number of currently active shift registers. The assumption that the amplitude of the pulses depends linearly on the number of shift registers has been confirmed by experiment. Figure 3.20 shows the measured amplitude of the impulse response caused by different numbers of active shift registers.

Furthermore, the noise component $n(t)$ has influence on the decoding quality. The quality of the signal $y(t)$ can be measured by the *signal to noise ratio* (SNR):.

$$SNR = c \cdot \left(\int_{-\infty}^{\infty} h(\tau) \cdot x(t-\tau)\, d\tau \right) \cdot \frac{1}{n(t)}. \tag{3.45}$$

If the SNR is low, it is difficult or impossible to correctly decode the corresponding symbol. The SNR is therefore an indicator of how reliable the decoding of a given signal is.

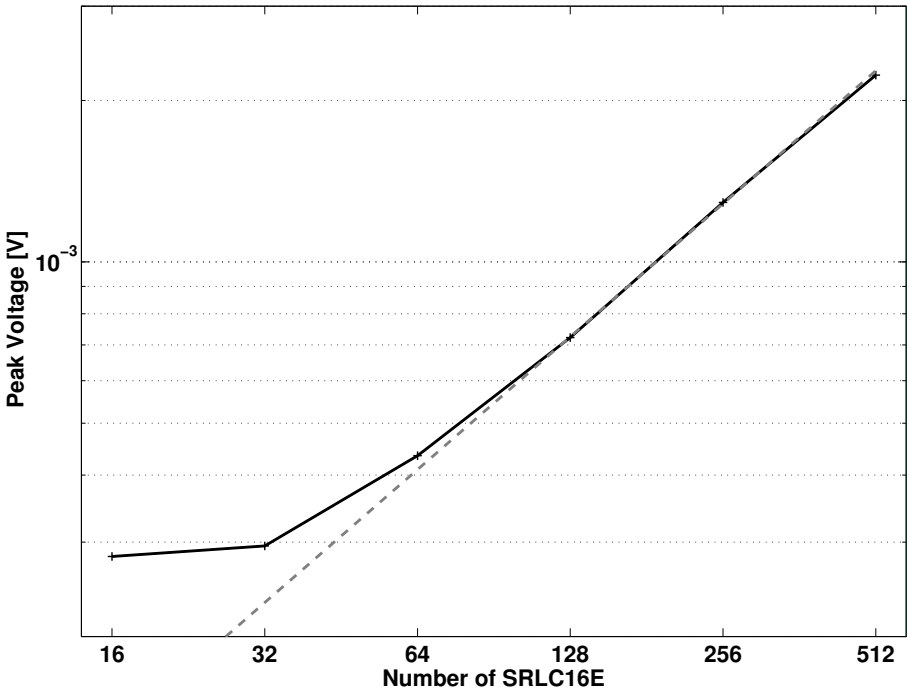


Figure 3.20: The amplitudes of the impulse responses caused by different numbers of active shift registers in the Spartan-3 FPGA on the Digilent Spartan-3 starter board. With more than 32 SRLC16E in use, the peak voltage is directly proportional to the number of shift registers. If less than 32 SRLC16E are used, the peak amplitude is in the same range as the noise amplitude, which falsified the measurement.

Intersymbol Interference

If the symbols superpose, for example when choosing a too small symbol time-lag, then *intersymbol interference* (ISI) occurs. This has the same effect as noise and decreases the quality of the decoding and results in a higher bit error rate. If a certain symbol σ_0 has the length τ_0 and the time slots for transmitting the symbols τ' are shorter than the symbol length ($\tau' < \tau_0$), then the symbol σ_0 interferes with $\left\lfloor \frac{\tau'}{\tau_0} \right\rfloor$ subsequent symbols.

A reconstruction of the symbol sequence might be possible nevertheless by decoding using the *Viterbi algorithm* [Vit95]. Hereby, the decoder stores all possible symbol paths in a so called *Trellis-diagram*. The symbol can only be decoded if it is received completely. However at this time, the successive symbols are already received partly if intersymbol interference happens. All possible symbol combinations

are stored in the Trellis-diagram, and if the decoder decides the value of the currently completely received symbol, all combinations using the other values for the current symbol are deleted from the diagram.

3.4.3 Basic Method

In this section, we describe the *basic method* for power watermarking of netlist cores. The concept, the embedding of the watermark, as well as the detection and verification procedure are described. The encoding and decoding for sending the signature through the FPGA power communication channel is relatively simple and straightforward in the basic method and will be refined later on with the enhanced methods. However, the basic concepts of embedding and the verification are very similar in all methods.

For power watermarking, two shift registers are used, a large one for causing a recognizable signature-dependent power consumption pattern, and a shift register storing the signature itself (see Figure 3.15 in Section 3.4.1). The signature shift register is clocked by the operational clock and the output bit enables the power pattern generator. If the output bit is a '1', the power pattern register will be shifted at the next rising edge of the operational clock. At a '0', no shift is done. Therefore, the channel encoding is $\mathcal{Z} = \{(\gamma, 1, 1), (\bar{\gamma}, 1, 1)\}$. To avoid interference from the operational logic in the measured voltage, the signature is only generated during the reset phase of the core.

As mentioned before in Section 3.3.5, a shift register can also be used as a lookup table and vice versa in many FPGA architectures (see Figure 3.12 in Section 3.3.5). A conversion of functional lookup tables into shift registers does not affect the functionality if the new inputs are set correctly. This allows us to use functional logic for implementing the power pattern generator. The core operates in two modes, the *functional mode* and the *reset mode*. In the functional mode, the shift is disabled and the shift register operates as a normal lookup table. In the reset mode, the content is shifted according to the signature bits and consumes power which can be measured outside of the FPGA. To prevent the loss of the content of the lookup table, the output of the shift register is fed back to the input, so the content is shifted circularly. When the core changes to the functional mode, the content must be shifted to the proper position to have a functional lookup table for the core.

The amplitude of the generated power signature depends on the number and content of the converted lookup tables. It will be assumed that the transitions between zeros and ones in the bit pattern of the lookup table contents are sufficient to produce a recognizable pattern on the supply voltage. Experimental results in [Bau08] show that, on average, 8 of maximal 16 transitions are generated in functional 4 input lookup tables of example cores if the content will be shifted.

To increase the robustness against removal and ambiguity attacks, the content of the power consumption shift register which is also part of the functional logic can be

initialized shifted. Only during the reset state, when the signature is transmitted, the content of the functional lookup table can be positioned correctly. So, normal core operation cannot start before the signature was transmitted completely. The advantage is that the core is only able to work after sending the signature. Furthermore, to avoid a too short reset time in which the watermark cannot be detected exactly, the right functionality will only be established if the reset state is longer than a predefined time. This prevents the user from leaving out or shorten the reset state with the result that the signature cannot be detected properly.

The signature itself can be implemented as a part of the functional logic in the same way. Some lookup tables are connected together and the content, the function of the LUTs, represents the signature. Furthermore, techniques described in Section 3.3.5 can be used to combine an additional watermark and the functional part in a single lookup table if not all lookup table inputs are used for the function. For example, LUT2 primitives in Xilinx Virtex-II devices can be used to carry an additional 12-bit watermark by restricting the reachability of the functional lookup table through clamping certain signals to constant values. Therefore, the final sending sequence consists of the functional part and the additional watermark. This principle makes it almost impossible for an attacker to change the content of the signature shift register. Altering the signature would also affect the functional core and thus result in a corrupt core.

The advantages of using the functional logic of the core also as a shift register are a reduced resource overhead for watermarking and the robustness of this method, because these shift registers are embedded in the functional design and it is hard, if not impossible, to remove the shift registers without destroying the functionality of the core. Furthermore, our watermarking procedure is difficult to be detected in a netlist file, because the main part of the required logic for signature creation depends on the functional logic for the proper core. Another benefit is that our watermark cannot be removed by an optimization step during the mapping into CLBs (Configurable Logic Blocks). Nevertheless, if an attacker had special knowledge of the watermarking method and of the EDIF netlist format, he may reverse-engineer the alteration of the embedding algorithm and remove or disable the sending method. This can be avoided by initializing the power pattern register with shifted lookup table contents (see above). If sending of the signature is prevented, the core will not function properly.

Embedding of the Watermark

In this section, we describe the procedure of watermarking a core. The first step is to generate the watermark W_L for embedding at the logic abstraction layer L. As described in the last section, the watermark is a bit sequence, consisting either of random choice bits, of partly functional bits of lookup tables, or completely of functional bits. The watermark generation procedure depends on the sequence type.

If only random choice bits are used, the watermark generated needs only the unique key K which identifies the author of the core: $\mathcal{G}_L(K) = W_L$. The unique key K can be processed as depicted in Figure 3.4 in Section 3.2.2. The pseudo random output can be split into different shift registers: $W_L = \{w_{L_1}, w_{L_2}, \ldots, w_{L_m}\}$. The number of used shift registers m depends on the strength of the generated signature and the FPGA architecture. For example, a 128-bit signature can be stored in the Virtex-II architecture in $m = 8$ shift registers.

If the content of functional lookup tables should be used as signature, the first step is to extract all lookup tables form the netlist core: $\mathcal{L}_L(I_L) = \{lut_{L_1}, lut_{L_2}, \ldots, lut_{L_r}\}$. The watermark generator $\mathcal{G}_L$ searches for suitable functional lookup tables, transforms these into shift registers and either adds the watermark bits form the pseudo random sequence $\mathcal{G}_L(K, \mathcal{L}_L(I_L)) = W_L$, or only uses the lookup table content as signature: $\mathcal{G}_L(\mathcal{L}_L(I_L)) = W_L$.

The watermark embedder $\mathcal{E}_L(I_L, W_L) = \widetilde{I_L}$ consists of two steps. First, the core I_L must be embedded in a wrapper which contains the control logic for emitting the signature. This step is done at the register-transfer level before synthesis. The second step is at the logic level after the synthesis. A program converts suitable lookup tables (for example LUT4 for Virtex-II FPGAs) into shift registers for the generation of the power pattern and attaches the corresponding control signal from the control logic in the wrapper (see Figure 3.21).

The wrapper contains the control logic for emitting the watermark and the shift register, holding the signature. If functional lookup tables are used for implementing the signature shift register, we add or convert this shift register in the second step so that the wrapper contains only the control logic. Some control signals have no sink yet, because the sink will be added in the second step (e.g., the power pattern generator shift register). So we must use synthesis constraints to prevent the synthesis tool from optimizing these signals away. The ports of the wrapper are the same for the core, so we can easily integrate this wrapper into the hierarchy. The control logic shifts the signature shift register, while the core is in reset state. Also, the power pattern shift register is shifted corresponding to the output of the signature shift register. If the reset input of the wrapper gets inactive, the function of the core cannot start at the same cycle, because the positions of the content in the shift register are not in the correct state. The control logic shifts the register content into the correct position and leaves the reset state to start the normal operation mode.

The translation of lookup tables of the functional logic into shift registers is done at the logic level. At Xilinx Virtex-II FPGAs, the usage of a LUT4 as a 16-bit shift register (SRL16) is only possible if the LUT4 is not part of a multiplexer logic, because the additional shift logic and the multiplexer share common resources in a slice. Also, if the lookup table is a part of an adder, the mapping tool splits the lookup table and the carry chain. In these two cases, additional slices would be required, so we do not convert these lookup tables into shift registers.

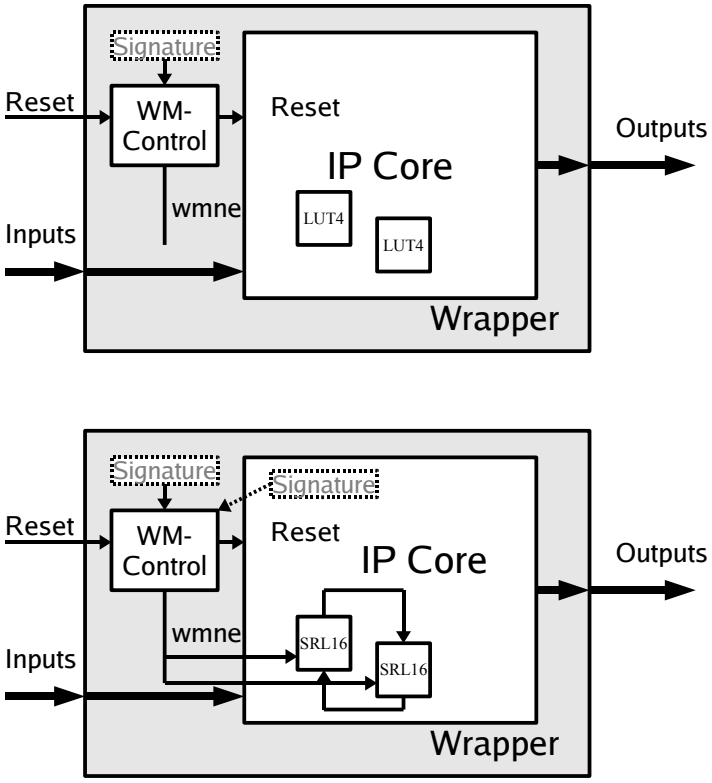


Figure 3.21: The core and the wrapper before (above) and after (below) the netlist alternation step. The signal "wmne" is an enable signal for shifting the power pattern generator shift register.

The embedding procedure for Virtex-II netlist cores is done by a program which parses an EDIF netlist and writes back the modified EDIF netlist. First, the program reads all LUT4 instances and only select those that are not a "MUXF5", a "MUXCY" or an "XORCY". Then, the instances are converted to a shift register (SRL16), if required, initialized with the shifted value and connected to the clock and the watermark enable (wmne) signal according to Figure 3.21. Always two shift registers are connected together to rotate their contents. Finally, the modified netlist is created. The watermarked core $\widetilde{I_L}$ is now ready for purchase or publication.

Detection Algorithm

A company may obtain an unlicensed version of the core $\widehat{I_L}$ and embeds this core in a product: $\widehat{I_P} = \mathcal{T}_{L\to B}(\widehat{I_L} \circ I'_{L_1} \circ I'_{L_2} \circ \dots)$. If the core developer has a suspicious fact, he

can buy the product and verify that his signature is inside the core using a detection function $\mathcal{D}_P(\widehat{I_P}, W_L) = true/false$.

Detecting the basic power watermark, the measured voltage will be probed, digitized and decoded by a signature detection algorithm (see Figure 3.22). To decode the digitalized voltage signal, the sampling rate, the clock frequency of the shifted signature and the bit length of the signature is needed. The clock frequency can be extracted using the *Fast Fourier Transformation* (FFT) of the measured signal. Our detection algorithm consists of five steps: *down sampling*, *differential step*, *accumulation step*, *phase detection* and *quantization* (see Figure 3.22). After successful extraction, the decoded signature can be compared to the signature inside the watermark W_L to establish the ownership. Furthermore, the signature must be verified by cryptographic methods with the author's unique key K. The key verification procedure is very similar to the bitfile watermarking methods presented in Section 3.3.

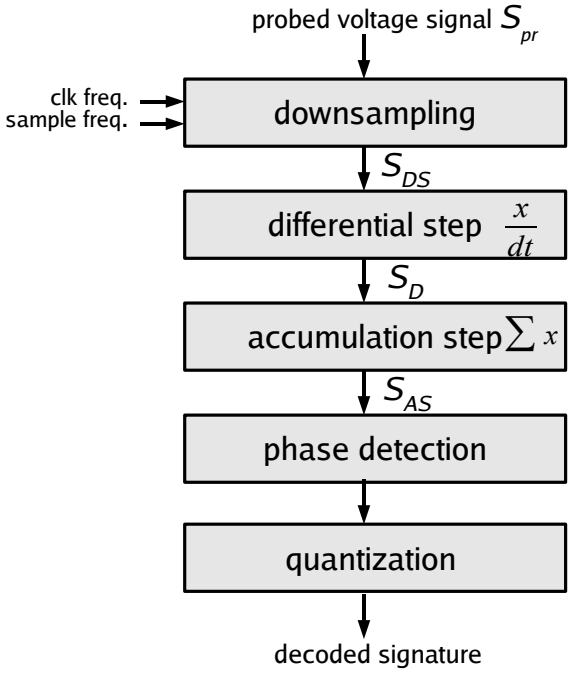


Figure 3.22: The five steps of the watermark detection algorithm: downsampling, differential and accumulation step, phase detection and finally quantization.

As mentioned before, the main characteristic caused by a switching event is the drop of the voltage followed by a subsequent overshoot. This results in extreme slopes. The basic method detection algorithm can find each rising edge as follows: First, the measured signal will be sampled down from the recorded sample rate to the

quadruple of the clock frequency, so each signature bit is represented by four samples. Then, the discrete derivative of the signal will be calculated. This transforms the rising edges of the switching events into peaks. The easiest way to calculate the discrete derivative at a discrete point in time $S_D[k]$ is to take the difference of two subsequent samples over time (see Figure 3.23).

$$S_D[k] = S_{DS}[k] - S_{DS}[k-1], \tag{3.46}$$

where S_{DS} is the down sampled probed voltage signal and k denotes the sample index.

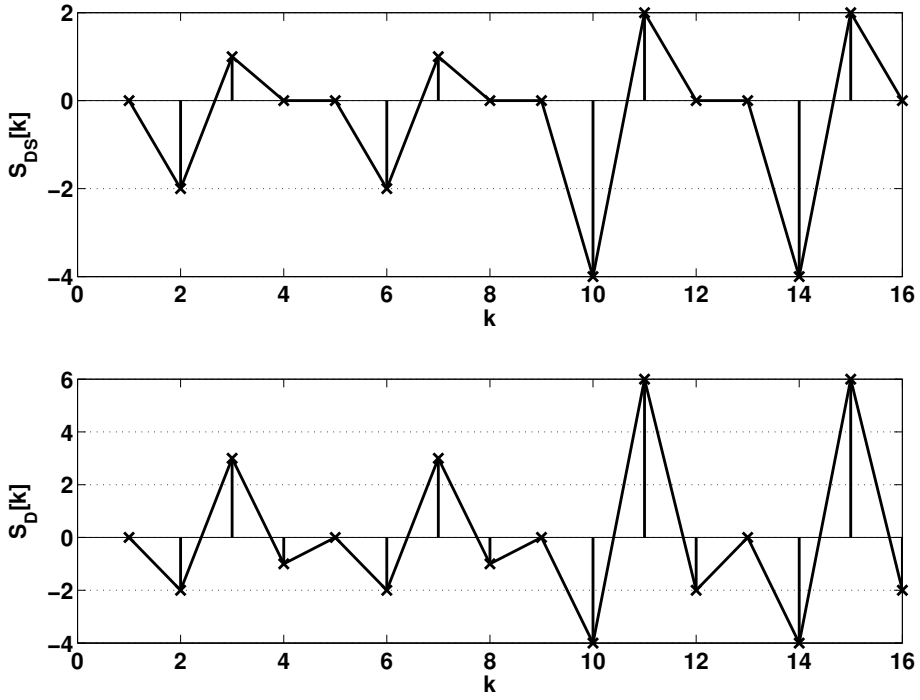


Figure 3.23: An example voltage signal which represents the signature "0011" (above). The example voltage signal after the differentiation step (below).

Since the signature is repeated many times during the reset state, the signal can be accumulated and averaged to reduce the noise level. To accumulate the coherent pattern, we need to know the bit length of the signature. If we record a longer signal sequence, we can accumulate more patterns and reduce noise as well as switching events which do not belong to the power consumption register of the watermarking algorithm. The disadvantage is that we would need a longer time for the reset phase.

After this third step, we have a signal in which each signature bit is represented by four samples. But only one sample carries the information of the rising edge. Since

the measurement is not synchronized with the FPGA clock, the phase (position) of the relevant sample of a bit is unknown. We divide the signal into four new signals, where one signature bit is represented in one sample. The four signals have a phase shift of 90^o to each other. Let

$$S_{AS}[k], \qquad k = 0,1,..,4m-1 \tag{3.47}$$

denote the sampled voltage signal after the accumulation step where m is the length of the signature. Then, we obtain the four following phase shifted signals

$$S_0 = S_{AS}[4k], \qquad k = 0,1,..,m-1 \tag{3.48}$$
$$S_{90} = S_{AS}[4k+1], \qquad " \tag{3.49}$$
$$S_{180} = S_{AS}[4k+2], \qquad " \tag{3.50}$$
$$S_{270} = S_{AS}[4k+3], \qquad " \tag{3.51}$$

where S_{AS} is the accumulated signal and S_0, S_{90}, S_{180}, and S_{270} are the phase signals (see Figure 3.24).

We are able to extract the right phase of the signal if we calculate the mean value of each phase-shifted signal. The maximal mean value corresponds to the correct phase, because the switching event should cause the greatest rising edge in the signal.

Now, we have a signal in which each sample is represented by the accumulated switching activities of one bit of the signature. The decision if the sample corresponds to a signature bit '1' or '0' can be done by comparing the sample value with the mean value of the signal. If the sample value is higher than the mean value, the algorithm decides a '1', in the other case a '0'.

Robustness Analysis

The most common attacks against watermarking mentioned in Section 3.2.1 are *removal, ambiguity, key copy*, and *copy attacks*. Once again, key copy attacks can be prevented by asymmetric cryptographic methods, and there is no protection against copy attacks.

Removal attacks most likely take place on the logic level instead of the device level where it is really hard to alter the design. The signature and power shift registers as well as the watermark sending control logic in the wrapper are mixed with functional elements in the netlist. Therefore, they are not easy to detect. Even if an attacker is able to identify the sending logic, a deactivation is useless if the content of the power shift register is only shifted into correct positions after sending the signature. By preventing the sending of the watermark, the core is unable to start. Another possibility is to alter the signature inside the shift register. The attacker may analyze the netlist to find the place were the signature is stored. This attack is only successful

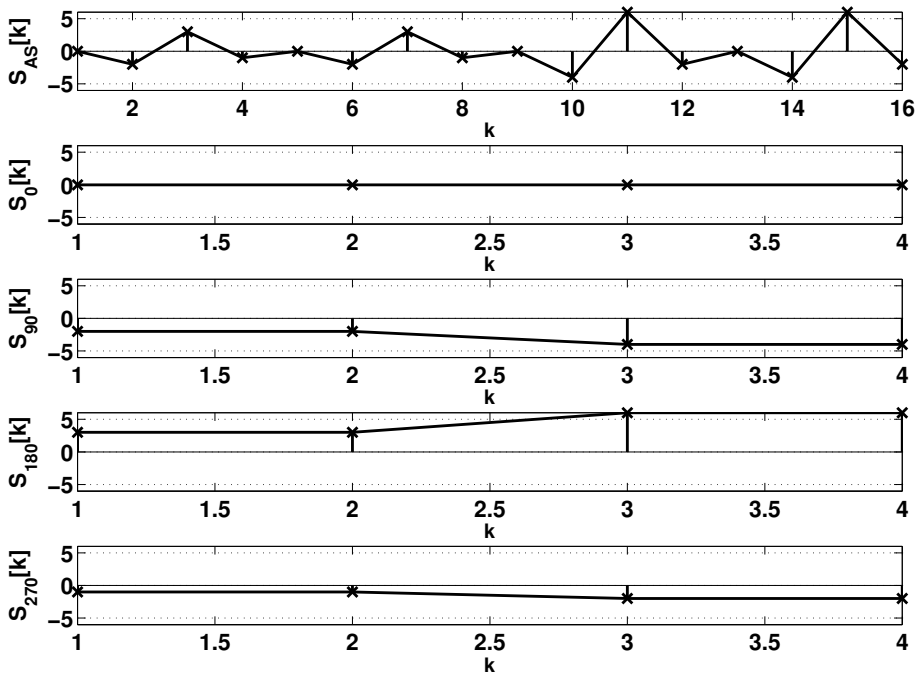


Figure 3.24: The example voltage signal after the accumulation step (above) and the four phase shifted signals (below). Here, S_{180} corresponds to the right phasing.

if there is no functional logic part mixed with the signature. By mixing the random bits with functional bits, it is hard to alter the signature without destroying the correct functionality of the core. Therefore, this watermark technique can be considered as resistant against removal attacks.

In case of *ambiguity attacks*, an attacker analyses the power consumption of the FPGA in order to find a fake watermark, or to implement a core whose power pattern disturbs the detection of the watermark. In order to trustfully fake watermarks inside the power consumption signal, the attacker must present the insertion and sending procedure which should be impossible without using an additional core. Another possibility for the attacker is to implement a *disturbance core* which needs a lot of power and makes the detection of the watermark impossible. In the next sections, *enhanced robustness encoding methods* are presented which increase the possibility to decode the signature, even if other cores are operating during the sending of the signature. Although a disturbance core might be successful, this core needs area and most notably power which increases the costs for the product. The presence of a disturbance core in a product is also suspicious and might lead to further investigation if a copyright infringement has occurred. Finally, the attacker may watermark another

core with his watermark and claim that all cores belong to him. This can be prevented by adding a hash value of the original core without the watermark to the signature like in the bitfile watermarking method for netlist cores. The sending of watermarks of multiple cores at the same time is addressed in Section 3.4.7.

3.4.4 Enhanced Robustness Encoding Method

Experimental results (see Section 3.5.5) have shown that the decoding of the signature with the basic method works well, but on some targets, problems occur in the decoding of signatures with long runs of '1' followed by many zeros, like "11111111-00000000". The problem is intersymbol interference (see Section 3.4.2), because the transmitting slot for one symbol in the basic method might be smaller than the symbol length. For the first eight bits, we see a huge amplitude in Figure 3.25. Then, a phase in which the amplitude is faded out is observed. The phase can last many clock cycles and may lead to wrong detection results of the following bits.

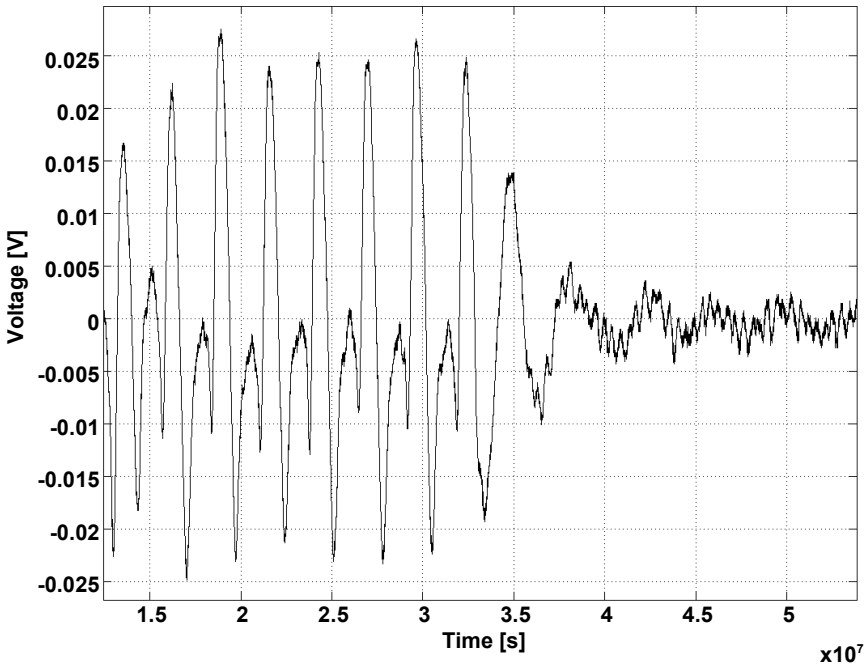


Figure 3.25: Measured voltage supply signal when sending "FFFF0000" with a large power pattern generator shift register.

This fading out amplitude belongs to an overlaid frequency which might be produced by a resonance circuit that consists of the capacitances and resistances from the

power supply plane and its blocking capacitances (see Section 3.4.2). This behavior is dependent on the printed circuit board and the power supply circuit.

To avoid such a false detection, the transmission time of one symbol is extended by the time of the swing out of the printed circuit board by sending the same signature bit multiple times: $\mathcal{Z} = \{(\gamma,1,\omega),(\bar{\gamma},1,\omega)\}$. The repetition rate for each signature bit is ω clock cycles. If we connect two SRL16 together, one period for this shift register needs 32 clock cycles. If the reset phase ends and we have finished sending one bit, the content in the shift register which also represents a part of the logic of the core is in the correct position.

The detection algorithm differs for this method. First, the signal will be sampled down and the approximate derivation will be calculated as in the original method (see Section 3.4.3). Now, we average the signal to suppress the noise. Here, the length of one signature word is the length of the signature (m) multiplied by the number of times each bit is sent (ω).

$$S_D[k], \qquad k = 0,1,..,K_{max}-1 \tag{3.52}$$

$$n_s = \left\lfloor \frac{K_{max}}{4\omega \cdot m} \right\rfloor, \tag{3.53}$$

$$S = \frac{1}{n_s} \sum_{i=0}^{n_s-1} D[4\omega \cdot m \cdot i,..,4\omega \cdot n \cdot i + 4\omega \cdot m - 1], \tag{3.54}$$

Here, S_D is the voltage signal after the differential step with index k and n_s being the number of repetitions of the pattern in S_D.

The phase detection of the shift clock is the same as in the original method (see Section 3.4.3), but we also need the position p where a new signature bit starts. This is done in a loop to detect this position. In the beginning, we assume that the starting position is the beginning of our trace ($p = 0$). First, we accumulate ω successive values where ω is the repetition of one bit:

$$S_p[j] = \sum_{i=0}^{\omega-1} S_\phi[i + p + \omega j], \qquad j = 0,1,..,m-1 \tag{3.55}$$

Here, S_ϕ denotes the signal after the phase detection step. Now, we subtract the mean value and generate the absolute value and calculate the sum of it.

$$F_p = \sum_{i=0}^{n-1} \left| S_p[i] - \frac{1}{n} \sum_{j=0}^{n-1} S_p[j] \right| \tag{3.56}$$

F_p identifies how good our signature bit starting position p fits the real position. Now, we shift our trace one value ($p = 1$) and calculate the fitting value again, and

so on. This is done ω times. The starting position with the best fitting value will be used.

The decoding of the watermark signature is done like in the basic method (see Section 3.4.3) by comparing the sample values with the mean value of the samples.

3.4.5 BPSK Detection Method

The enhanced robustness method introduced above works well, but if other cores with the same clock frequency have a very high toggle rate in the reset phase of the watermarked core, the quality of decoding may suffer. In the worst case, the decoding is not possible, because the watermarked signal is too weak in contrast to the interferences with the same frequency generated by the high toggle rate of the other cores (see the experimental results reported in Table 3.14 case *C arithmetic coder core*).

To enhance the robustness of decoding our transmit signal in case of interferences with the same frequency, we combine a new sending scheme with a new detection algorithm. The basic idea is to shift the carrier frequency of our watermarking signal away from the clock frequency of the chip, where we expect most of the interferences to occur.

We introduce a new binary signal S_{BPSK} with the frequency f_{BPSK}, where the signature bits are modulated using *Binary Phase Shift Keying* (BPSK) modulation. Using BPSK modulation, each value of a signature bit $(0, 1)$ is represented by a phase (usually 0° and 180°). Practically, by sending a '0', the carrier signal is not altered, and is inverted by sending a '1' (see Figure 3.26).

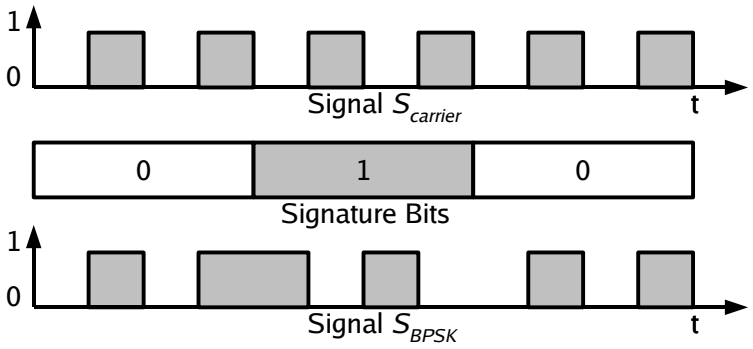


Figure 3.26: Shown is a carrier signal $S_{carrier}$ and the *BPSK* modulated signal S_{BPSK}. The signature bit value '0' is decoded with 0° and the value '1' is decoded with 180°.

We generate the new frequency f_{BPSK} by an *on-off keying* (OOK) modulation, a binary *amplitude modulation* (AM) of the clock frequency f_{clk}. So, the frequency

f_{BPSK} must be a rational fraction of the clock frequency f_{clk}. However, interferences from working cores have also an impact here, because these frequencies could also be produced by working cores with different toggle rates. The measurements suggest that frequencies $f = \frac{f_{clk}}{2^n}$ may have high interference from working cores due to whereas other frequencies have lower interference. The interferences decrease as well at lower derived frequencies. In the following, we choose $f_{BPSK} = \frac{f_{clk}}{10}$ as our carrier frequency.

To generate the new watermark signal, the power pattern generator is driven by the signal S_{BPSK} and performs the OOK modulation. The encoding scheme for the signal S_{BPSK} is: $\mathcal{Z} = \{(\gamma, 1, \omega), (\bar{\gamma}, 1, \omega)\}$, where ω is chosen 10 in our case. To send the signal S_{BPSK} for one period, we first send five ones (the power pattern shift register is shifted five times) and then five zeros (the power pattern shift register is not shifted) in case the signature bit is '1'. If the signature bit is '0', first five zeros and then five ones are sent (see Figure 3.27). For each signature bit, we repeat this period 32 times to ensure that the content of the power pattern shift registers which are also functional lookup tables are in the correct positions after sending one signature bit. Repetition allows to detect the signature with a higher probability. The decreased bit rate results in a smaller bandwidth for our watermarking signal. Using this method, we need more time to send the signature than the previously presented methods. The signature bit rate f_{wm} is:

$$f_{wm} = \frac{f_{BPSK}}{32} = \frac{f_{clk}}{10 \cdot 32} = \frac{f_{clk}}{320} \tag{3.57}$$

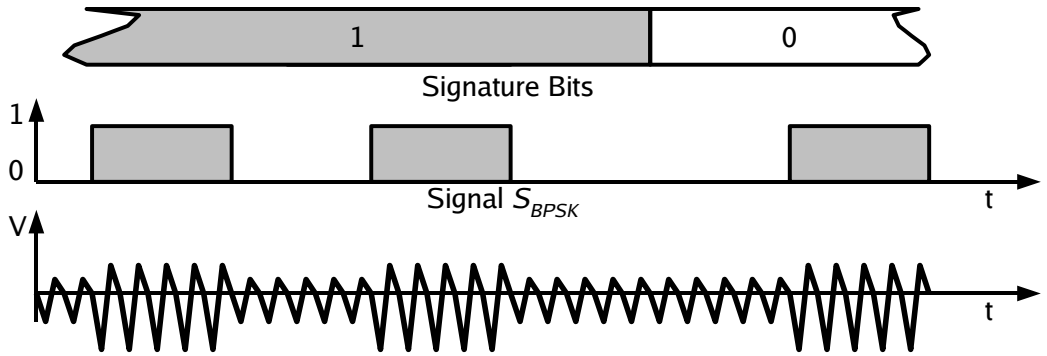


Figure 3.27: The signal S_{BPSK} is the *BPSK* modulated signal of the signature above. The signal below is the voltage signal which is the OOK modulated signal of S_{BPSK}. This figure also illustrate the different frequencies.

The watermark control inside the wrapper (see Section 3.4.3) is altered to control the power pattern generator in this way. Only few additional resources are used to implement this enhanced watermark protocol.

If we look at the spectrum of the recorded signal (see Figure 3.28), we detect the clock frequency f_{clk} and two side bands from the OOK modulation $f_{clk} - f_{BPSK}$ and $f_{clk} + f_{BPSK}$.

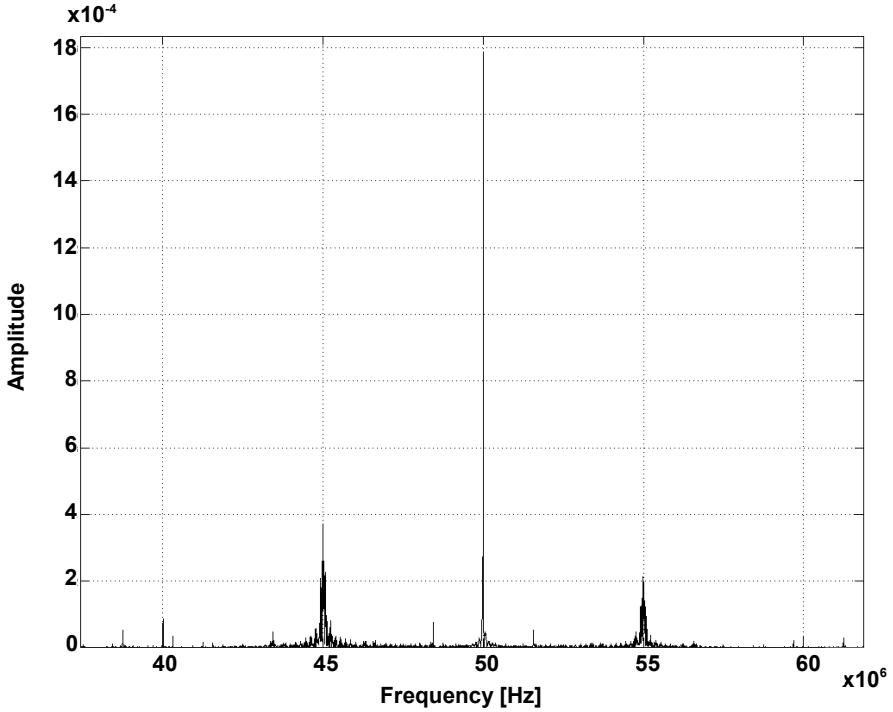


Figure 3.28: The spectrum of a measured signal. The clock frequency of $50MHz$ and the two side bands of the modulated signal S_{BPSK} are shown at $45MHz$ and $55MHz$.

The detection algorithm for this method is different from the previous methods. Only the first (down sampling) and the last steps (quantization) are identical (see Figure 3.29). After down sampling, the two side bands of the carrier signal are mixed down into the base band (S_{sb1} and S_{sb2}) and are combined (S_{cc}) as follows:

$$S_{DS}[k], \qquad k = 0, 1, .., K_{max} - 1 \tag{3.58}$$

$$S_{sb1}[k] = S_{DS}[k] \cdot e^{-j2\pi \cdot (\frac{1}{4} - \frac{1}{40}) \cdot k}, \tag{3.59}$$

$$S_{sb2}[k] = S_{DS}[k] \cdot e^{-j2\pi \cdot (\frac{1}{4} + \frac{1}{40}) \cdot k}, \tag{3.60}$$

$$S_{cc}[k] = S_{sb1} + S_{sb2}, \tag{3.61}$$

where S_{DS} is the voltage signal after down sampling with index k. The clock frequency is $f_{clk} = \frac{1}{4} \cdot f_{sample}$, and the frequency $f_{BSPK} = \frac{1}{10} \cdot f_{clk} = \frac{1}{40} \cdot f_{sample}$. The sample frequency of the recorded voltage signal is f_{sample}. After low pass filtering of S_{cc}, we get the complex carrier signal S_{BPSK} (see Figure 3.30).

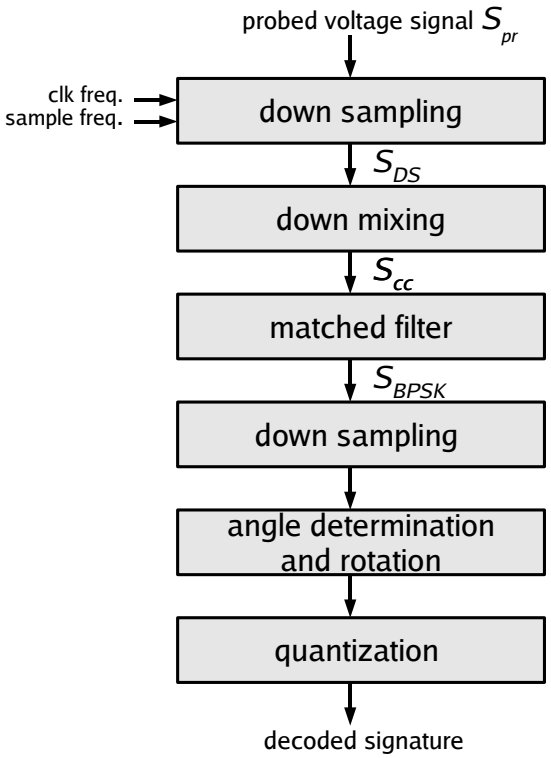


Figure 3.29: The different steps of the BPSK detection algorithm.

S_{cc} is filtered using a *matched filter* to obtain the limits of one signature bit and the correct sample point. All samples of S_{BPSK} which belong to one signature bit are summed up into this sample point by the matched filter. At the down sampling step, only these points are used to represent the signature bits. Now, the angle of the signal is calculated from the signature bit with the highest amplitude, and the signal is rotated into the real plane. From the real valued signal, the value of the bits and the quality of the signal are determined similar to the other detection algorithms (see Section 3.4.3).

The advantage of the BPSK method is its robustness with respect to interferences coupled with the clock frequency. The disadvantages are the longer reset phase and the fact that we can only detect bit value changes and not the signature bit value directly due to the BPSK modulation. Using proper encoding methods and preambles, the bit values can be reconstructed.

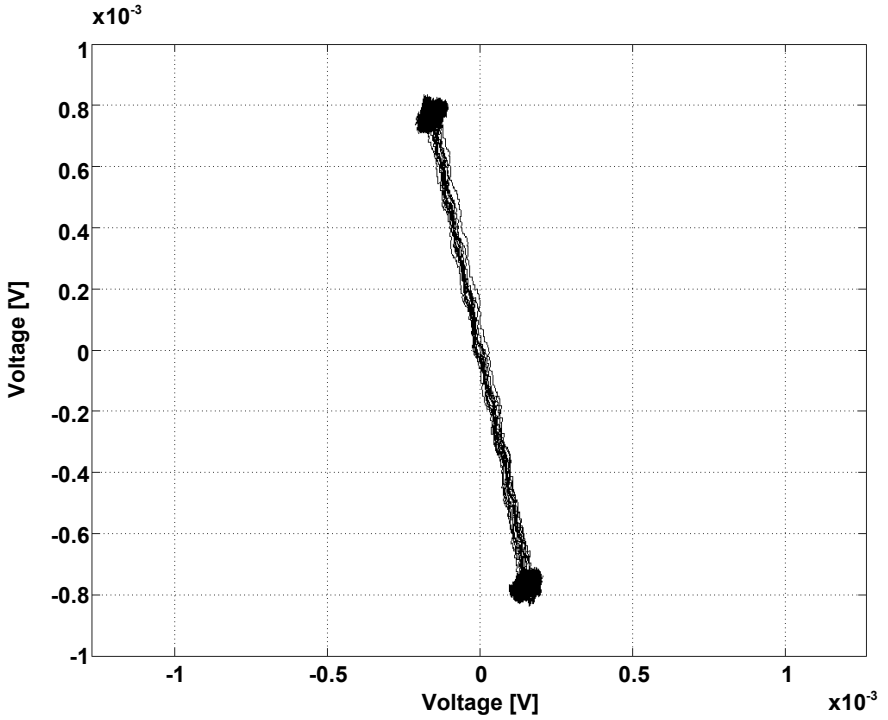


Figure 3.30: The *constellation diagram* of the down mixed complex signal S_{BPSK}. Here, the two different BPSK constellation points for the signature bit '1' and '0' are shown.

3.4.6 Correlative Detection Methods

In this section, we present methods where the signature is detected by correlation. To achieve good correlation results, the encoding of the signature should be interference-free. To avoid ISI, the time slots τ' for sending the symbols must be larger than the symbol length τ_σ. For the impulse response $h(t)$ to be usable directly for correlation, all symbols should consist only of one pulse. Therefore, we use the encoding: $\mathcal{Z} = \{(\gamma, \delta, 1), (\bar{\gamma}, \delta, 1)\}$. The symbol length is equal to the length of $h(t)$, and a time slot is $\tau' = \delta \cdot \frac{1}{f_{clk}}$. A necessary condition for an interference free code is: $\delta \geq \tau_\sigma \cdot f_{clk}$.

The question is how to estimate the symbol length τ_σ which is also the length of the impulse response. The length of $h(t)$ depends on the combination of FPGA and board which can be measured and approximated (see Section 3.4.2). However, the FPGA and board combination is not known at the embedding process of the watermarking, because it is up to the core customer to choose the FPGA and the board. Furthermore, the clock frequency is also not known. Clearly, there is an upper limit that is determined by the critical path of the netlist core. Due to these reasons, a safety margin should be considered. Experimental results in Section 3.5.5 show that

between 80 and 100 *ns* after the start of the symbol, 90% of the energy is emitted and after 125 *ns*, over 95%. If we assume clock frequencies $f_{clk} < 200\ MHz$, then the number of clock cycles $\delta = 25$ used for sending one symbol should be sufficient.

In this method, the detection of the signature is done by correlation. The *cross-correlation* $Z_{x,y}(t)$ of two function $x(t), y(t)$ is defined as:

$$Z_{x,y}(t) = \int_{-\infty}^{\infty} x^*(t) \cdot y(t+\tau)\, d\tau. \tag{3.62}$$

In Equation 3.62, $x^*(t)$ denotes to the complex conjugate of $x(t)$. The cross-correlation is a measure of the similarity of the functions $x(t)$ and $y(t)$, if $y(t)$ is shifted over the time axis t. The maximum of $Z_{x,y}(t)$ denotes the time with the highest similarity.

To detect a signature, we can use two different correlation methods. In the first method, the existence of a certain signature should be proven. From the adopted sequence, the expected signal is constructed by inserting the approximated impulse response $h(t)$ at times where the signature bit '1' should be sent, corresponding to the encoding scheme: $\mathcal{Z} = \{(\gamma, \delta, 1), (\bar{\gamma}, \delta, 1)\}$. The constructed signal is cross-correlated with the measured signal. If the maximum of cross-correlation is a distinctive peak, a watermarked core with this signature may be present. The disadvantage of this method is to define the notion of a distinctive peak. If the signature is not present in the core or another signature is present, the cross-correlation has also peaks (see Figure 3.31). It is hard to decide if the maximum peak belongs to the correct signature or not. Furthermore, if the expected and the actual present signature are very similar and differ only in a few bits, the maximum of the cross-correlation is also a distinctive peak which is only a little bit lower than the peak of the correct signature. Due to this problem, this approach does not seem to be applicable.

The second detection approach correlates the signal with the approximated impulse response $h(t)$. If the signature bit '1' was sent, then the correlation result has a peak at this position, otherwise, when a '0' was sent, no peak occurs. For better decoding, the signal and the impulse response are mixed down into the base band of one frequency component of $h(t)$, e.g., $h_1(t)$. Then, the mixed down signal and impulse response are correlated. Figure 3.32 shows such a correlation result. A possible decoding algorithm can look at the positions of the symbols in order to find peaks. If the first symbol position is known, the other positions can be calculated based on the encoding scheme and the clock frequency. The symbols on these positions can be decoded by making a threshold decision. If the signal value is higher than a certain threshold, the decoder decides on a '1', otherwise on a '0'. However, measurements shows that after transmission of several bits with the value '1', the absolute peak values are increasing slowly (see Figure 3.32). Therefore, the threshold value must be adapted dynamically based on the precedented values.

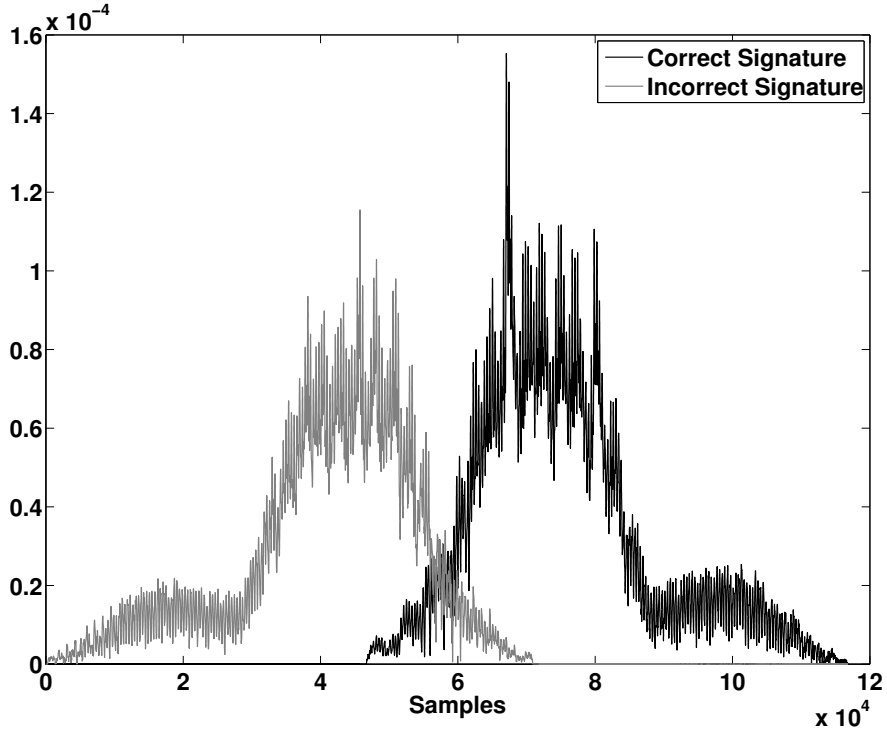


Figure 3.31: The cross-correlation $Z_{x,y}(t)$ of a signal with the correct signature (the right one) and with an incorrect random signature (the left one). A distinctive peak on the correct signatures can be detected. However, also the incorrect signature has peaks [Bau08].

Usually, the frequency component $h_i(t)$ of the impulse response with the highest energy content is used for decoding. Obviously, better results could be achieved if more frequency components are integrated into the decoding process. If this is necessary, the decoding is done for each component alone and the decoded data is compared.

The first symbol position in a data packet can be detected, e.g., by *synchronization methods*. In order to increase the detection ratio for the first symbol, we use a *preamble* to determine the correct start position of the signature. The preamble is a known bit sequence that has the same encoding as the user data and is transmitted directly before the signature.

Experimental results in Section 3.5.5 show that the decoding of the signature is possible with low bit error rates. However, other methods, like the BPSK method, exceed this method in case of bit error rates, particularly if disturbances from other cores are present. The reason is the low energy content of one symbol, caused by only one pulse in contrast to 32 pulses in the other methods. Nevertheless, this method

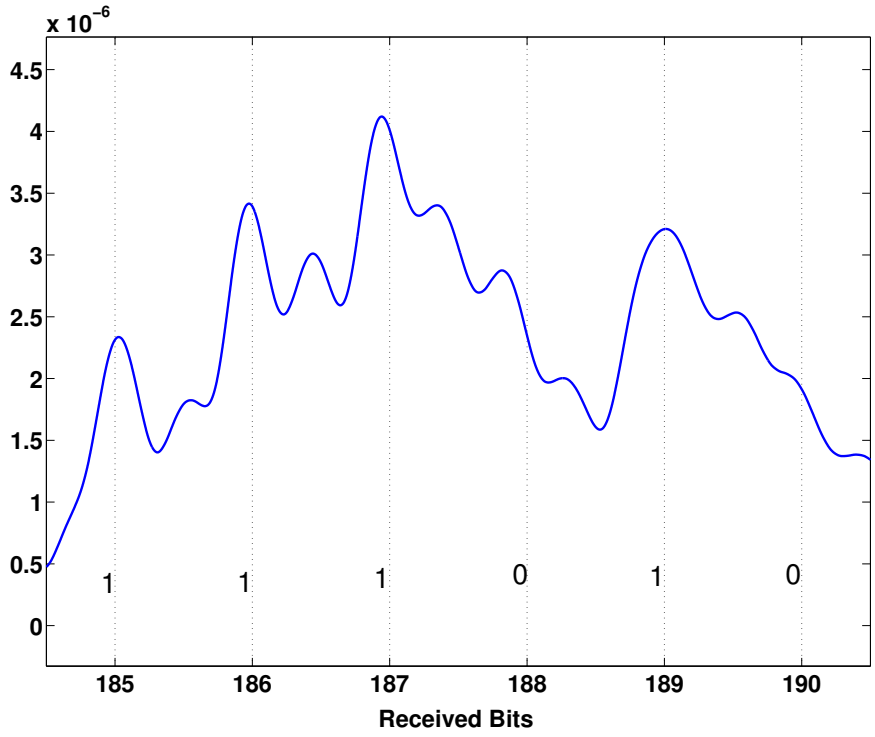


Figure 3.32: Detection of the data bits through down mixing into the base frequency of the impulse response component with the most energy content (here $f_i = 43\ MHz$) and correlation with the down mixed $h_i(t)$. Detection is done by searching the maxima and applying a dynamic threshold decision. The sample times for the decisions are depicted by dotted lines.

strictly avoids intersymbol interference and is more flexible for different FPGA and board combinations as well as for multiplexing methods where multiple cores can concurrently send signatures, discussed in the following section. More about decoding using the correlation method can be found in [Bau08].

3.4.7 Multiplexing Methods

All introduced power watermarking methods so far are applicable to netlist cores. The customers of netlist cores, the product developers, can combine different cores and integrate them into an FPGA design which is embedded into the product. Therefore, it is possible that more than one power watermarked cores are present in the design. The different sending mechanism of different cores do not know of each other and send their signature with the programmed encoding scheme. The results

are superpositions and interferences which complicate or even prohibit the correct decoding of the signatures.

To achieve the correct decoding of all signatures, we propose *multiplexing* or *multiple access* methods. Multiplexing methods divide the communication channel into multiple logical information channels – one channel for each transmitted signature. One can distinguish four different categories of multiplexing methods: *Space Division Multiplexing* (SDM), *Time Division Multiplexing* (TDM), *Frequency Division Multiplexing* (FDM), and *Code Division Multiplexing* (CDM).

Space Division Multiplexing

Characteristic for *space division multiplexing* methods is that the transport media for the different information channels are physically independent. For example, different isolated wires or antennas with directional radiation characteristics.

In case of power watermarking, this space division multiplexing can be implemented by measuring the voltage swing on different power pins. An FPGA has usually many power pins to effectively support the FPGA with power. However, inside the FPGA, these power pins are usually connected. Therefore, only small amplitude differences can be measured. One remedy is to exploit the idea that the location of the power shift register of a watermarked core has different distances to the power pins. Measurements on power pins which are near the shift register should have a higher amplitude as measurement on other power pins. If many watermarked cores are used, these are usually spread across the FPGA. Therefore, measurements on many power pins might successfully decode all signatures, even if sent simultaneously.

Unfortunately, experimental measurements on a Spartan-3 FPGA yields that the maximum amplitude difference between different power pins is less than 0.05 mV or 2% of the amplitude if 512 shift registers are used. Therefore, space division multiplex for power watermarking is not pursued.

Time Division Multiplexing

In case of *time division multiplexing*, the communication on the channel is divided into several time slots. Each peer to peer communication uses blocks which are sent in assigned time slots on the same communication medium. There exist two different categories of time division multiplexing techniques: *synchronous* (STDM) and *asynchronous time division multiplexing* (ATDM).

For STDM methods, each sender is assigned a fixed time slot. Furthermore, the definition of the time slot length and the assignment is done at design-time and must be known by all senders. If the assigned sender has no data to transmit, the corresponding time slot remains unused. ATDM methods use the time slots dynamically. A sender only occupies a time slot if data will be sent and therefore, the channel utilization is increased compared to STDM methods. Due to losing the fixed assignment

of the slots, the demultiplexer must know the receiver of the data. This is usually done by sending the receiver information over the channel, for example, in the header of the data. Therefore, these methods are also known as *address multiplexing methods*. The time slot length is either fixed or variable, depending on the used technique.

The problem of adapting the TDM approach for power watermarking is that the senders, the different power watermarked cores, have no synchronization possibilities. The reset signal might be used for synchronization, however, the different cores may use different resets or clocks and due to the power watermarking methods, the reset length might be further altered. Therefore, a probabilistic approach for time division multiplexing is chosen, where each signature is sent with a given period. During one period, the signature is sent once and after that, the power pattern generator inside the watermarked core is inactive until the period ends. The period length ϕ_i consists of the time for sending the signature $t_{sig,i}$ and the waiting time $t_{wait,i}$: $\phi_i = t_{sig,i} + t_{wait,i}$. By choosing different waiting times $t_{wait,i}$ for different cores, the sending time of the different signatures drifts away over the time and the probability of a successful decoding increases with a higher measurement time (see Figure 3.33). One constraint on this method is that every watermarked core has its own unique period length ϕ_i which should be relatively prime to the period length of the other cores in order to minimize the superposition of different signatures. If n_{max} is the overall number of all existing power watermarked cores, then

$$GCD(\phi_i, \phi_j) = 1 \qquad \forall i, j \in \{1, \ldots, n_{max}\}. \tag{3.63}$$

If all cores begin the transmission at the same start time, then the time span t_{dec} for the first collision-free decoding for two cores is:

$$t_{dec} = \phi_i \cdot \left\lceil \frac{t_{sig,i}}{\phi_j - \phi_i} \right\rceil, \quad \text{if } \phi_i < \phi_j, \tag{3.64}$$

and for three cores:

$$t_{dec} = LCM\left(\phi_i \cdot \left\lceil \frac{t_{sig,i}}{\phi_j - \phi_i} \right\rceil, \phi_i \cdot \left\lceil \frac{t_{sig,i}}{\phi_k - \phi_i} \right\rceil, \phi_j \cdot \left\lceil \frac{t_{sig,j}}{\phi_k - \phi_j} \right\rceil\right), \quad \text{if } \phi_i < \phi_j < \phi_k. \tag{3.65}$$

To estimate the period length for real systems, we must first know how long the time t_{sig} for sending the signature is. For the correlation method, the usual symbol length is 200 ns. If we assume a signature length of 32 bit plus 12-bit preamble, the time for sending the signature is: $t_{sig} = 8.8\ \mu s$. Furthermore, we need some time to shift the power shift register to the right position for the functional logic. For the enhanced robustness method, the time for sending is $\omega \cdot n \cdot f_{clk}^{-1}$. With $n = \omega = 32$ and $f_{clk} = 50\ MHz$, we need 20.48 μs for sending the signature. Therefore, a value

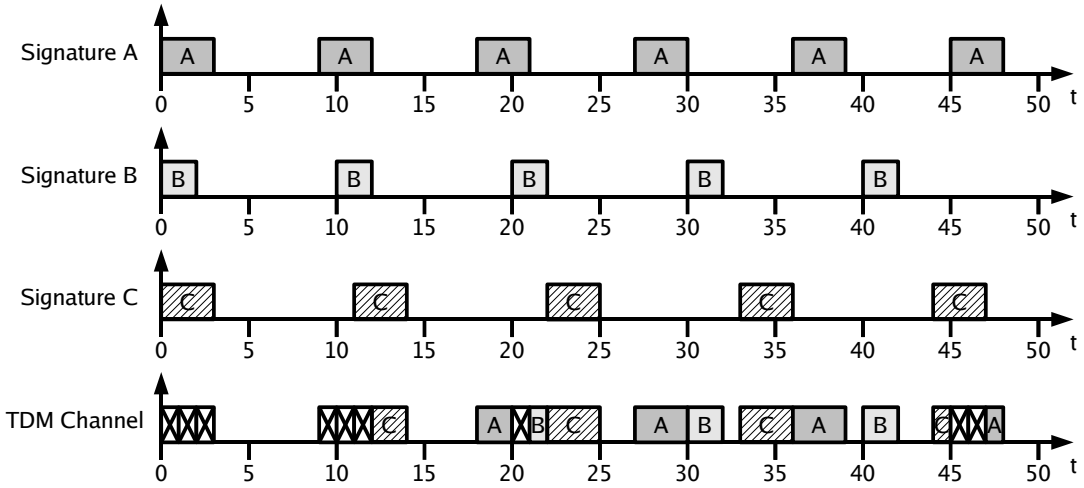


Figure 3.33: A schematic example for the application of the TDM method with the transmission of three different signatures of different lengths. The used periods are very short: $\phi_A = 9, \phi_B = 10$, and $\phi_C = 11$ slots. The time slots with collisions are marked on the TDM channel row below. The first collision-free sending of all signatures is after time step $t = 22$.

of 40 μs is a good approximation. One exception is the BPSK method. The sending time of a 32-bit signature there is 205 μs with $f_c = \frac{f_{clk}}{10}$ and $f_{clk} = 50\ MHz$.

On the other hand, the sending of the watermarks is limited by the minimal reset time, because all our power watermarking methods send the signature in the reset phase. If we assume a reset time of $t_{rst} = 100\ ms$, and we would like to send each signature at least 20 times for reasons of better decoding, then the maximum repetition period is $\phi_0 = 5\ ms$. For each additional signature, we reduce the period length by the approximated 40 μs for sending the signature: $\phi_1 = 4.96\ ms, \phi_2 = 4.92\ ms, \ldots$. Using this scheme, we have enough period times for signatures and a realistic reset time. Figure 3.34 shows a measurement of sending three different signatures using our TDM method.

Frequency Division Multiplexing

In case of *frequency division multiplexing* (FDM), the different information channels are assigned to different *carrier frequencies*. This can be done by modulating the information on a certain frequency. Different modulation methods, like *amplitude* (AM), *frequency* (FM), or *phase modulation* (PM) can be used. Furthermore, digital

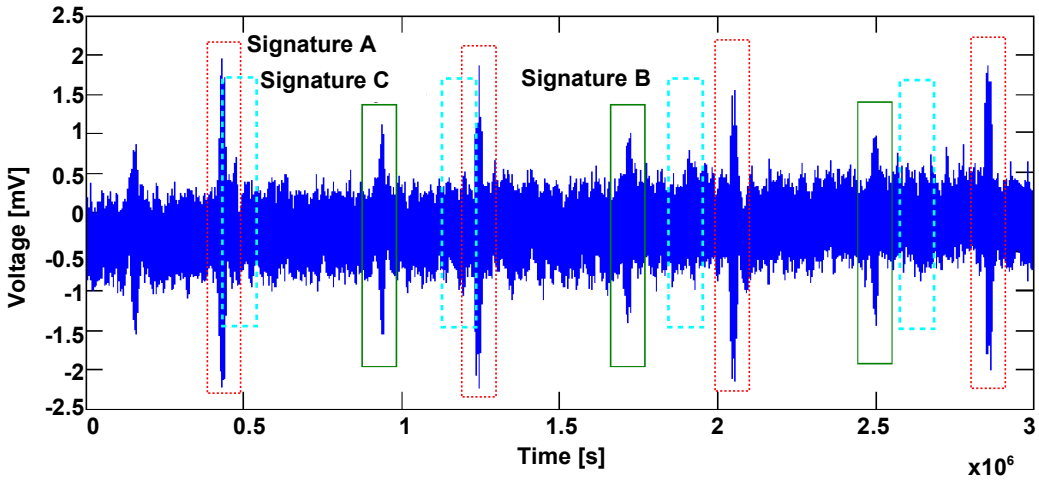


Figure 3.34: Shown is the superposed sending of three different signatures using the TDM method on a Spartan-3 FPGA. The minimal period difference between the signatures is approximately 16 μs. Signature A is produced by 64 SRL16 shift registers, signature B uses 40 SRL16 shift registers, and signature C uses 25 SRL16 shift registers. In the measurement, collisions and proper decodeable signature transmissions are highlighted.

signals are often transmitted using *shift keying modulation*, for example *Frequency Shift Keying* (FSK) or *Phase Shift Keying* (PSK).

The BPSK method, introduced in Section 3.4.5 uses an on-off keying (OOK) modulation to shift the carrier frequency for sending the signature away from the clock frequency. The aim of this modulation was the improved decoding, because most interferences from other cores are on the clock frequency or its divisions. However, using different carrier frequencies, more than one core can simultaneously send its signature. The information of the signature bits is embedded into the carrier signal S_{BPSK} by a BPSK modulation. Each watermarked core has its own carrier signal $S_{BPSK,i}$ which is OOK-modulated onto the clock frequency with different encoding schemes: $\mathcal{Z} = \{(\gamma, 1, \omega_i), (\bar{\gamma}, 1, \omega_i)\}$. By choosing different repetition rates ω_i, the different signals $S_{BPSK,i}$ are sent on different frequencies which enables a congruent sending of all signatures. However, the number of usable frequencies or repetition rates ω_i is constrained by the clock frequency and the sending time of the signature. Nevertheless, this is an interesting approach for further research.

Code Division Multiplexing

Finally for *code division multiplexing* methods, the transmitted data from different senders are spread up with different unique codes, so called *chips*. A chip is either

able to encode one bit or a sequence of bits [Vit95]. The chip encoded signals superpose on the communication channel. By knowing the chip sequences, all different transmitted data sequences can be reconstructed from the measured signal.

The widest used code division method is the encoding from *Hadamard and Walsh* [Vit90]. The data sequences are mapped on longer code sequences by multiplication with certain chip sequences. The different orthogonal chip sequences can be calculated by so called *Hadamard-Walsh functions* [Vit90]. The binary data sequence must be present in a *Non-Return-To-Zero* (NRTZ) code which means that the two bits are encoded with '1' and '-1'. To reconstruct a data sequence, the measured superposed signal is multiplied by the corresponding chip sequence and the average is calculated over the bit length. Figure 3.35 shows a simple example of transmitting two data sequences with a chip length of four. It is important for this method that all encoded data are synchronized to ensure a successful decoding.

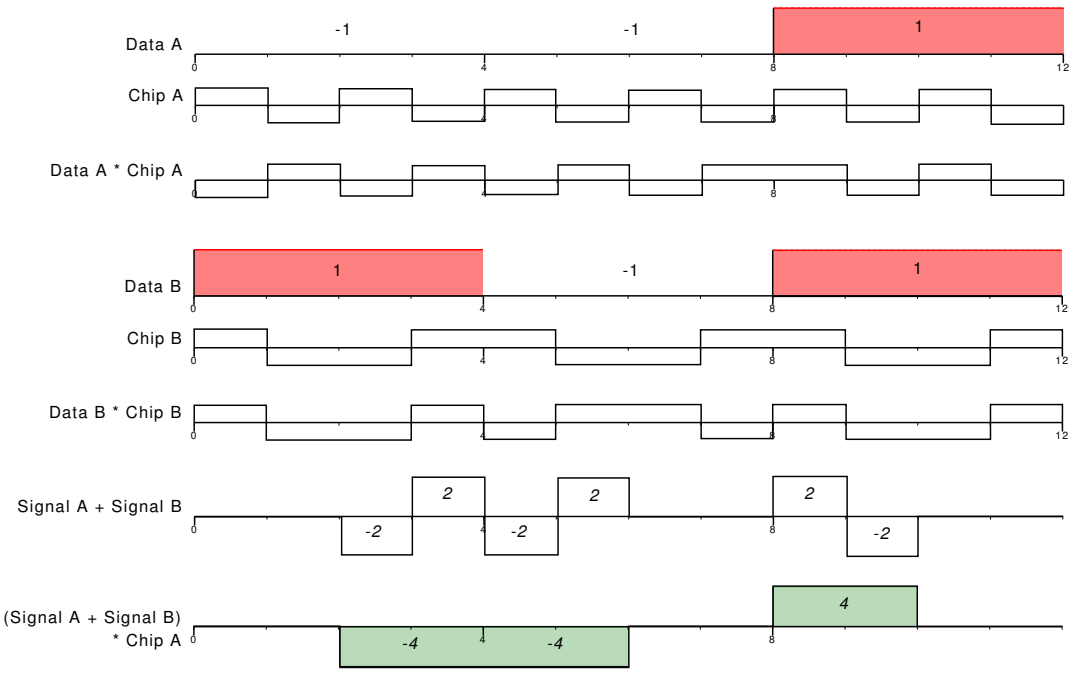


Figure 3.35: This illustration shows an example with the concurrent sending of two data sequences using CDM. The data is spread using different chip sequences and superpose each other on the communication channel. By multiplying the received signal with the corresponding chip sequence, the data can be reconstructed [Bau08].

Experimental results have shown that this method is very sensitive to interference [Bau08]. Furthermore, by using different amplitudes for each core due to different numbers of shift registers and the lack of synchronization possibilities for power

watermarking, CDM with Hadamard-Walsh encoding seems not to be applicable for concurrently sending different signatures.

Another CDM approach uses codes which are adopted from optical transmissions, the so called *Optical Orthogonal Codes* (OOC) [Sal89, SB89]. The main characteristic of these codes is the good cross- and autocorrelation capacity which makes them robust against shifting and suitable for asynchronous CDM [CSW89]. The codes consist of long runs of zeros separated by only few ones. By using these codes, the superposed signatures can be reconstructed like the Hadamard-Walsh-codes. The disadvantage is the longer chip sequence which results in longer sending times for the signatures. Experimental results in Section 3.5.5 show that the usage of CDM with OOC is applicable for power watermarking.

This section gave an overview of multiplexing methods suitable for power watermarking with many watermarked cores inside an FPGA sending simultaneous signatures. More details about these methods can be found in [Bau08].

3.5 Experimental Results

In this section, we present experimental results for the watermarking and identification methods presented in Section 3.3 and 3.4. The IP cores used for watermarking and identification were mostly obtained from *opencores.org* [Opec]. Netlist cores were synthesized using the Xilinx synthesis tool *XST* [Xili]. First, results of bitfile watermarking and identification methods are presented. Then, the results of the power watermarking methods are shown.

3.5.1 Identification of Netlist Cores by Analysis of LUT Contents

In order to give experimental evidence of the identification method for netlist cores presented in Section 3.3.2, we used a keyboard controller as an example [Opeb]. The design consists of four cores: *strober*, *producer*, *analyser*, and *fsm*. For all of these cores, the lookup table values and the corresponding unique functions were extracted from the netlists. The whole design was implemented on a Xilinx Virtex-II FPGA using the Xilinx tool *ISE 6.3i*. From the resulting bitfile, all lookup table values were extracted by the method described in Section 3.3.1. Each core was identified using the methods in Section 3.3.2 and the results are given in Table 3.2.

The results show that all lookup tables for the core *producer* and *strober* were found. During the implementation, many lookup tables for the core *analyser* were removed and are not found in the bitfile. This can result from unused outputs, or constant inputs. The core developer can evaluate which lookup tables are affected in these cases by comparing the implementation with reference designs to improve the interpretation of the results of the identification process. The mean distance to

Core	r	q	f_u	found r	distance d
producer + strober + fsm + analyser					
producer	40	450	3982	40	4.225
strober	93	450	3982	93	3.797
fsm	6	450	3982	5	0.883
analyser	379	450	3982	217	5.946

Table 3.2: Results for identifying individual cores in a design including four cores. The number of LUT functions used in the core r and in the design q as well as the number of different unique functions f_u are depicted. The values for mean distance to the core center d are in LUT positions.

the calculated core center in all four cases is small. In combination with the high percentage of found lookup tables, this result supports the assumption that the core is included in the bitfile. To verify the calculated core centers, we compare the core centers with the real placement of the slices in the cores. Figure 3.36 shows that the calculated core center positions correspond with the real positions of the cores.

To evaluate the robustness, we try to find the cores in a bitfile were they were not included. For this case, we implemented a *Des56* design [Opea] on a Xilinx Virtex-II FPGA and extracted all lookup tables. Table 3.3 shows that the percentage of the found lookup tables is low and the mean distance to the calculated core center is high. These values give evidence that the cores are not included in the design. For a more realistic scenario, we take two cores of the same functionality but from a different developer. For demonstration, a *cordic core* from *Xilinx Coregen* [Xilg] and *opencores.org* [Opec] are taken. We implement the Coregen core into the bitfile and search for the lookup table values from the opencores.org core. Table 3.3 shows that more than half of the lookup table contents were found, however, the distance is relatively high. This indicates that the opencores.org core is not included in the bitfile.

3.5.2 Identification of HDL Cores by Analysis of LUT Contents

To obtain experimental results for the HDL core identification technique, introduced in Section 3.3.3 we used several cores from *opencores.org* [Opec]: The cryptographic cores *Des56* and *3DES*, the processor cores *t400*, *minimips*, and *68hc08*, as well as the general IP cores *LCD*, and *colorconv*. Furthermore, we used different synthesis tools: *Xilinx Synthesis Technology (XST)* [Xili], *precision synthesis* from *Mentor Graphics* [Gra], and *Synplify Pro* from *Synopsis* [Synb]. We use the Xilinx *Spartan-3* as FPGA target technology.

Figure 3.36: The calculated core centers compared with the real LUT placements for the example cores shown in Table 3.2.

Core	r	q	f_u	found r	distance d
Des56					
producer	40	1574	3982	2	5.55
strober	93	1574	3982	44	15.97
fsm	6	1574	3982	3	12.597
analyser	379	1574	3982	69	13.865
cordic from Xilinx Coregen					
cordic from opencores	1052	1762	3982	633	26.39

Table 3.3: Results for identifying cores in a design where the cores are not included. First, the cores from the keyboard controller are searched for inclusion inside a Des56 core. The second result stems from trying to identify an opencores.org cordic core in a bitfile containing a Xilinx Coregen cordic core.

First, we take the same core, synthesized using different synthesis tools and analyze the different resulting lookup table contents. The first step is to map these lookup table contents into unique functions. For identification, these functions are labeled with the lowest content value of all lookup table contents which implement the same unique function. Table 3.4 lists, for example, the unique functions of the core *colorconv* synthesized with all three synthesis tools. It can be seen that the distribution of the most used unique functions is very similar. Only the functions generated by Synplify differ slightly from the other tools which can also be seen in the overall number of used lookup tables.

colorconv				
No. LUT inputs	unique function	XST	Precision	Synplify
1	2	57	54	60
2	2	0	18	2
2	6	312	309	207
2	8	0	0	203
3	28	0	0	99
3	2e	48	48	48
3	ac	144	126	144
4	0008	6	15	4
4	00ac	0	0	15
4	eefa	0	0	15
overall number of LUTs		579	581	820

Table 3.4: This table shows the different unique functions of the *colorconv* core synthesized with different synthesis tools. All lookup table contents which implement the same unique function are mapped to the lowest content value. Note that only functions which appear at least ten times after synthesis by one synthesis tool are depicted.

To test the *criterion method*, we first compare the netlists which implement the same core, but were synthesized with different synthesis tools. The results are shown in Table 3.5. The value $P_{a\ in\ b}$ denotes to the percentage of the unique functions from the core a which can be found in core b. If the criterion value C is bigger than one, then it is likely that these netlists were synthesized from the same HDL core description. This is true for all cores, except of the *LCD* core where the lookup table content distribution is different for all three synthesis tools.

Next, we compare the lookup table contents of netlists which are generated from different cores. We compare netlists synthesized with XST or Precision. As it can be seen in Table 3.6, the criterion value C is less than 1 for most cores which indicates

core	synthesis tool *a*	synthesis tool *b*	$P_{a\ in\ b}$	$P_{b\ in\ a}$	criterion value *C*
Des56	XST	Precision	48.05%	38.86%	**2.36**
Des56	XST	Synplify	55.75%	34.44%	**2.54**
3DES	XST	Precision	31.42%	28.46%	**1.12**
3DES	XST	Synplify	40.10%	26.78%	**1.40**
t400	XST	Precision	60.06%	50.31%	**3.81**
t400	XST	Synplify	54.52%	54.46%	**3.71**
minimips	XST	Precision	58.79%	36.10%	**2.81**
minimips	XST	Synplify	54.36%	36.62%	**2.59**
68hc08	XST	Precision	50.16%	41.22%	**2.61**
68hc08	XST	Synplify	41.90 %	37.42%	**1.97**
LCD	XST	Precision	20.69%	34.62%	**0.96**
LCD	XST	Synplify	18.50%	15.38%	**0.36**
colorconv	XST	Precision	93.80%	94.13%	**11.04**
colorconv	XST	Synplify	57.07%	80.83%	**5.94**

Table 3.5: The identification results of the criterion method, when the same core is synthesized using different synthesis tools. A criterion value of $C > 1$ indicates that the two netlists were synthesized from the same core description.

that the compared netlists were not generated from the same core. However, we also have three exceptions.

The experimental evaluation of the criterion method shows that it is possible to identify netlists which were generated from the same source using different synthesis tools in most cases. The negative as well as positive false detection shows that there is no guarantee of the correctness of this method. Nevertheless, this method can be useful to estimate if a netlist implements a specific core and trigger further analysis.

3.5.3 Watermarks in LUTs for Bitfile Cores

In this section, the experimental results for the bitfile watermarking method for bitfile cores, introduced in Section 3.3.4, are presented. Again, we used cores from opencores.org [Opec]: The *cordic*, *Des56*, and the *RSA* core. The first step is to examine if adding watermarks in bitfiles corrupts the functionality of the core. This was verified by comparing the output of the watermarked core to the original core.

This bitfile watermarking method embeds the watermarks into unused lookup tables in used slices. One question is if there are enough possible locations to insert the watermarks. Hereby, we implement the example cores in a Xilinx Virtex-II Pro FPGA XC2VP7 which provides 4928 slices in total. To achieve a certain degree of

core *a*	synthesis tool *a*	core *b*	synthesis tool *b*	$P_{a\ in\ b}$	$P_{b\ in\ a}$	criterion value *C*
Des56	XST	*3DES*	Precision	11.74%	42.02%	**0**
Des56	Precision	*3DES*	XST	13.63%	66.60%	**0**
t400	XST	*Des56*	Precision	18.55%	11.60%	**0.28**
t400	Precision	*Des56*	XST	12.64%	11.66%	**0.18**
Des56	XST	*minimips*	Precision	9.75%	48.03%	**0**
Des56	Precision	*minimips*	XST	7.35%	72.85%	**0**
t400	XST	*minimips*	Precision	17.32%	65.93%	**0**
t400	Precision	*minimips*	XST	10.99%	81.34%	**0**
t400	XST	*3DES*	Precision	5.16%	14.29%	**0.12**
t400	Precision	*3DES*	XST	3.76%	13.70%	**0**
minimips	XST	*3DES*	Precision	20.83%	9.30%	**0.28**
minimips	Precision	*3DES*	XST	32.53%	26.11%	**1.07**
68hc08	XST	*3DES*	Precision	8.08%	5.40%	**0.06**
68hc08	Precision	*3DES*	XST	16.59%	14.90%	**0.31**
68hc08	XST	*Des56*	Precision	42.19%	6.37%	**0**
68hc08	Precision	*Des56*	XST	44.08%	10.02%	**0**
68hc08	XST	*t400*	Precision	82.07%	16.61%	**0**
68hc08	Precision	*t400*	XST	72.77%	21.40%	**0**
68hc08	XST	*minimips*	Precision	36.11%	33.23%	**1.5**
68hc08	Precision	*minimips*	XST	26.39%	48.11%	**1.73**

Table 3.6: The comparison of netlists which are generated from different cores. The criterion value should be $C < 1$. Note, on many combination the criterion value $C = 0$, because the difference of the number of lookup tables between the two netlists is more than the average number of lookup tables (see criterion formula in Section 3.3.3).

diversity, five different versions of each core were created by modifying the core. The results in Table 3.7 show that there is not a real common link between the utilization and the available watermark locations. However, designs which are worth to watermark should provide enough possible locations to insert a sufficient number of watermarks.

3.5.4 Watermarks in Functional LUTs for Netlist Cores

In this section, we show experimental results for watermarking functional lookup table in netlist cores as introduced in Section 3.3.5. The results were measured using the same cores and the same FPGA target technology (XC2VP7) as for the experi-

cordic					
version	1	2	3	4	5
% FPGA utilization	8%	8%	9%	10%	11%
No. possible locations	15	14	15	17	17
Des56					
version	1	2	3	4	5
% FPGA utilization	19%	20%	20%	19%	19%
No. possible locations	305	332	406	395	174
RSA					
version	1	2	3	4	5
% FPGA utilization	11%	11%	11%	12%	12%
No. possible locations	171	164	165	172	160

Table 3.7: This table shows the number of possible watermark locations for the lookup table watermarking method for bitfile cores as introduced in Section 3.3.4. An interesting aspect is that the cryptographic cores have more possible locations than the cordic core even though similar FPGA utilization.

ments on watermarking for bitfile cores. The functionality of the watermarked cores was verified by simulation and on the FPGA device.

Using the example cores, timing and area overhead of our method were measured. Table 3.8 shows the properties of the unwatermarked cores. First, we examine the effect on the overheads by embedding a different number of watermarks. The watermarks are inserted into shift registers which are transformed functional lookup tables. For this measurement, we randomly chose LUT2 and LUT3 primitive cells for holding a part of the signature. For an easy verification, always four shift registers are chained together.

core	Timing (ns)	Resources (Slices)
cordic	5.909	369
Des56	6.220	965
RSA	13.557	540

Table 3.8: The critical path delay (timing) and the used resources of the unwatermarked cores implemented into the Xilinx Virtex-II Pro FPGA.

The results in Table 3.9 show that the resource overhead for the Des56 and RSA core is negligible. The decrease of resource usage may be explained by the observation that the mapping tool packs the chained shift registers more densely into the

available slices. The high increase of the critical path delay might be caused by the restriction of the optimization degrees due to the transformation of functional lookup tables into shift registers. The timing impact might be mitigated by performing a *timing analysis* and restricting possible watermark locations to non-critical paths.

No. Watermarks	32	64	128	256	384
cordic					
% Timing	39.19%	32.53%	64.05%	77.81%	78.76%
% Resources	4.97%	9.39%	17.98%	35.23%	52.57%
Des56					
% Timing	10.34%	6.98%	24.18%	18.71%	29.8%
% Resources	-1.28%	0.28%	0.48%	-0.17%	-3.83%
RSA					
% Timing	10.11%	55.42%	57.20%	29.93%	44.37%
% Resources	-0.74%	2.78%	1.91%	9.07%	9.75%

Table 3.9: The timing and resource overhead of the example cores with different number of inserted watermarks. Always four of the transformed shift registers are chained together.

Next, we investigate the effect of different shift register chain lengths. In the last experiments, we chained four watermark shift registers together. For the measurement, we only use the Des56 core to examine the effect of various chaining lengths on timing and resource overhead. Additionally, we vary the amount of inserted watermarks by converting different percentages of the available lookup table candidates into watermarked shift registers. The results in Table 3.10 show that converting around 30% of the lookup table candidates into shift registers achieves the best timing behavior. However, this is only an exemplary result and cannot be generalized. Furthermore, the impact of the chain length is negligible, except for shift registers with chain length 1, which leads to an increased resource overhead. Notable is the decreased resource overhead, if more shift registers are used, except these shift registers which are chained only with themselves (chain length 1). A possible explanation for this behavior is that the tools pack these shift registers more densely into the available slices.

Finally, we investigate the verification capability of the approach. We measure for the same test setup as the measurement in Table 3.10 (different chain length and different numbers of watermarks), how many watermarks can clearly be identified and how many indeterminable duplicates there are. Indeterminable duplicates are watermarks which exist more than one time in the bitfile. It is impossible to determine which one is the watermark and which one is similar to the watermark by chance.

% Shift Registers	10%	20%	30%	40%	50%	70%	90%
Chain Length 1							
% Timing	18.2%	12.3%	9.1%	36.8%	39.%9	36.2%	14.0%
% Resources	0.2%	3.0%	4.3%	5.1%	6.7%	8.6%	14.5%
Chain Length 4							
% Timing	15.4%	12.3%	3.2%	27.7%	25.5%	37.7%	23.2%
% Resources	1.2%	-0.5%	0.3%	-0.2%	-5.0%	-7.0%	-5.7%
Chain Length 8							
% Timing	15.2%	8.3%	5.4%	26.3%	25.7%	29.4%	20.8%
% Resources	0.9%	1.0%	2.4%	-0.4%	-2.5%	-5.7%	-4.9%
Chain Length 16							
% Timing	14.4%	9.5%	5.7%	23.7%	28.4%	27.3%	19.2%
% Resources	2.6%	1.8%	2.8%	-2.5%	-3.3%	-5.6%	-5.2%
Chain Length All							
% Timing	15.7%	13.2%	7.3%	30.0%	25.5%	36.8%	20.1%
% Resources	1.5%	1.8%	3.4%	-3.6%	-4.1%	-6.8%	-6.3%

Table 3.10: The effect of timing and resource overhead at different chain length of watermarked shift registers and at different amounts of inserted watermarks for the *Des56* core.

The results in Table 3.11 show that chaining the shift registers helps to avoid indeterminable duplicates.

3.5.5 Power Watermarking

In this section, measurements and experimental results for the power watermarking methods as introduced in Section 3.4 are presented. First, the experimental setup and measurements for the communication channel are provided. Then, the results of the different methods are shown and finally, the results of the multiplexing methods are presented.

Experimental Setup

In the following experiments, we used two FPGA-boards, the Digilent Spartan-3 Starter Board [Dig], and a board equipped with a Xilinx Virtex-II XC2V250 FPGA. On the second board, many other components such as an ARM micro-controller and interface chips are integrated to demonstrate that the algorithm is also working on multi-chip boards. The Spartan-3 board operates at a clock frequency of 50 MHz, the Virtex-II board at 74.25 MHz.

% Shift Registers	10%	20%	30%	40%	50%	70%	90%
Chain Length	% Duplicates						
1	4.41%	6.62%	5.76%	3.47%	3.48%	2.32%	1.76%
4	1.10%	1.29%	2.21%	1.30%	0.71%	0.91%	0.66%
8	0%	0.37%	1.47%	1.13%	0.64%	0.76%	0.59%
16	0%	0.18%	0.49%	1.39%	0.43%	0.60%	0.43%
All	0%	0%	0.37%	0.70%	0.50%	0.61%	0.20%

Table 3.11: The percentage of indeterminable duplicates for the watermark verification, compared to different chain length and different number of inserted watermarks.

On both boards, the voltage is measured on the back of the printed circuit board directly on the via which connects the FPGA with the power plane of the printed circuit board. We used a 50 Ω wire with a 50 Ω terminating resistor soldered directly on the vias (see Figure 3.37). We have used a *DC block* element and a 25 MHz high pass filter to filter out the DC component and the interferences of the switching voltage controller. We used a *LeCroy Wavepro 7300 oscilloscope* with 20 Giga Samples per second to measure the voltage. The voltage amplitude of the measured switch peak is very small, so we used a *digital enhanced resolution filter* to improve the dynamics, at the cost of a decreased bandwidth. The signal of the length of 200 μs is recorded on the internal hard disc of the oscilloscope. This trace file is then transferred to a personal computer and analyzed there.

Communication Channel

First, we measured the impulse responses of the two boards of the experimental setup. The impulse is generated using a shift of 64 *16-bit* shift registers (SRL16) inside the FPGA which are initialized with a "010101..." pattern to achieve the highest possible toggle rate. Figure 3.38 shows the measured impulse responses of the two boards. It can be seen that the two impulse responses are very similar. The impulse responses can be approximated using the following formula from Section 3.4.2:

$$h_i(t) = P(n_i,t) \cdot \alpha_i \cdot sin(2\pi f_i t + \phi_i), \tag{3.66}$$

$$h(t) = \sum_{i=1}^{l} h_i(t), \tag{3.67}$$

where l is the number of different frequency components.

The approximation of the impulse responses of the two boards was done using a genetic algorithm and for three frequency components ($l = 3$) each. The resulting

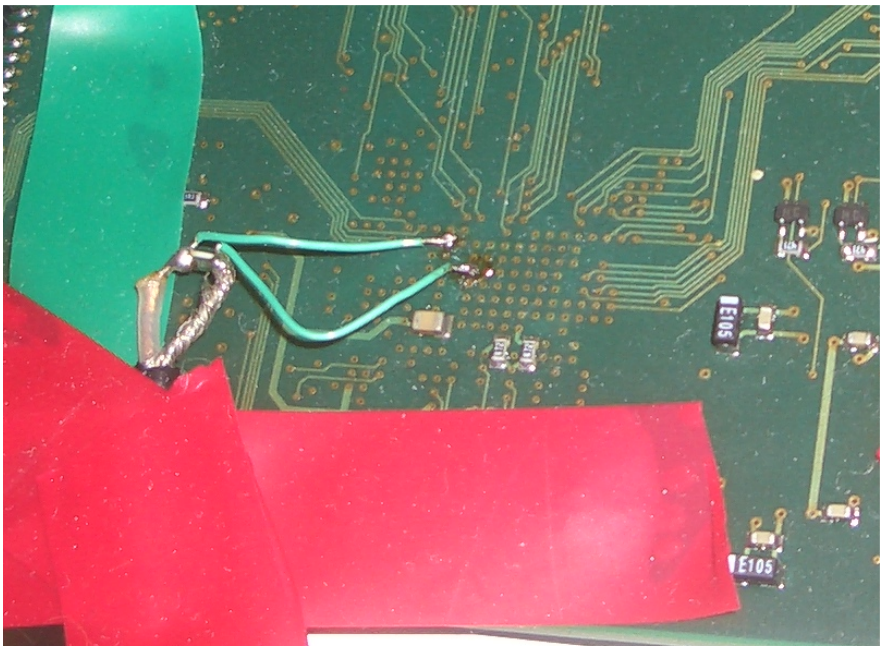


Figure 3.37: The rear side of the Virtex-II board with the soldered wires on the power vias. Furthermore, the 50 Ω terminating resistor can be seen between the wires and the coaxial cable.

parameters α, f, ϕ, and n are shown in Table 3.12. The resulting curves with the approximation errors are depicted in Figure 3.39 for the Spartan-3 board and in Figure 3.40 for the Virtex-II board.

The measurements for the synchronization between the clock frequency of the board and the sample frequency of the oscilloscope depicted that the sample drift for the Spartan-3 board is not too high for power watermarking, whereas for the Virtex-II board, we need a subsequent resynchronization (see Figure 3.41). This was done by removing sample values for each symbol to lower the drift into a specified range.

Basic Method

The functionality of our proposed watermark detection methods is evaluated for a *Des56* core from *opencores.org* [Opea] and an *arithmetic coder core*.

After the synthesis step, only 16 out of 715 lookup tables from the *Des56* core have been transformed into SRL16 and an $n = 32$ Bit signature has been added. Also, for the arithmetic coder core, 92 out of 1332 lookup tables have been transformed into SRL16 cells. Both cores' inputs were stimulated using a pseudo random sequence generated by a linear feedback shift register to simulate input data.

The decoded sequence was compared with the encoded signature from the core to evaluate the bit error rate. Furthermore from the bit decision signal, two quality

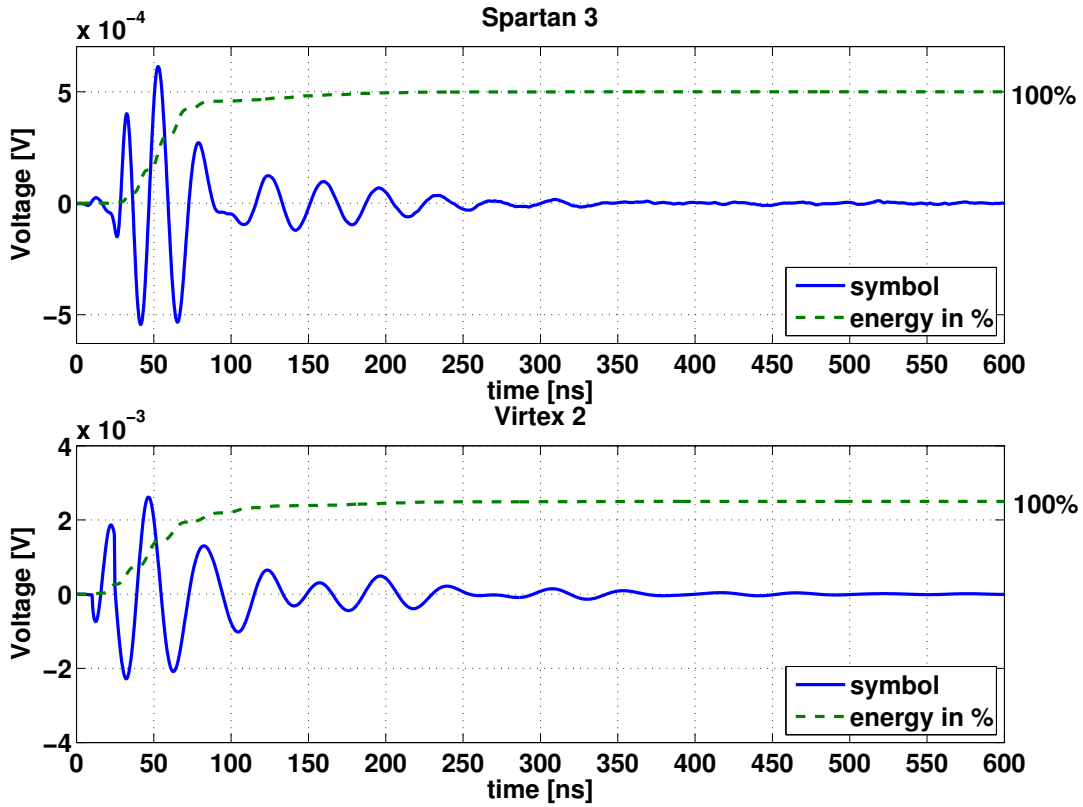


Figure 3.38: The measured impulse responses of the Spartan-3 Board (above) and the Virtex-II Board (below). The impulse responses were averaged over many single impulse responses on each board. Furthermore, for each impulse response the percentage of the energy content is shown over the time [Bau08].

	α	f	ϕ	n
Spartan-3 Board Approx. Error: 1.5%				
$h_3(t)$	0.56 mV	43.0 MHz	1.7π	11.3
$h_2(t)$	0.17 mV	26.9 MHz	1.6π	4.49
$h_1(t)$	0.16 mV	75.7 MHz	1.5π	19.2
Virtex-II Board Approx. Error: 4.7%				
$h_3(t)$	1.96 mV	36.5 MHz	0.06π	12.6
$h_2(t)$	1.49 mV	26.6 MHz	1.1π	12.3
$h_1(t)$	1.02 mV	66.5 MHz	1.7π	23.8

Table 3.12: The approximation values and errors for $h(t)$ according to Equation 3.66 and 3.67 for the Spartan-3 and Virtex-II board.

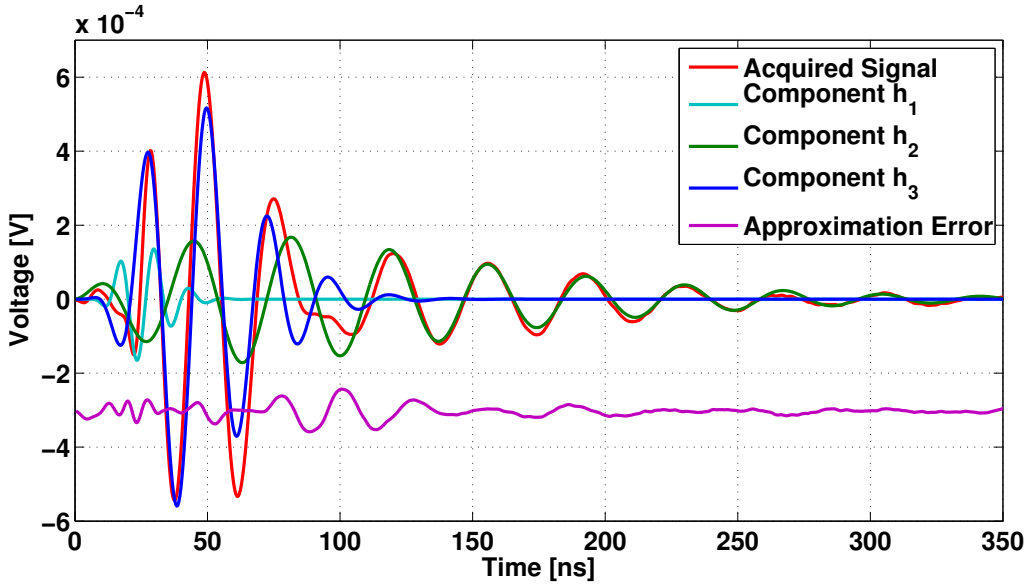


Figure 3.39: The approximation of the impulse response for the Spartan-3 board with three frequency components and the approximation parameters from Table 3.12. The approximation error is shifted to $-0.3\ mV$ for a better readability of the diagram [Bau08].

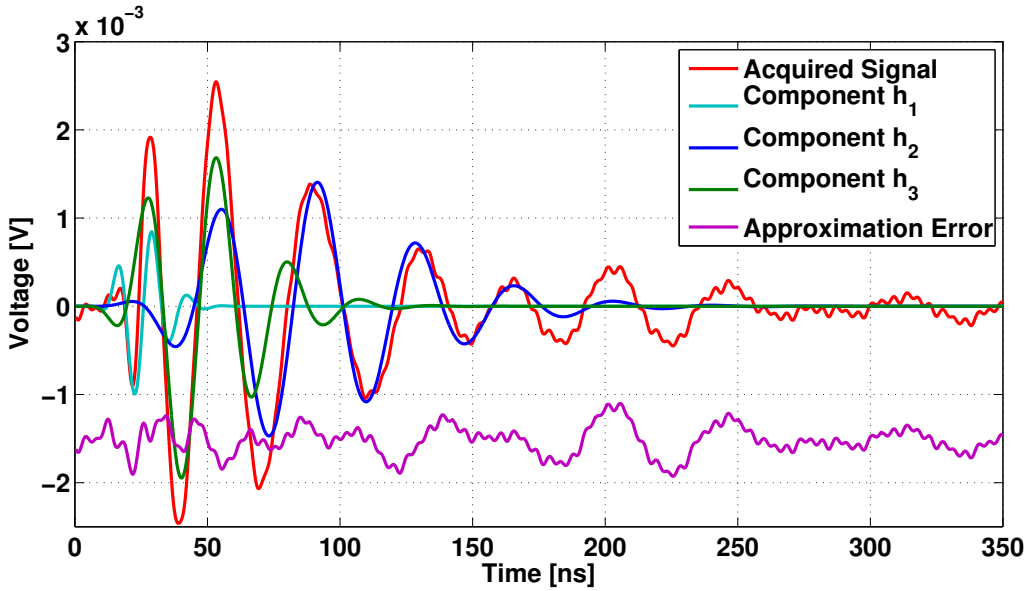


Figure 3.40: The approximation of the impulse response for the Virtex-II board with the approximation parameters from Table 3.12. The approximation error is shifted again to $-0.3\ mV$ for a better readability of the diagram [Bau08].

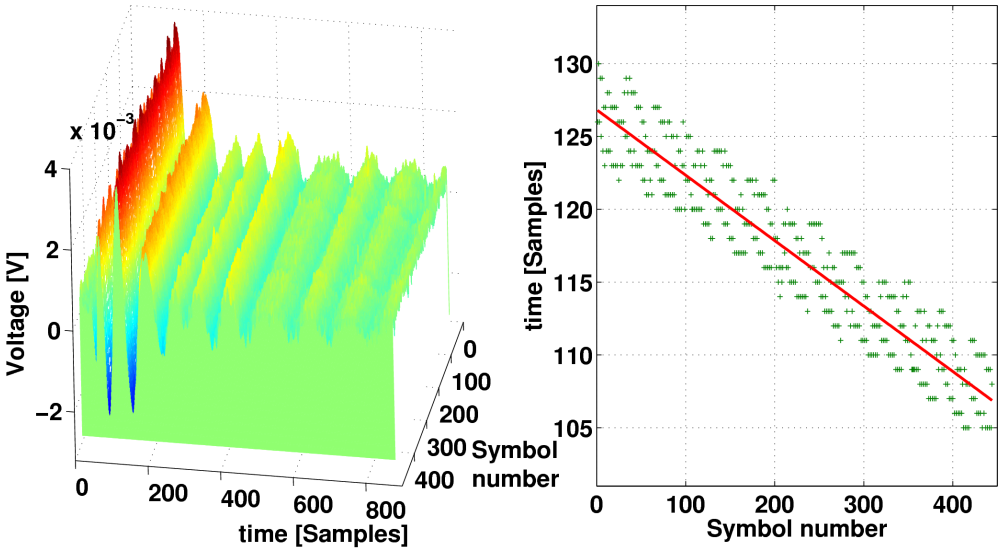


Figure 3.41: Many subsequent measured impulse responses (symbols) are depicted on the left side. The measurement is done for the Virtex-II Board, where the measured drift is too high. By plotting the maximum of each symbol, the drift can been seen on the right side. Each cross corresponds to a maximum, and the straight line depicts the average drift [Bau08].

indicators were calculated. One is the *signal to noise ratio* (SNR) of the signal. Because we make a threshold decision, SNR values under 4 dB are difficult to decode. We also calculate the SNR from the decoded sequence, so bit errors falsified our SNR. In these cases, the real SNR is lower than the calculated SNR. The second indicator, called *bit gain*, is the difference from the mean level of the bits and the threshold level. This indicator shows how big the difference of the voltage swing between ones and zeros of the signature is. Also, the *root mean square* (RMS) from the recorded signal without the DC part is measured. Figure 3.42 shows a signal of good quality before the bit decisions with a SNR value of 37 dB. The signal shown in Figure 3.43 is of lower quality and has a SNR of 9 dB.

First, the basic method described in Section 3.4.3 is evaluated (see Table 3.13). We decoded the signature with both boards and the *Des56* core where only 16 lookup tables are transformed into SRL16. We have evaluated two cases, one where only the watermarked core is implemented (case *A*) and one where the watermarked core and the original core is implemented to check the functionality of the watermarked core (case *B*). This is done by connecting both cores to the same pseudo random input data and compare the output when the cores are not in the reset state. We embedded three signatures (s_1, s_2 , s_3) in the core. The Signature s_1 is “5C918CBA” and

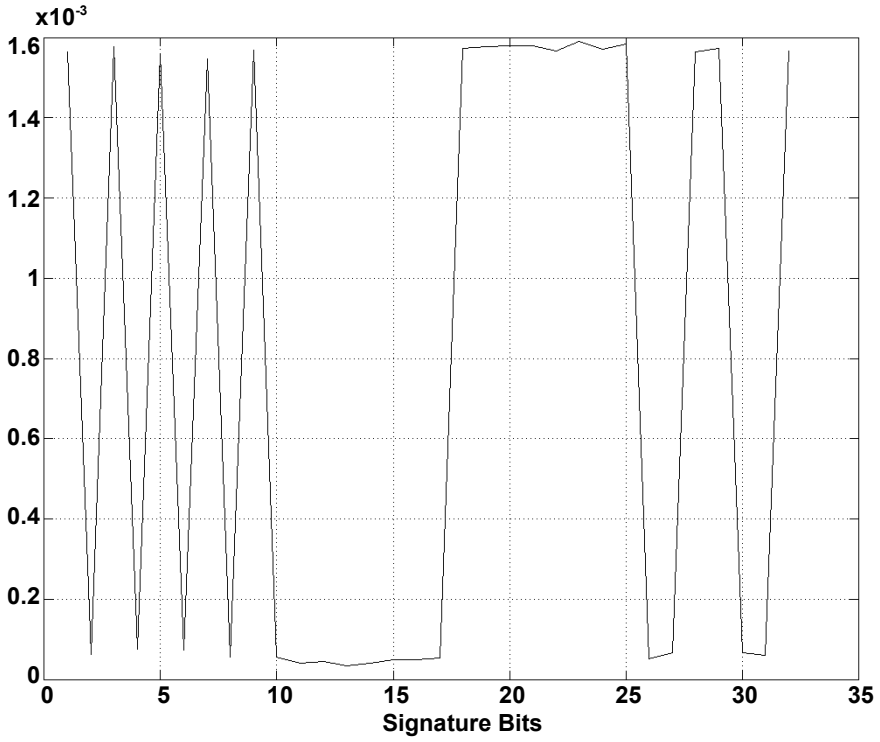


Figure 3.42: A signal of good quality for the bit decisions with a SNR value of 37 *dB*.

represents a realistic random signature. Signature s_2 is "333333333" and signature s_3 is "FF335500". With these signatures, we can evaluate the decoding method with different bit toggle rates.

Table 3.13 shows that decoding does not always work without bit errors. In case of *A*, the detection works better than in case *B*. In case *B*, more logic is used, but this logic is in the reset state. Nevertheless, the clock tree is still active which can be seen in the higher signal RMS value. The signature s_3 is difficult to decode, because there are many equal bits lumped together and so the printed circuit board works as a resonator.

Enhanced Robustness Encoding

To evaluate the enhanced robustness approach as described in Section 3.4.4, we use the same test cases and implement only the signature s_3 which is harder to decode (see Table 3.14). Furthermore, we define two additional test cases. In *C*, the unwatermarked core has an inverted reset, so the core is working when the watermark is sending the signature. In *D*, two cores are working, while the signature is emitted.

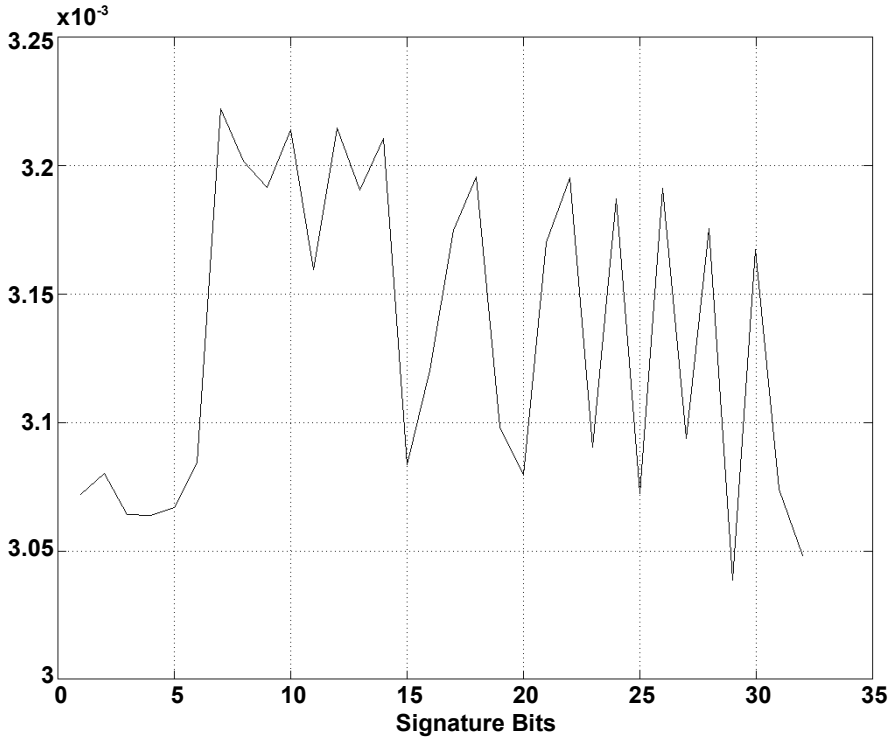


Figure 3.43: A signal of lower quality with a SNR of 9 *dB*, but without bit errors.

Not all combinations in *D* are possible because the FPGA is too small to implement all three cores. Additionally, we have evaluated this method with the *arithmetic coder core*.

Table 3.14 shows that the detection of the watermarked signature works much better than using the basic method. The decoding for the Des56 core works fine. Even if one or two of the same *Des56* cores operate at the same time, the signature is emitted and detected correctly. The arithmetic coder core requires more lookup tables than the *Des56*, and if no other core operates, the decoding results are better than for the *Des56* core. However, if another arithmetic coder core is active, the decoding fails. The signal RMS indicates that the arithmetic coder core has a very high toggle rate.

In Table 3.15, we decreased the recording length to see the impact of the quality of our results. This is done using the *Des56* core in all four cases. The quality degenerates but with the recording length of 50 μs, it is still possible to detect the watermark without bit errors in case *D*, even if two other cores are simultaneously active.

Case	Board	Bit Error Rate in %	Signal RMS in mV	SNR in dB	Bit Gain
Signature s_1					
A	Spartan-3	0	0.376	8.5	0.126
B	Spartan-3	9.4	0.511	4	0.112
A	Virtex-II	21.9	0.821	4	0.277
B	Virtex-II	31.2	1.047	4	0.263
Signature s_2					
A	Spartan-3	0	0.374	8.5	0.147
B	Spartan-3	3.1	0.513	4.5	0.137
A	Virtex-II	6.2	0.859	4	0.561
B	Virtex-II	0	1.063	8.5	0.632
Signature s_3					
A	Spartan-3	6.2	0.380	4	0.111
B	Spartan-3	12.5	0.516	3	0.122
A	Virtex-II	na	0.841	3.5	0.368
B	Virtex-II	9.4	1.073	3.5	0.381

Table 3.13: The decoding results of the basic method for different signatures and boards. The bit error rate and the signal RMS as well as the decoding quality indicators SNR and bit gain are shown.

BPSK Detection Method

Next, the *BPSK* detection algorithm as described in Section 3.4.5 is evaluated using the same test cases as for the enhanced robustness method (see Table 3.16). The results show that error-free decoding is possible in all test cases, also in the critical test case for the arithmetic coder *C* with the Spartan-3 FPGA, where a correct decoding is not possible using the enhanced robustness method. This shows that the *BPSK* method can deal better with cases which have high interferences on the clock frequency like other working cores with a high toggle rate.

Correlation Detection Method

The experimental results for the correlation detection method are shown in Table 3.17 with the same cases as in the tables before. This methods uses an intersymbol interference-free encoding. Therefore, the experimental results do not depend on the signature like the results of the other methods. The decoding was done with the correlation of the highest frequency component of the corresponding impulse response. On both boards, this is the component $h_3(t)$ (see Table 3.12). To enhance the results, the second strongest frequency component, $h_2(t)$ on both boards, is additionally used.

Case	Board	Bit Error Rate in %	Signal RMS in mV	SNR in dB	Bit Gain
Des56 Core					
A	Spartan-3	0	0.384	22	0.087
B	Spartan-3	0	0.508	23	0.110
C	Spartan-3	0	1.21	22	0.109
D	Spartan-3	0	2.15	10.5	0.0539
A	Virtex-II	0	0.794	18	0.067
B	Virtex-II	0	1.022	22.5	0.191
C	Virtex-II	0	2.698	12	0.067
Arithmetic Coder Core					
A	Spartan-3	0	0.618	37	0.758
B	Spartan-3	0	0.617	38	0.720
C	Spartan-3	na	4.488	3	0.216
A	Virtex-II	0	1.347	37.5	1.248
B	Virtex-II	0	1.343	37	1.191

Table 3.14: Results of the enhanced robustness encoding scheme from Section 3.4.4 for different cores and boards.

	200 μs		100 μs		50 μs	
Case	SNR in dB	Bit Gain	SNR in dB	Bit Gain	SNR in dB	Bit Gain
A	22	0.087	21.5	0.091	16	0.090
B	23	0.110	19.5	0.110	16.5	0.111
C	22	0.109	18	0.107	18.5	0.107
D	10.5	0.054	10	0.057	9.5	0.061

Table 3.15: Results with decreased recording time obtained from the Spartan-3 board for the *Des56* core. The signature decoding has no bit errors in all cases.

Case	Board	Bit Error Rate in %	Signal RMS in mV	SNR in dB	Bit Gain
Des56 Core					
A	Spartan-3	0	0.431	22	0.091
B	Spartan-3	0	0.530	22.5	0.086
C	Spartan-3	0	1.410	25.5	0.093
D	Spartan-3	0	1.432	20	0.044
A	Virtex-II	0	1.003	19	0.152
B	Virtex-II	0	1.353	19	0.073
C	Virtex-II	0	3.030	23	0.178
Arithmetic Coder Core					
A	Spartan-3	0	0.593	23.5	0.322
B	Spartan-3	0	0.703	29.5	0.438
C	Spartan-3	0	4.207	14	0.188
A	Virtex-II	0	0.340	27	0.654
B	Virtex-II	0	0.510	19	0.239

Table 3.16: Results using the BPSK method with different cores and boards. The decoding is successful without bit errors in all cases.

Case	Board	Bit Error Rate in %	Signal RMS in mV	SNR $h_3(t)$ in dB	SNR $h_2(t)$ in dB
Des56 Core					
Signature s_1, Decoding with one pattern					
A	Spartan-3	0	-	3.6	-9.6
B	Spartan-3	18.5	0.698	1.9	-
C	Spartan-3	15.3	1.13	1.6	-
B	Virtex-II	6.2	2.44	3.4	2.5
C	Virtex-II	15.3	3.28	2.8	1.9
Signature s_1, Decoding with four pattern					
B	Spartan-3	3.1	0.648	2.0	-
C	Spartan-3	6.2	0.987	1.6	-
B	Virtex-II	3.1	2.18	3.6	2.7
C	Virtex-II	3.1	2.93	2.9	2.2

Table 3.17: The decoding results of the correlation detection method obtained for the *Des56* core. For decoding, the frequency component $h_3(t)$ of the impulse response and in some cases additionally $h_2(t)$ are used.

The SNR values in Table 3.17 are small when compared to the other methods. One reason for this is that only the used frequency component counts to the signal and all other frequency components of the impulse response are calculated to the noise. Furthermore, the above values of Table 3.17 are acquired by decoding only one signature pattern, whereas the decoding for the other methods used the average of many repeated signatures to lower the noise level. The values at the end of Table 3.17 correspond to a decoding which uses the average of four repeated signature patterns.

Nevertheless, the BPSK and the enhanced robustness encoding method exceeds this method in terms of the bit error rate for these boards. However, this method is signature- and board-independent and can be further enhanced, e.g., for multiplex methods where multiple cores can concurrently send signatures.

Multiplexing Approaches

For the problem of concurrently sending of different signatures, we investigate *time division multiplexing* (TDM) and *code division multiplexing* (CDM) using *optical orthogonal codes* as introduced in Section 3.4.7. To obtain experimental results, we applied the correlation detection method. However, the other approaches might also be adapted for concurrently sending of different signatures. We use the *Des56*, the *3DES*, the *keyboard controller*, and the *i2c* core from *opencores.org* [Opec] for the following experiments.

To evaluate TDM, we choose a different signature s_i and period time ϕ_i for each core and embed the signatures with the corresponding sending logic into the cores. The decoding results in Table 3.18 show that the bit error rate is similar to the correlation method without multiplexing. It is further possible to use other encoding methods to enhance the detection. Unfortunately, the best decoding technique, BPSK, is not suitable due to the long signature length, which results in higher and fewer possible periods for sending different signatures.

Finally, the results for optical orthogonal code multiplexing are shown in Table 3.19. The cores and signatures are the same as in the evaluation of TDM. For this experiment, we use 48-bit chip sequences. This means that for transmitting one bit of the signature, we need 48 times longer than in the original correlation detection method. Unfortunately, in these chip sequence family, only 8 *optical orthogonal sequences* exist, which means that we are only able to watermark 8 different cores. To get more optical orthogonal sequences, the chip must be extended by 6 digits for each additional sequence. This might be a problem if every watermarked core should get its own chip sequence to ensure correct decoding for all combinations of different watermarked cores. Nevertheless, the results show that it is possible to decode simultaneously sent signatures with this method. Note that the results are obtained from decoding only one pattern of each signature. By averaging many patterns to lower the noise, the results can be further improved.

Core	Number of used SRLs	Signature s_i	period time ϕ_i	Bit Error Rate in % Dec. one pattern	Dec. four pattern
Des56	40	0x153CA9F8	460 μs	9.4	3.1
3DES	64	0x128E92C1	500 μs	9.4	3.1
keyboard contr.	18	0x928CB241	420 μs	18.3	6.2
i2c	25	0x74DE4FC1	380 μs	21.3	6.2

Table 3.18: The decoding results for one pattern and for the average of four pattern of the TDM method. All four cores are implemented in one design and sending the corresponding signatures. Furthermore, the number of used shift registered, converted from functional lookup tables are shown.

Core	Number of used SRLs	Chip Sequence	Bit Error Rate in % $h_3(t)$	$h_3(t)+h_2(t)$
Des56	40	0xC00000002000	21.9	15.6
3DES	64	0xA00000400000	15.6	12.5
keyboard contr.	18	0x900010000000	6.25	6.25
i2c	25	0x880400000000	18.75	18.75

Table 3.19: The results from applying the code division method using optical orthogonal codes by sending all signatures simultaneously. The bit error rates are achieved from decoding only one signature pattern for each core.

3.6 Summary

In this chapter, we have presented different approaches for watermarking and identification of IP cores. Our methods follow the strategy of an easy verification of the watermark or the identification of the core in a bought product from an accused company without any further information. Netlist and HDL cores, which have a high trade potential for embedded systems developers, are in the focus of our analysis. We concentrated on FPGA technology which represents a dynamic and increasing market. Nevertheless can the methods, especially the power watermarking techniques, also be adapted to ASIC designs.

To establish the authorship in a bought product by watermarking or core identification, we have discovered different new techniques, how information can be transmitted from the embedded core to the outer world. In this work, we concentrated on methods using the *FPGA bitfile* which can be extracted from the product and on methods where the signature is transmitted over the *power pins* of the FPGA.

In Section 3.2, we adapt the *theoretical general watermark approach* from Li et al. [LMS06] for IP core watermarking and identification and show possible threats and attacks. Section 3.3 deals with IP core watermarking and identification methods where the authorship is established by analysis of the extracted bitfile. The extraction of lookup table contents of a binary bitfile was demonstrated for Xilinx Virtex-II and Virtex-II Pro devices.

Subsequently, we have presented a new method to identify IP cores in FPGA bitfiles. Possible transformations of the mapping tools and the effect of the robustness of the method were discussed. The experimental results show that it is possible to identify a core inside a design with high probability. The identification process is based on two parameters, namely the number of found lookup tables of the core in the design and the mean distance to the core center. However, it must be taken into account that lookup tables of the core may be removed by optimization tools, if a part of the core is not used because the outputs are unconnected or constant values apply to inputs.

Furthermore, methods towards an identification of HDL cores are introduced. The first step is to identify two netlist cores to determine if they were generated from the same HDL source. Experimental results are promising. However, for a complete approach for HDL core identification in a product, there are some gaps which might be stuffed by future research.

We introduced watermarking methods for bitfile and netlist cores which use a bitfile verification strategy. For bitfile cores, unused lookup tables in used slices carry the watermark and are easy to extract. For netlist cores, the watermark is inserted into functional lookup table which have less inputs as the corresponding primitive cell by restricting the reachable address space. By using functional lookup tables, the watermarks are tightly integrated into the core and prevent the watermarked lookup tables from being removed by either an attacker or synthesis and optimization tools.

Experimental results show that the watermarks are correctly extractable with high probability and cause only low overhead.

In Section 3.4, we have presented new watermark techniques for IP cores where the signature can be extracted easily over the power pins of the chip. The main idea is that during a reset phase of a chip, a watermark circuit is responsible to emit a characteristic power pattern sequence that may be measured by voltage fluctuations on power pins. With these techniques, it is possible to decide with high confidence, whether an IP core of a certain vendor is present on th FPGA or not. We have shown how a watermark can be integrated into a core. For Xilinx FPGAs, it is possible to integrate the watermark algorithm and the signature into the functionality of the core, so it is hard to remove the watermark, and only very few additional resources are required for control. We have investigated also the communication channel from the power pattern generator inside the core to an oscilloscope which is used to measure the voltage trace outside the FPGA. A *basic algorithm* was introduced to detect a signature over the voltage trace of the FPGA, and experimental results have shown that the functionality of the core is not altered. Also, we introduced an *enhanced robustness technique*, and a new decoding principle based on *BPSK* (Binary Phase Shift Keying) modulation which may improve the decoding quality of the signature even further. Finally, *correlation detection* was presented which uses special communication channel characteristics to avoid *intersymbol interference*. With these enhanced decoding methods, we are able to decode a signature even if other cores are simultaneously active on the same hardware device and emitting watermark specific power pattern simultaneously. We also introduced quality indicators to evaluate the result of the decoded signature and prove the robustness of the techniques.

The experimental results have shown that decoding is possible in all test cases, but it is possible to further improve the quality of the results if more lookup tables are transformed into shift registers or if the recording time is extended. Additionally, the signature width might be increased to insert error codes or cyclic redundancy check (CRC) values.

If an FPGA design includes multiple watermarked cores, mutual interference between different signatures may occur, lowering the detection probability. We investigated different *multiplexing methods* for scenarios where multiple unsynchronized cores may simultaneously send their signatures. Here, *asynchronous time division multiplexing* (TDM) which uses different sending periods and *code division multiplexing* (CDM) which uses *optical orthogonal codes* are applicable.

In this chapter, we have presented many novel watermarking methods for different kinds of cores. Their practical applicability was proven by experiments. For all methods, we analyzed the strengths and weaknesses in case of removal of ambiguity attacks. Nevertheless, this is a very interesting topic and research is far away from being complete or finished.

4 Control Flow Checking

This chapter presents new techniques for control flow checking for embedded CPUs and general IP cores. First, an introduction is given. The next section deals with fault injection which is necessary for the verification of the proposed methods and implementations. In Section 4.3 two novel methods for control flow checking are explained, and Section 4.4 provides hardware architectures for the implementation of the corresponding methods. A concrete example implementation with results on memory and logic overhead for an FPGA target is presented in Section 4.5. Furthermore, a case study is provided and finally, the chapter is summarized.

4.1 Introduction and Scope

Robustness, reliability and security are essential requirements of today's systems-on-a-chip. Modules and their integration in a system have to be designed to be still operational also in difficult and interference-prone situations as well as insecure environments.

In this chapter, the goal is to investigate methods to detect, analyze, and correct transient and/or permanent errors (see Chapter 1) occurring in the control paths of embedded RISC-CPUs or IP cores. The basic idea thereby is to define autonomously behaving components that may resolve functional errors locally inside the core at runtime (during the program execution or operation) and, i.e., prevent false instructions to be executed. This means that no faulty branch or jump instruction shall be executed that would lead to a vulnerable or wrong program state (error-resilience).

4.1.1 AIS Project Overview

Most of the methods presented in this chapter were developed during the AIS (Autonomous Integrated Systems) project funded by the BMBF (German: Bundesministerium für Bildung und Forschung, Federal Ministry of Education and Research) and the edaCentrum e.V.. The project researches a new approach for reliable MPSoCs (*Multiprocessor-System-on-Chip*) which deals with underlying unreliable components. The approach proposes a new architecture which consists of three logical layers (see Figure 4.1) [SBE+07b, SBE+07a, SBH+09].

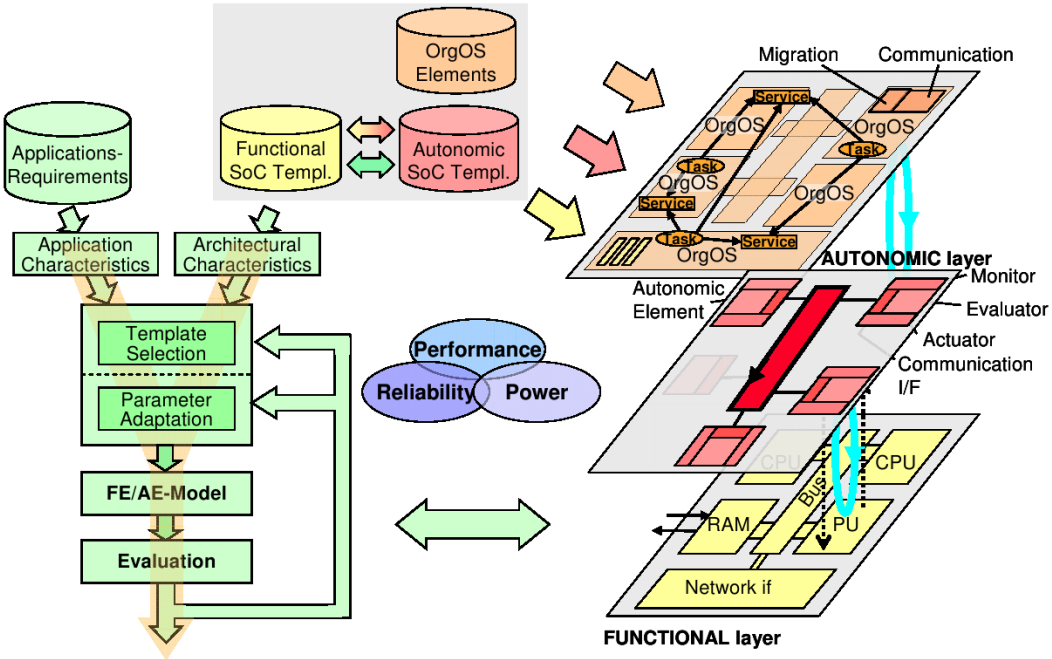


Figure 4.1: On the right side, three behavior levels and the connections between them are shown. On the left side, the flow of modeling and system exploration for autonomic components is depicted [SBE+07b, SBE+07a].

The Three Layer Model

The well known functional layer consists of usual functional elements, like CPUs, special purpose cores, and memories, as well as communication structures, for example buses.

On the autonomous layer, autonomous elements reside which are able to communicate to each other over an interconnection structure. The autonomous elements shall provide monitoring, error detection, and avoidance of failures for the underlying

functional elements. They consist of sensors, evaluators, and actuators which feature the element to detect and analyze faults, errors, or disturbances of the corresponding functional element. In case of an error, they trigger adequate reactions. The combination of functional and autonomous elements transform the functional elements into reliability-improved autonomous units. The functional and autonomous layers are only logical layers; both are implemented on the same die.

Finally, the third layer is the autonomic operating system layer which consists of an operating system, running on the different CPUs, which are able to evaluate the information from the autonomous elements, and, if necessary, may migrate tasks.

4.1.2 AIS Work Packages Overview

To reach these goals AIS organized its research with two *work packages* (WP). A new kind of system design methodology for autonomous integrated systems is explored in the first work package. The second work package introduces the design of components to fulfill the previously named requirements on the component level. With this new component design methodology and components of architecture will be dimensioned with autonomous characteristics and provided for system design. In a process of exploration and integration these components will be combined with an autonomous behavior-based operating-system environment at system level. Beside the hardware design, the new design process will include the operating system level of the MPSoC.

System Level Techniques (WP1)

The exploration of system level techniques for increased error resiliency consists of new design methods, an autonomous operating system, and performance analysis. The research work on system level design methods consists of a transaction-level-based system model which can be explored for the well known optimization goals *performance, area* and *power dissipation* as well as the introduced new optimization goal *reliability*. The focus lies on modeling of degeneration faults which are accelerated by a huge number of temperature cycles caused by aggressive power management [SSB09].

The distributed autonomous operating system efficiently exploits the restricted resources of the MPSoC and provides communication and migration services for each CPU. To achieve fault tolerance, the operating system generates replicas of services, which come active in the case of a node failure. Also, reliability information from the autonomous elements is analyzed and taken into account by the distribution of services. If a node becomes unreliable, the services will be migrated to other nodes [BSR09].

If error correction measures are active, they often have impact on the timing and the performance of the system. The result may be system failures, if deadlines are

missed and buffers may overflow or underflow. With a sophisticated fault model, the impact of fault and error correction measures can be simulated and countermeasures like traffic shaping or a better scheduling can be adapted [SE09].

Component Level Techniques (WP2)

The component level techniques work package of the AIS project analyzes and enhances exemplarily three common functional components of an MPSoC in order to transform them into autonomous units. The three components are *CPU data path*, *CPU control path*, and *on-chip communication resources*.

The data path of a modern RISC CPU consists of different pipeline stages. To detect soft errors like single event upsets and transients, a shadow register approach may be used [ABMF04]. The shadow registers are clocked with a delayed clock signal whose delay is usually the duration of a soft error effect. To correct a detected error, history registers are used. Using these registers, the system can perform a rollback of one clock cycle to correct the error [BZS+06] (see also Section 4.5.5).

The reliability of the communication resources between different components is as important as the reliability of the components itself. The necessary degree of reliability depends highly on the type and amount of transferred data. For example, audio or video streams require a lower level of error correction than program data, where a single bit error can cause a system failure. A solution is that for different categories of transferred data, different error detection and correction techniques can be used. Furthermore, error correction measures on a higher level of the protocol stack can be used. At the data link level, e.g., self-calibration and coding techniques may be combined to dynamically change the coding strength at runtime [MWB+10].

The methods and techniques discussed in this chapter and elaborated during the project as part of this dissertation consider the development of concepts for an *autonomous CPU control path* that is able to guarantee the correct execution of the control flow of a given CPU. The main task of the control logic in a CPU is to control the program flow. The actual state of execution of a program is in general given by the state of the program counter and the CPU registers. Usually, the next instruction to fetch is given by an increment of the program counter, but also branches and jumps may occur. Now, it is possible to extract information about the program flow from program executable at compile-time and use this information to check the control flow of the processor with a so called *control flow checker* unit at runtime (see Figure 4.2). If an error is detected, the autonomous control flow checker should be able to initiate a re-execution of the control flow instruction immediately so to prevent a security bottleneck or guarantee reliable execution of the correct control flow. A quite general definition of the control flow checking may be given as follows:

Definition 4.1 *Control flow checking denotes the task of testing whether a sequence of program counter values is correct with respect to a given program specification.*

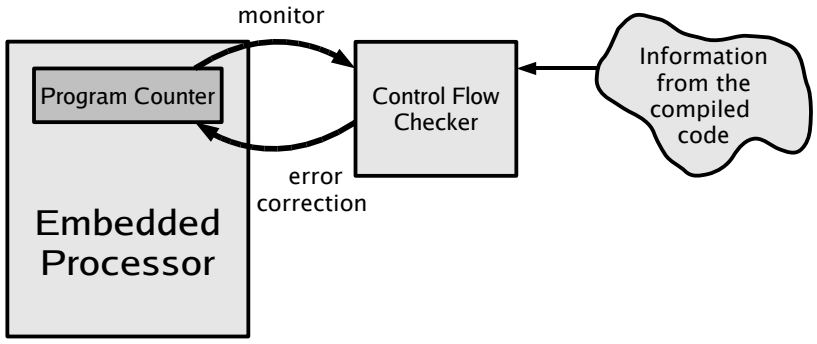


Figure 4.2: Concepts for autonomously interacting control flow checking that can monitor the program counter of a given processor and detect false jumps or branches based on information in the compiled code. Moreover, the checker should also be able to correct false jumps and branches.

The main contribution in the context of control flow checking will be concepts, methods, and implementations of novel techniques for autonomous control flow checking with the following improvements over the state-of-the-art: Low overhead, detection of all errors which affect the control flow, and modularity.

Prior to presenting results, we discuss the problem of fault injection.

4.2 Fault Injection

This section discusses fault injection which is necessary for the simulation and verification of control flow checking methods. In order to evaluate error detection and correction methods, we need to know the failure rate of the system. We can detect a failure, if the output of the system is different from the specification (for example, a wrong, unintended or neglected `printf()` output).

Usually, the *MTTF* (mean time to failure) is very high in embedded systems, so we must think of fault injection techniques to stimulate faults. We can inject faults naturally or through a fault model.

Natural faults can be injected in a radiation chamber, by applying extreme temperatures, or through a high excessive clock frequency. The advantage to inject faults naturally is that we must not model theses faults. The disadvantage is that normally not all types of fault can be covered and the equipment can be very expensive. Also, if we use FPGAs to demonstrate our methods, the FPGA configuration may be affected instead of the logic implemented in the FPGA.

The other option is to use a fault model to stimulate faults. These fault models can either be used in RTL system simulation, or be implemented as a fault injection

component on the chip. Fault injection methods which are using a fault model can be categorized into two classes: *Intentional fault injection* and *random fault injection*.

4.2.1 Intentional Fault Injection

Using intentional fault injection, only faults which lead to an error are injected to be detected by the error detection method. To test a new error detection method, this is usually the first step. If the error detection method works correct, all errors which are caused by the faults may be detected and, in the case of an error correction method, also corrected. In the following, some examples of intentional fault injection for common embedded systems are given:

Permanent Memory Errors

Permanent memory errors are the result of permanent or transient memory faults, for example, degeneration, manufacturing faults, SET or SEU inside the memory. To inject permanent errors into the memory in embedded systems, an *in-circuit debugger* can be used. For example, the in-circuit debugger of the Leon3 [Gaib] system has unlimited access to the system memory. The program code is written into the system memory only once at startup. If the code is manipulated inside the memory, permanent memory errors can be simulated. For testing control flow checking methods, it is also easy to manipulate some bits in the control flow instruction in order to the program counter jumps to an incorrect destination. For simulating permanent memory faults, it is also possible to manipulate the executable file (the program binary).

At RTL simulation, the content of the system memory is often stored in a file. At this level, it is also very easy to manipulate instructions.

Transient and Permanent On-Chip Errors

Transient and permanent errors in logic and registers inside IP cores can be injected by altering values of signals. This can be done, for example, for permanent errors (faults) by clamping the signal to '0' or '1' (stuck at fault). For transient errors, signals can be XORed with an error signal (see Figure 4.3). If the error signal is high, the value of the original signal is inverted. Other possibilities are to take an AND or OR gate instead of the XOR gate to tie a signal up or down. So, SETs and SEUs can be simulated.

The error signal can be read from a file in the case of RTL simulation, or can be generated by a *PRNG* (pseudo random number generator) which can be, for example, an *LFSR* (linear feedback shift register) or in the trivial case, a counter. Using a counter, the error signal is set high if the counter has reached a specified value. These techniques belong also to the class of intentional fault injection, because the time and

place were the fault appears is intended. The advantage of the PRNG method is that it is possible to run an RTL simulation and also implement it on the chip.

Transient On-Chip Faults

To model a SET in RTL simulation, the affected signal is set to high at a specific time and after the usually time period which a SET affected the signal (usually far fewer than a clock cycle), the signal is reset to the original value. With this technique, the effects of a SET can be simulated. The trigger of the injection can also be achieved by a PRNG. Some RTL simulators support the alteration of signals without changing the VHDL model (for example the `force` command in Modelsim). This technique can also be used to inject faults.

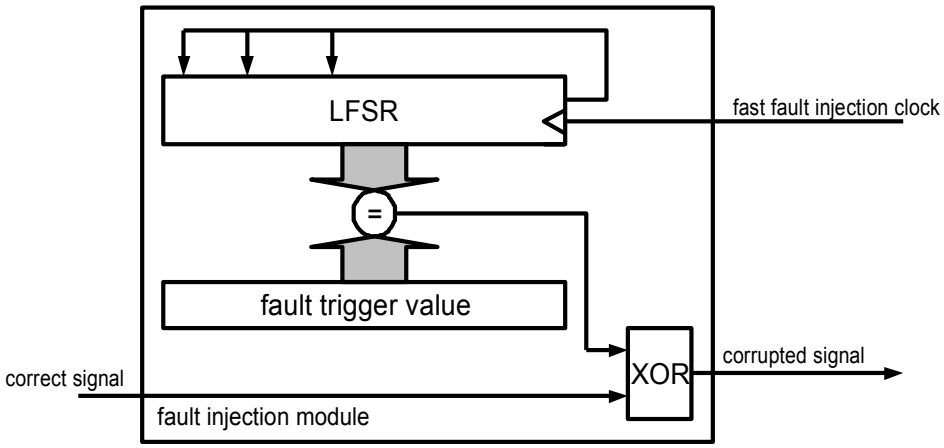


Figure 4.3: A fault injection module. An LFSR is clocked by a fault injection clock from outside the module or internally in case of an RTL simulation. If the LFSR reaches a certain state, a fault is triggered. To inject faults, a signal or net is split, and the fault injection module is inserted.

4.2.2 Random Fault Injection

To measure the reliability of a whole system, it is necessary to evaluate the *MTTF*. This can be done by using *random fault injection*. Here, the time and the place (the signal) where the fault is injected is randomly distributed. To compare the reliability with and without error correction methods, two measurements are necessary, one using the error correction module and one without. To get meaningful results, the faults should be injected in both measurements at the same place and time, so the

same fault injection configuration should be used. The measurement can be done through simulations or in hardware on an FPGA platform.

To inject faults at random, we can use the following strategy: First, the core or the system is synthesized and an EDIF netlist is written. A tool chooses signals (or nets for EDIF netlists) randomly, and inserts fault injection modules (see Figure 4.3) as black boxes. Afterwards, for simulation, the altered EDIF netlist is converted back to an HDL simulation model. To inject faults on the hardware demonstrator, this step is not necessary.

Inside the fault injection module, the original signal is combined with an error signal using an XOR, AND or OR gate (same technique as in 4.2.1). The error signal is generated either from a file reading module or by a PNRG. To inject a fault in an FPGA demonstrator, only the PNRG method can be used. To inject faults at random times, the PNRG can be initialized with different random values. To inject faults with a PNRG at a higher time resolution than the clock frequency, a faster clock can be used for the PNRG. If the PNRG reaches a specified value, a fault injection is triggered. The probability P_f that a fault is injected in a PNRG clock cycle is

$$P_f = \frac{1}{2^{n_{pnrg}}}, \tag{4.1}$$

where n_{pnrg} is the bit width of the PNRG and the trigger value. With the bit width n_{pnrg} and the number of fault injection modules, different fault probabilities can be simulated.

4.3 Methods for Control Flow Checking

In this section, new methods for control flow checking in embedded processors and IP cores are presented. First, a classification of control flow instructions in embedded RISC processors is given. Subsequently, two different methodological concepts for control flow checking of direct jumps and branches are introduced and possibilities to check indirect jumps are discussed. Then, methods for repairing a corrupted program path are proposed. Finally, methods for checking *Finite State Machines* (FSMs) in general IP cores are analyzed.

4.3.1 Branches and Jumps

Control flow instructions (CFI) can be categorized into conditional branches and unconditional jumps. Conditional branches depend on the result of a logical or an arithmetic operation. On most processor architectures, the arithmetic operation affects a register, called *integer condition codes* (icc). This register consists of flags which describe properties of the result, for example whether the result is greater than zero, or negative. Conditional branches evaluate this register for the decision to take or not

take the branch. The way of evaluation (e.g., branch if the zero flag is set) is statically coded in the instruction itself, whereas the evaluation of the condition is performed at runtime.

Both groups of control flow instructions can be further subdivided into *direct* (static) and *indirect* (dynamic) jumps or branches. The destination of direct branches or jumps is fixed at compile-time and is encoded into the jump or branch instruction in an absolute or relative address. For indirect jumps or branches, the destination address is determined during program execution. The destination address is given in absolute or relative manner by either a register value or as the result of an operation with registers or the result of an operation with a register and a constant value which is encoded into the instruction.

In summary, four types of control flow instructions exist:

- *(Unconditional) direct jumps* (e.g., `call, goto`),
- *(Conditional) direct branches* (e.g., `if .. then .. else`),
- *(Unconditional) indirect jumps* (e.g., `return` from subroutine), and
- *(Conditional) indirect branches*[1].

Furthermore, the class of unconditional indirect jumps can be subdivided into *returns from subroutine*, *register indirect calls* and *other jumps*. A return from subroutine is an example of an indirect jump, because the program counter jumps to the address where the routine is called from, and this address is only known at runtime. Register indirect calls are calls where the address of the called subroutine is determined at runtime. This usually happen in C++ if a *virtual function* is called.

Finally, jumps which are not triggered by an instruction can occur such as *interrupts* and *traps*. The destinations of interrupts are typically given by the start address of the main interrupt service routine, and so, interrupts belong to the class of direct jumps. Traps occur on exception conditions (like divide by zero). Here, the program redirects to the address of an exception handler, and so, traps can be treated as direct jumps, too.

Table 4.1 presents an analysis of the quantity of these different types of branches and jumps in the code on the *SPARC V8* [SPA] architecture for the *SPEC CINT2000 benchmark* [SPE] for a given list of programs. As can be seen, indirect calls and jumps occur relatively rarely as opposed to direct branches and jumps.

4.3.2 Methods for Checking Direct Jumps/Branches

In SoCs, a CPU often executes only a few specified programs over its lifetime. This holds true particularly for embedded applications where the system is often only

[1] Note that conditional indirect branches are not supported by any instruction set architecture that we know of.

SPEC program	all instructions	direct		indirect		
		branches	jumps	returns	calls	other jumps
gzip	19979	1426	599	111	4	0
gcc	566280	54791	22446	2236	140	273
vpr	51771	2764	2012	269	2	7
mcf	3881	288	82	26	0	0
crafty	82891	4814	4074	108	0	13
parser	36862	3189	1701	320	0	2
gap	236181	18733	4158	828	1262	5
vortex	174567	12537	8491	913	15	21
bzip2	12162	748	380	73	0	0
twolf	102899	5701	2060	189	0	2

Table 4.1: Accumulated number of all and different kinds of control flow instructions of benchmarks of the SPEC CINT2000 test suite [SPE] when compiled to the SPARC V8 [SPA] architecture.

programmed once, and the code is never changed during the lifetime of the product, except for the update of the SoC with a new firmware and software. Furthermore, it is well known that in many computational intensive problems, most of the execution time is spent in only few subroutines. So, it is beneficial to analyze these subroutines for branches and jumps statically.

If we assume that only direct jumps and branches exist in a given code segment, we will show that we are able to verify the control flow of this code and guarantee the correct execution of each direct control flow instruction as well as the (successively) linear execution of all the other instructions (the program counter value is incremented by one word address after each instruction). To verify the correct execution of control flow instructions, we need to check whether the address of the control flow instruction and the target address are correct. The program counter value before and after the execution of a control flow instruction can be compared to these addresses. If there is a mismatch, an error signal is raised. To check a non control flow instruction, the program counter before and after the execution of the instruction can be compared. If the second one is not an increment of the first one, the error signal is also raised.

In the following, we propose two alternative methods to obtain the correct addresses of control flow instructions of a given machine program and the corresponding targets. The first method is called *basic block* or *control flow method* (CF). The second method is called *control flow instruction method* (CFI).

Control Flow Method

First, a given compiled machine code is separated into a set of *basic blocks* (*BB*). A basic block is a sequence of code which is executed successively without any jumps or branches except, possibly, at the end. The basic block can only be left at the end of a block and can only be entered at the beginning. Only the last instruction can be a jump or branch and only the first instruction can be a jump or branch destination. The following instructions define the beginning of a basic block [TH07]:

- the first instruction in a program or segment,
- the instruction following a control flow instruction,
- the instruction which is a destination of a control flow instruction.

From this information, the control flow graph $CFG(BB,T)$ is built: Each node $BB_i \in BB$ of the control flow graph represents a basic block. The nodes are sorted with increasing start address of the corresponding basic block in ascending order. Each edge $t_j \in T$ represents a transition of the control flow from one basic block to another. If the last instruction of a basic block BB_i is a direct branch instruction, the basic block has two successors. One is the basic block next in the list BB_{i+1} (if the branch is not taken), and to a basic block where the first instruction is the branch destination (if the branch is taken). Jumps have only one successor, and if the last instruction is not a control flow instruction, the successor basic block is always the next basic block BB_{i+1}. An example program and the corresponding CFG are shown in Figure 4.4 which is separated into basic blocks.

With the given CFG, we have all information to check a sequence of program counter values for correctness leading to the specification of a proper control flow checker unit according to Figure 4.2 as follows: The information of the CFG can be either used to directly define a *finite state machine* (FSM) to check the correctness of a sequence of control flow instructions. Alternatively, an implementation using micro-instructions of a micro-programmed circuit can be deducted from the CFG.

For an implementation of the checker unit as a micro-programmed circuit, the information of the CFG can be stored inside memories. For each basic block, we need to store the start and the end address and also the indices of the successor basic blocks. The start address of each basic block is the end address of the previous basic block incremented by one. To minimize the memory overhead, we can store only the end address and a global start address. Also, we only need to store one successor of a basic block for branches because if the branch is not taken, the basic block with the next index (BB_{i+1}) is always executed.

With these memory overhead improvements, we need three memory items for each basic block inside the memory:

- One for the basic block end address ($addr$),

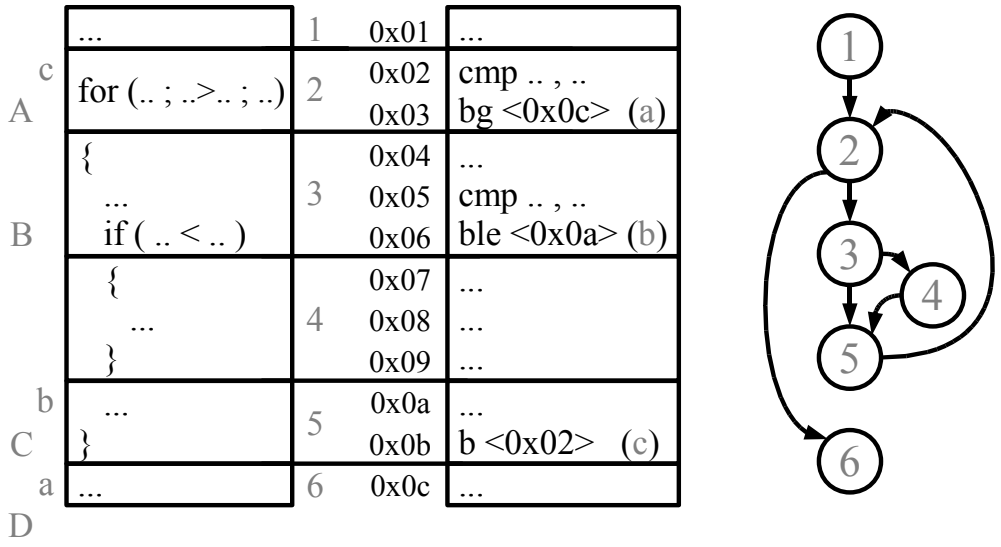


Figure 4.4: An example program code is given on the left hand side together with the corresponding assembler code. The CFIs are denoted A to C, and the CFI destination addresses a to c. D denotes the end of the program or segment to be checked. Furthermore, the code is divided into basic blocks BB_i, $i = 1,\ldots,6$. On the right hand side, the corresponding control flow graph (CFG) is shown.

- one for the index of the branch taken successor basic block (*suc*), and
- a flag ($flag$) which identifies the type of the last instruction of the basic block.

The flag is needed to choose the right transition to the next basic block (see Figure 4.5). Note that if the last instruction of a basic block is not a CFI, the successor basic block index (*suc*) is not needed.

The control flow checking algorithm is depicted in C language in Listing 4.1. For checking the control flow, we need the current program counter (*PC*) and the following program counter (*nPC*). The algorithm, implemented as a C function, returns 0 if the control flow for the program counter and its successor is correct, and -1 if the control flow differs from its specification. Further, the index i of the current basic block and the three memories (*addr*, *suc*, and $flag$) are needed. The function $addr(i)$ returns the entry with index i of the memory *addr*.

By looking up the basic block end address in the *addr* memory ($addr(i)$), we know when the basic block end is reached (Line 3). If the basic block end is not reached, the address of the next program counter must be the current address incremented by one (Line 23). If not, an error occurs. If the basic block end is reached, we must distinguish between the different types of the last instruction inside the basic block (Line 4, 9, and 17). If this is an unconditional jump, like a *call*, only the jump

index	*addr*	*suc*	*flag*
1	0x01	-	N
2	0x03	6	B
3	0x06	5	B
4	0x09	-	N
5	0x0b	2	J
6	0x0c	-	N

Figure 4.5: Three memory areas are necessary to store required information for each CFG. In the first column (*addr*), the address of the last instruction of the basic block is stored. The successor basic block for a taken branch is stored in the second column (*suc*). In the third column ($flag$), a flag is stored which identifies the type of the last instruction of a basic block. An N denotes a non control flow instruction, whereas a B denotes a branch. This example memory stores the values for the example program in Figure 4.4.

target must be checked for correctness. The corresponding target address is the start address of the successor basic block, given by its index. To get this address, the end address of the basic block with the previous index is fetched and the address is incremented ($BB_{i-1}+1$ or Line 5). Furthermore, the current index i must be updated to the successor basic block index (Line 6). If the last basic block instruction is a conditional *branch*, two possible successor basic blocks exist. If the branch is taken (Line 10), the handling is the same as for an unconditional jump. If the branch is not taken (Line 13), the next program counter value should be the current value incremented by one ($nPC == PC+1$). Hence, the next instruction belongs to the successive basic block and also the basic block index i must be updated (Line 14). Finally, if the last instruction of a basic block is not a CFI, the checking behavior is the same as on conditional branches, where the branch is not taken (Line 18).

One very similar approach of a CF method is described in [ARRJ06]. Here, the CFG is implemented in hardware by a finite state machine and a lookup table for resolving the control flow instruction addresses and indices (in memory). The disadvantage of this approach is that the checker unit must be synthesized new for each program. Here, in our memory-based approach, only the contents of the memories need to be reconfigured in order to check a new program.

Listing 4.1 Control flow (CF) checking algorithm

```
1  int check_cf( PC, nPC) {
2    static int i;                          // index i
3    if (addr(i) == PC) {                   // PC is BB end
4      if (flag(i) == 'J') {                // uncond jump
5        if ((addr(suc(i)-1)+1) == nPC) {   // correct?
6          i = suc(i);
7          return 0;
8        } else return -1;
9      } else if (flag(i) == 'B') {         // cond branch
10       if (((addr(suc(i)-1)+1) == nPC) { // branch taken
11           i = suc(i);
12           return 0;
13       } else if (PC +1 == nPC)) {        // branch not taken
14           i++;
15           return 0;
16       } else return -1;
17     } else {                             // non CFI
18       if (PC +1 == nPC) {                // correct?
19          i++;
20          return 0;
21       } else return -1;
22     }
23   } else {                               // non BB end
24     if (PC +1 == nPC) return 0;          // correct?
25     else return -1;
26   }
27 }
```

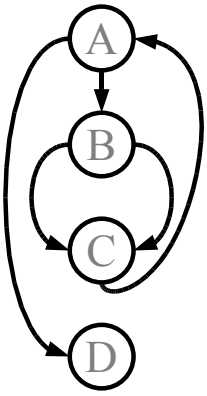
Control Flow Instruction Method

In contrast to the control flow (CF) method, the control flow instruction (CFI) method is based on storing control flow instructions instead of basic blocks. In case of direct branches and jumps, the start and target address are known at compile-time. So, it is possible to extract this information from the binary or the disassembled program code by decoding the instructions. The control flow instructions are then sorted by increasing addresses in ascending address order.

Then, the *control flow instruction graph* $CFIG(CFI,T)$ is built: Here, each control flow instruction in the code which should be checked represents a node ($CFI_i \in CFI$). Directed edges $t_j \in T$ of the CFIG denote transitions to the next following control flow instruction in the given code.

Like in a CFG, each node can have a maximum of two successors: two for a branch instruction and one in case of a jump instruction. For a branch instruction CFI_i, one successor is CFI_{i+1} (branch is not taken). The other successor of a direct branch and jump instruction is CFI_n which is the next control flow instruction in

the program code after the branch destination (branch is taken). The CFIG of the example program code form Figure 4.4 is shown on the left side of Figure 4.6. Note that *D* is not a CFI, rather it refers to the end of the checking segment or function.



index	*addr*	*target*	*suc*	*flag*
A	0x03	0x0c	D	B
B	0x06	0x0a	C	B
C	0x0b	0x02	A	J
D	0x0c	-	-	-

Figure 4.6: For the example program code in Figure 4.4, the corresponding CFIG is shown on the left hand side. The nodes correspond to the control flow instructions, whereas the edges denote transitions. On the right hand side, the four memory areas are shown which are necessary to store the CFIG. In the *addr* memory, the address of the CFI is stored and the corresponding target address is stored in the *target* column. In the third column (*suc*), the successor CFI index is stored. Finally, the kind of instruction is stored in the last column ($flag$).

Like in the CF method, the information of the CFIG can be used as a specification of the correct branching behavior inside a control flow checker unit and implemented either directly by an FSM or by micro-instructions of a micro-programmed circuit. In case of a micro-programmed circuit implementation, we store for each CFI the start and the target address in memory (*addr* and *target* in Figure 4.6). Also, the index of the successor CFI must be stored inside this memory (*suc* in Figure 4.6). For direct branches, we store the successor CFI for taken branches. If the branch is not taken, the successor CFI is CFI_{i+1}. Finally, we need a flag ($flag$) to distinguish between the different CFI types.

A proper control flow instruction checking algorithm is shown in Listing 4.2. Like the CF algorithm, the inputs are the current program counter *PC* and the next program counter *nPC* and the output is a 0 in case of a correct control flow, or a -1 in case of an error. The checking algorithm needs the four memory columns, introduced in Figure 4.6 and the index *i*, which denotes the next CFI from the current program flow position.

The algorithm is quite similar to the CF method, with the difference of accessing the jump or branch targets and the missing check of basic block ends with a non

Listing 4.2 Control flow instruction (CFI) checking algorithm

```
1  int check_cfi(PC, nPC) {
2    static int i;                          // index i
3    if (addr(i) == PC) {                   // PC is CFI
4      if (flag(i) == 'J') {                // uncond jump
5        if (target(i) == nPC) {            // correct?
6          i = suc(i);
7          return 0;
8        } else return -1;
9      } else {                             // cond branch
10       if (target(i)) == nPC) {           // branch taken
11           i = suc(i);
12           return 0;
13       } else if (PC +1 == nPC)) {        // branch not taken
14           i++;
15           return 0;
16       } else return -1;
17     }
18   } else {                               // non CFI
19     if (PC +1 == nPC) return 0;          // correct?
20     else return -1;
21   }
22 }
```

CFI. In Line 3, we check if the current executed instruction is a CFI. If it is not, the linear successive program flow is checked (Line 18). If the current program counter references to a CFI, we must also distinguish between the different types of CFIs (Line 4 and 9). If the CFI is an unconditional jump, the next program counter should be the value stored in the target memory column (*target*, Line 5). Also, we must update the index i to the index of the successive CFI ($suc(i)$, Line 6). If the current CFI is a conditional branch, we must check if the branch is taken or not (Line 10 and 13). If the branch is taken, the same checking strategy as in the case of unconditional jumps is used. If the branch is not taken, the next program counter must be the current one, incremented by one (Line 14). In both cases, the index i must be recently updated.

Memory Overhead Discussion

In the following, the different memory overheads of both methods shall be compared. The example program in Figure 4.4 has 6 nodes in the CFG and 4 nodes in the CFIG. For the CF method, we need to store only one address for a CFG node (basic block end address *addr*) and the index of the successor block. In the CFI method, we need to store two addresses (control flow instruction address *addr* and target address *target*) and the index of the successor block for each CFIG node. Both methods also

need bits to store the flags for distinguishing the different CFI or basic block types. Usually, the index needs less bits than the addresses of instructions, so the CF method uses less memory than the CFI method for this example.

For measuring the memory overhead for standard user programs, we use the programs from the SPEC CINT2000 [SPE] benchmark in the following (see Section 4.3.1). Table 4.2 shows the memory overhead caused to implement the CF and CFI method for the SPEC CINT2000 benchmark when compiled to the 32-bit SPARC V8 [SPA] architecture. The smallest possible index bit width is chosen for the given program to calculate the memory overhead in bits.

Also, the memory overhead of the checking methods are compared with the memory usage of the test programs. The number of instructions of the test programs are presented in Table 4.1. On the SPARC V8 architecture, each instruction needs 32-bit of memory space. The additional memory overhead of the checker methods are shown in absolute values and in percentage of the memory usage of the test program in Table 4.2.

SPEC prog.	CF method				CFI method			
	# BB	Ind. w. [bits]	Overhead [bits]	[%]	# CFI	Ind. w. [bits]	Overhead [bits]	[%]
gzip	2615	12	109830	17.2	2140	12	154080	24.1
gcc	90819	17	4268493	23.6	79886	17	6151222	33.9
vpr	6029	13	259247	15.6	5054	13	368942	22.3
mcf	514	10	20560	16.6	398	9	27324	22.0
crafty	10205	14	449020	16.9	9009	14	666666	25.1
parser	6384	13	274512	23.3	5212	13	380476	32.3
gap	30484	15	1371780	18.1	24986	15	1873950	24.8
vortex	24978	15	1124010	20.1	21977	15	1648275	29.5
bzip2	1502	11	61582	15.8	1201	11	85271	21.9
twolf	9827	14	432388	13.1	7952	13	580496	17.6

Table 4.2: Required memory overhead of the programs of the SPEC CINT2000 benchmark in bits for the CF and CFI method. Also, the number of basic blocks and control flow instructions, and the corresponding index width is shown. The memory overhead is shown in absolute values and in percentage of the memory usage of the corresponding test program.

The results in Table 4.2 show that the CF method usually produces a lower memory overhead than the CFI method and in a range of typically less than 20%. Note that the shown overhead is for checking the whole program, with all subroutines which is not always the best way. By restricting the checking to only few subroutines which are executed very often and should have a high reliability and security, the memory overhead can be significantly reduced.

Instruction Integrity Checker

The *instruction integrity checker* (IIC) is an extension to the control flow method to check all types of instructions, not only control flow instructions. A *CRC* (cyclic redundancy check) or hash value may be calculated offline for all instructions inside a basic block (at compile-time) and online inside the checker unit [MLS91]. The offline calculated CRCs are stored inside an additional memory which extends the other checker memories (see Figure 4.5). For each basic block, we store the end address $addr$, the index of the next basic block *suc* (for a jump or a taken branch), the flags $flag$ and additionally the CRC or hash value in the memory *iic* (instruction integrity check).

The checker unit calculates a CRC from the instructions during the execution of a basic block. At the last instruction of the basic block, the calculated CRC can be compared with the offline calculated CRC, stored inside the *iic* memory. If the CRCs are not equal, one or more bits are false in the instruction stream. This error can be signaled to the operating system by an interrupt or the system might be rebooted by a hardware reset. A re-execution or correction is not or hardly possible, because we are able to detect an error inside a basic block only at the end of the block.

The instruction integrity checking is not applicable for the CFI method, because there may exist more than one path to a CFI node, whereas in the CF method a basic block is always traversed on the same path. As an example, consider the *if clause* in the program in Figure 4.4. In the CF method, if the branch (if) is not taken, only basic block 5 is traversed. If the branch is taken, basic blocks 4 and 5 is transversed, but basic block 5 is executed on the same way as if the branch was not taken. Unlike in the CFI method, if the branch is taken or not, always the CFI *C* is the successor. However, through the way to *C* the program flow takes different paths, depending on whether the branch is taken or not.

Walking through different paths to a node results in different CRC or hash values. With the IIC method, we are only able to store one CRC or hash value in the additional memory column for one node. Surely, we could extend the memory to store a value for each possible path, but this would result in a huge memory overhead, because we must reserve memory space for each node. This shows that the instruction integrity checker is not practical to the CFI method.

Conclusions

Both introduced methods can only check direct branches and jumps, where start and destination addresses can be extracted from the compiled code.

The advantage of the CF method is that in most cases, fewer additional memory resources are needed than for the CFI method. The disadvantage of the CF method is that memory handling is more difficult. On many processor architectures, the fastest execution of one instruction is one clock cycle. Consider Algorithm 4.1, where we

need access to the *addr* memory for each control flow instruction twice, once for the end address of the basic block (Line 3) and once for the start address of the successor basic block (Line 5 and 10). To achieve this in a single clock cycle, we need a dual-port memory which is more expensive than single port memories. Furthermore, for the second access to the memory, we need first the successor index from the *suc* memory. To do both memory accesses in one clock cycle is nearly impossible on high-clocked processors. Furthermore, the access to the *suc* memory cannot be scheduled one clock cycle before, because if the current basic block consists only of one instruction, and the previous basic block ends with a branch instruction, the current index *i* depends on the result of the executed branch (taken or not). This shows us that we need at least two clock cycles to check a transition in CFG. To ensure that on a basic block end the correct start and destination addresses are available, we might pre-read both values. This can also be done with a single ported *addr* memory. On the first clock cycle, the basic block end address is read from the *addr* memory and the successor basic block index is read from the *suc* memory. On the second clock cycle, the target address is read from the *addr*. But this pre-read can only be done if the basic block consists of more than one instruction. If a basic block consists of only one instruction, we must stall the processor pipeline to verify the control flow instruction to prevent possible erroneous behavior. Fortunately, basic blocks with only one instruction are very rare.

The CFI method, on the other hand, requires only one memory access for each memory. In Line 3 of the Algorithm 4.2, we access the *addr* memory to get the next CFI address. In the same clock cycle we can access the *target* memory to fetch the correct destination address (Line 5 and 10). With the CFI method, it is possible to check a transition in the CFI graph with at least one clock cycle. Therefore, the CFI method has no execution time overhead at all.

The advantages of the CFI method are that the checker unit is very simple and uses only few logic resources. Also, we have no performance impact, because the correct control flow instruction address and target address may be loaded from the memory in a single clock cycle. The disadvantages are that usually more memory resources are needed as for the CF method and that we are not able to check the integrity of non control flow instructions.

Finally, both introduced concepts for control flow checking have the big advantage over [ARRJ06] in being reprogrammable. Thus, only the memory of the control flow checker unit needs to be reprogrammed so to check a different program. No adjustments of the hardware are thus necessary. Moreover, we have no performance impact for verify the control flow like the software-based methods.

4.3.3 Methods for Checking Indirect Jumps/Branches

Checking *indirect jumps* or *branches* is more difficult than direct branches or jumps, because the jump destination cannot be determined from the compiled program code.

In fact, according to the instruction specification of indirect jumps, all possible targets which are inside the reachable area of the jump, are allowed. From the hardware side, also a falsified indirect CFI which jumps to a wrong address is in accordance to the processor specification. Almost all code injection attacks target this behavior by manipulating indirect jump targets (the *return stack*). However, from the software or logical side, there are certainly some restrictions of indirect jump targets: Compilers use indirect jumps in a stylized manner which can be analyzed [LB94]. Almost all indirect jumps which are compiled from a modern program language, like C, C++, or Java, are returns from subroutine, or either belong to a `switch case` clause, which is implemented using a jump table, or are indirect calls which are mainly used in object-oriented languages, like C++ or Java. Indirect jumps which appear in hand-written code are nearly impossible to analyze. Fortunately, hand-written assembler code is used more and more rarely today.

The results reported in Table 4.1 show that returns from subroutine are clearly the main usage of indirect jumps. Upon a call, the address of the back-jump is stored inside a register or a memory stack, and on a return from subroutine, a back-jump to this address is initiated.

Indirect jumps are also used for jump tables to efficiently implement `switch case` clauses. Here, the alternative `case` targets are assembled in a jump table which is addressed by the previously calculated operator. Furthermore, the targets may be direct jumps which lead the control flow to the desired code segment (see Figure 4.7). Another way to use a jump table is to call different functions, depending on an input. Here, the alternative function pointers are stored inside a jump table, whereas the index of the table is calculated with the input value. The address of the desired function is fetched from the table and is called with an indirect jump. Note that jump tables are not often used by compilers. Usually, `switch case` clauses are translated to an `if .. else if` tree. But, depending on the compiler and optimization parameters, indirect jumps might nevertheless occur. Indirect jumps which result from jump table implementations are listed in Table 4.1 under the category “other jumps”.

Finally, the indirect jumps are also used as *indirect calls* (see Table 4.1). During indirect calls, the address of the target function is loaded inside a register and with an indirect jump the function will be called. This occurs mainly in object-oriented programming languages that support function pointers and virtual functions. However, functions called by a jump table also use indirect calls.

The methods for checking indirect jumps that will be described in the following can be categorized into methods using information which are gathered by analysis or simulation at compile-time and methods which are only using runtime information.

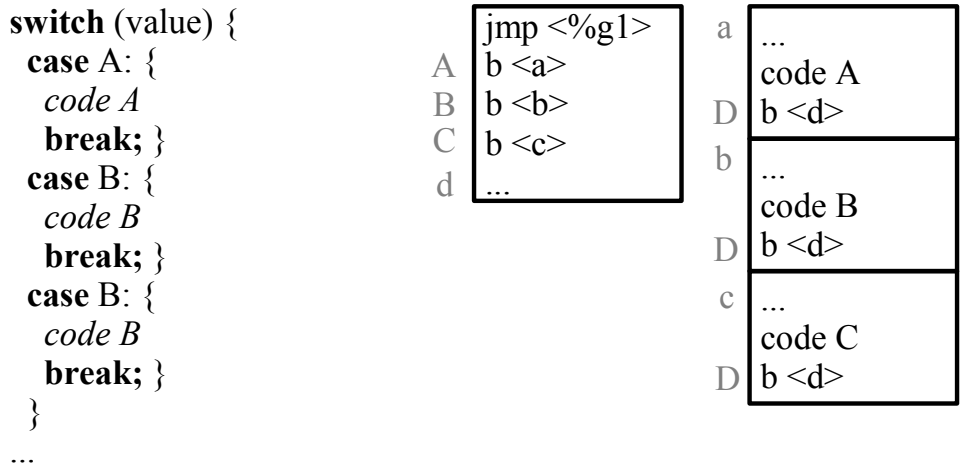


Figure 4.7: An pseudo C example code of a `switch case` clause is shown on the left side. On the right side, a possible implementation in assembler with a jump table and an indirect jump is depicted. The upper case letters (A-D) are direct jump instructions and the corresponding targets are depicted in lower case letters (a-d).

Methods Using Compile-time Information

If we are able to analyze the targets of indirect jumps at compile-time, we can extend our hardware checking units to support multiple jump destinations for monitoring program code that includes indirect jumps (see Section 4.4.2). Cifuentes and Emmerik [CE99] present a method to identify indirect branch targets, if the indirect jump is used within a jump table. Furthermore, simulations with different input stimuli may also be helpful to identify indirect jump targets. However, to get the possible jump targets by simulation requires a high effort.

Another approach is to convert all indirect jumps into direct jumps and branches. Bergstra and Middelburg [BM07] present a method to convert most indirect jumps in a compiled program, including jump tables and returns, into direct jumps and branches. The length of program code could be extremely increased and the performance could be reduced by this method.

Methods Using Runtime Information

Methods using runtime information do not need information from the compiled code. Here, we monitor the control flow at runtime to decide if the execution of the indirect jump is correct or not.

Most indirect jumps are returns from subroutine (see Table 4.1). By executing the return from subroutine instruction, the program counter jumps to the next address

after the instruction, were the subroutine was called. The return address is typically stored in a register inside the CPU so the return instruction is a special indirect jump. Returns can be verified by implementing an additional *hardware stack* [KE91]. On a call (direct or indirect), the return address is stored in the stack and when the return instruction is executed, the back-jump can be verified.

Furthermore, indirect branch prediction units can be used to evaluate an indirect jump address. Branch prediction is used in pipelined processors to avoid pipeline stalls on branches. A prediction is made if a branch will be taken or not and the next instructions will be fetched according to the prediction. If the prediction was right, no stall occurs, if not, the pipeline must be stalled and the right instructions must be fetched.

Indirect branch prediction units predict destinations of indirect jumps. The predictions are made based on the jump behavior in the past [CHP97, SFF+02, JMKP07]. The result of an indirect branch predict unit might be used to evaluate how reliable the jump destination is. If the prediction is correct, then the probability that this jump is correct is high, but if the prediction is incorrect, the jump destination could be false. A non-predicted indirect jump target has a lower trustworthiness. With this method, no exact proposition can be made, but, for example, a higher level autonomic operating system can evaluate this jump confidentially to increase the reliability of the whole system.

4.3.4 Methods for Handling a Corrupt Control Flow

In the sections above, methods for autonomous monitoring the control flow were described. However, what can we do if an error is detected? There are three opportunities:

- The faulty instruction can be re-executed.
- The CPU can be transferred into a secure state.
- The CPU can continue executing the code at a lower reliability level.

If an error in the control flow occurs, the faulty instruction might be re-executed as follows: The error should be detected fast enough to ensure that the state of the CPU is not altered by the erroneous instruction execution. To guarantee this, a possible checker unit must monitor the program counter in the first pipeline stage of a given RISC CPU. Unfortunately, in most architectures, the jump or branch instructions need more than one cycle to execute. So, until the error is detected, some other instructions after the jump might be executed already. After error detection, the program counter is reset to a value previous to the error by looping back the program counter value from a subsequent pipeline step or by a calculated value from the

checker unit. The details of the re-execution process depend highly on the processor architecture and design.

For example, the SPARC V8 architecture allows to execute one instruction after a branch instruction or two instructions after a jump instruction before the branch or jump is performed (see SPARC Architecture Manual [SPA]). If an error is detected and the jump or branch instruction must be re-executed, also these following instructions must be re-executed. It must also be ensured that these instructions cannot alter the state (e.g., register content or memory operations) of the CPU before re-execution. If the retry also fails, the instruction cache can be invalidated to ensure that on the next re-execution, the instructions are transferred again from the memory. If a predefined number of retries fail, the checker unit can lead the CPU into a secure state. Also, the number of retries can be reported to the operating system to show how reliable the CPU is.

Another possibility to react in the case of an error is to transfer the CPU into a secure state. This state can be the reset state or any other state until the program was executed correctly. After reaching this state, the operating system can initialize the CPU with correct data and the CPU can start to execute from this clean state. The invalidation of the data resulting from the erroneous task can be also done by the operating system.

However, the CPU might continue executing the code at lower reliability level with deactivated checker if the task has a low reliability requirement or is further checked by another process.

In all cases, the operating system should be informed about the error and update the internal reliability state of the CPU. If many errors occur, the CPU should only be allowed to execute tasks with low reliability requirements or unimportant tasks, or should finally be excluded by the dispatcher and shut down.

4.3.5 IP Core Control Flow Checking

Control paths in general hardware IPs can be checked also using redundant checker units. These units are able to monitor the correct states and state transitions. In an IP core, the data and control path are typically mixed together. Separation is not easy, but elements like *finite state machines* (FSMs) or *counters* can be assigned to the control paths. For registers spread over the whole core, it depends on the input behavior. If the input of these registers is mainly dependent on the input data of the core, these registers can be assigned to the data paths. If the states and the transitions of these registers can be modeled by an FSM, these registers can be counted to the control paths. One indication for the control path behavior can be, if the number of valid states is restricted to a defined *state space*, which can be calculated offline. The assignment of registers to the data or control paths is not always clear and must be decided individually. Registers which have more control paths characteristics presenting a state space, where not all states or transitions between states may be valid.

The validity of these states and transitions can be checked using additional checker units.

The control path registers can be grouped to so-called *checking domains*. A checking domain can be, for example, an FSM, a counter, or individual registers which are related. For each checking domain, it is reasonable to generate a separate checking unit. These checking units can monitor the state only or the state and transitions of the corresponding checking domain. If a checking unit detects an error, a controller can signal the error to a higher level observation unit or operating system (see Figure 4.8).

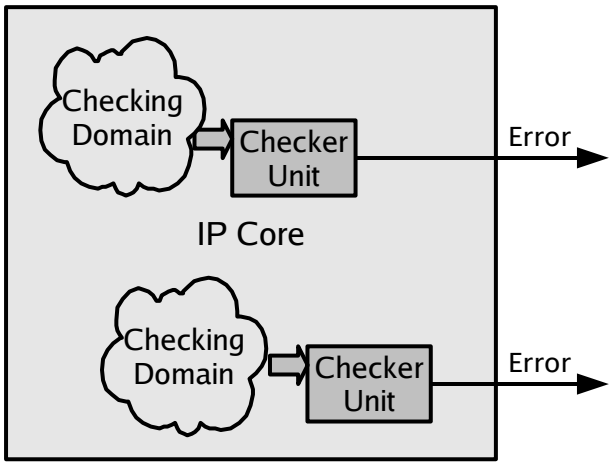


Figure 4.8: Different checking domains in a hardware IP core. Each checking domain has its own checker unit and an error signal.

The valid states and transitions of a checking domain can be specified at design or synthesis time, and the checker unit can automatically be generated and inserted in the core. Valid states of checking domains can be, for example, the states of an FSM or a specified counter range. Valid transitions can be, for example, the transitions of an FSM or the increment or decrement of a counter.

If an error occurs, the checker unit or the higher level observation unit may trigger a reset of the IP core to prevent an erroneous state. The problem of this behavior is that the current processed data get lost and the core must be re-initialized. Another possibility is to run the core at a lower reliability level or invalidate the processed data.

4.4 Architectures for Control Flow Checking

In this section, we present hardware architectures to implement the control flow checking methods described in the last section.

First, architectures for methods of checking the control flow of embedded CPUs are presented. These architectures are implemented in a *module-based* way so that methods for checking different types of instructions can be easily combined. First, in Section 4.4.1, architectures for checking direct jumps and branches as well as non control flow instructions are presented. These two direct CFI checking architectures provide basic modules which can be extended by other checking modules or interfaces. For indirect jumps, we provide extensions in Section 4.4.2 which use compile-time information and a return stack architecture. Techniques to handle interrupts and traps are implemented in a further extension. Also, techniques for checking the integrity of all instructions are presented. Finally, the extension to re-execute erroneous instructions to correct a corrupt control flow is presented in Section 4.4.6. Our architecture concept for checking the control flow of embedded CPUs is modular in the sense that the above coverage aspects can be traded off for implementation overhead.

Furthermore, architectures and techniques for checking general IP cores are introduced in Section 4.4.8, and in Section 4.4.9 and 4.4.10, a fault coverage and area overhead discussion of all proposed architectures and extensions are presented.

4.4.1 Handling Direct Jumps and Branches

In this section, we describe the basic architectures for the CF and CFI method introduced in Section 4.3.2 to check the control flow of direct jumps and branches as well as non control flow instructions. In particular, the CFG/CFIG information is mapped to dedicated memories as it will be shown next.

To check the control flow, the checker must know the instruction's program address PC_n and the address of the next instruction PC_{n+1} to execute. Since most CPU architectures today are pipelined, these addresses can easily be taken from successive pipeline stages of the program counter.

If no jump or branch instruction occurs, the next instruction address is typically one instruction word higher than the value of the current program counter ($PC_{n+1} = PC_n + 1$). So, an incremented instruction address can be compared to the address after the instruction (see comparator a in Figure 4.9 and 4.10). Otherwise, if the current instruction is a direct jump or branch instruction, the next program counter is the jump or branch destination if the branch is taken. If the branch is not taken, the next address in the program code is the next instruction address.

In Section 4.3.2, we have shown how the correct address of direct jumps and branches as well as the correct targets can be gathered from the program code. Here, we present the architectures for the corresponding methods.

An Architecture for the CFI Method

Each pair of control flow instruction address and target address is stored in two memories of the checker unit, one for the start and one for the target address (see Figure 4.9). The addresses of the branch or jump instructions are stored subsequently in the start address memory (*sAdrRam*) and the corresponding targets in the jump address memory (*jAdrRam*). Also, a checker unit program counter (*CUPC*) is needed which points to the index of these memories where the address of the next direct branch or jump is stored. The CUPC implements the index of the next CFI described in Section 4.3.2. The start address in the *START ADDR* register is compared to the current program counter to determine whether a branch or jump instruction is executed (comparator *b*). In this case, the following program counter value is compared to the address of the jump address RAM (*TARGET ADDR* register) to verify the correct execution of the branch or the jump (comparator *c*). Now, the CUPC must point to the next branch or jump address. This can be achieved by introducing a third memory (*ctrlRam* or the *suc* memory in Section 4.3.2) where the next CUPC is stored for each branch or jump. The next CUPC value is a part of the *ctrlRam* and therefore for the *CTRL* register. After successfully checking a CFI, the values for the next CFI are loaded from all three memories into the corresponding registers: *START ADDR*, *TARGET ADDR*, and the *CTRL* register. By reaching the next CFI (comparator *b* delivers true), the CFI can be checked without any wait cycle.

If the CFI is a branch, it must also be determined if the branch is taken or not. If the branch is taken, the next CUPC has the value which is stored in the *ctrlRam*. If the branch is not taken, then the CUPC is incremented. To distinguish between jumps and branches, we need additional memory space to store this information (see the *flags* memory in Section 4.3.2). Here, we join the *suc* and *flag* memory columns to one memory, called *ctrlRam*.

The CUPC and the *ctrlRam* present a micro-programmed architecture which implements the CFIG. The CUPC can be compared with the index of the CFI. The transitions of the CFIG are stored in the *ctrlRam*.

An Architecture for the CF Method

In the following, we describe an architecture for implementing the CF method introduced in Section 4.3.2. To check the correct execution of control flow instructions, we need the address of the currently executed instruction PC_n and the addresses of the next instruction to execute (PC_{n+1}). Checking the right execution sequence of program counter values is done similar to the CFI method (see Section 4.4.1). We are also using three comparators and a control RAM (*ctrlRam*) with a control unit program counter (*CUPC*) (see Figure 4.10).

Different from the previously proposed CFI method, however, the CF method is using only two RAMs instead of three: the address RAM (*adrRam*) and the *ctrl-*

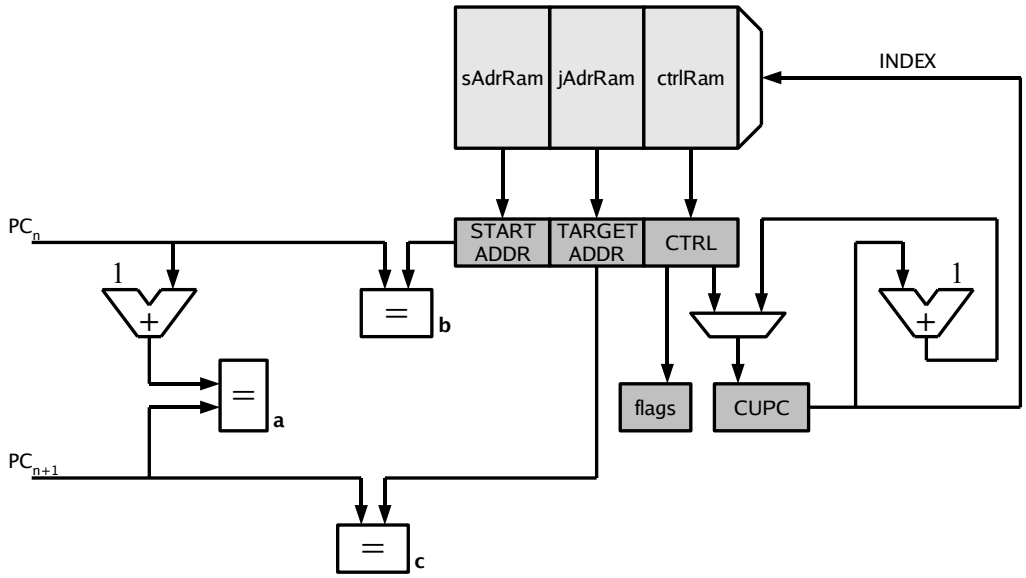


Figure 4.9: Architecture for a control flow checker unit with three memories and three comparators *a*, *b*, and *c*. Also, the control unit program counter (*CUPC*) is shown.

Ram. The end addresses of all basic blocks (*BB*) of the checked code sequence are stored successively in the memory *adrRam*. A memory for the target addresses is not needed, because the jump and branch targets can be calculated from the values in the *adrRam* (see Section 4.3.2). The index (CUPC value) of the following basic block to be executed after the currently executed basic block and some control flags are stored in the memory *ctrlRam*. The control flags denote the kind of the last instruction of the basic block. This can be either a *CALL, BRANCH, RETURN,* or a non control flow instruction.

The correct address of a control flow instruction and its target address cannot easily be read out of the RAMs like in the case of the CFI method. The end of the current basic block is stored in $adrRam[CUPC]$, and the correct address of the following instruction is stored in $adrRam[ctrlRam[CUPC]-1]+1$. In contrast to the CFI method, however, we need two accesses to the *adrRam* in order to obtain the correct pair of instruction address and next instruction address at the end of a basic block. However, the control flow check should be possible within one clock cycle to have the ability to re-execute the instruction. To avoid a wait cycle, the correct pair of address and next address for the end of the basic block can be read when entering the basic block. Nevertheless, if the basic block consists of only one instruction, a wait cycle needs to be introduced.

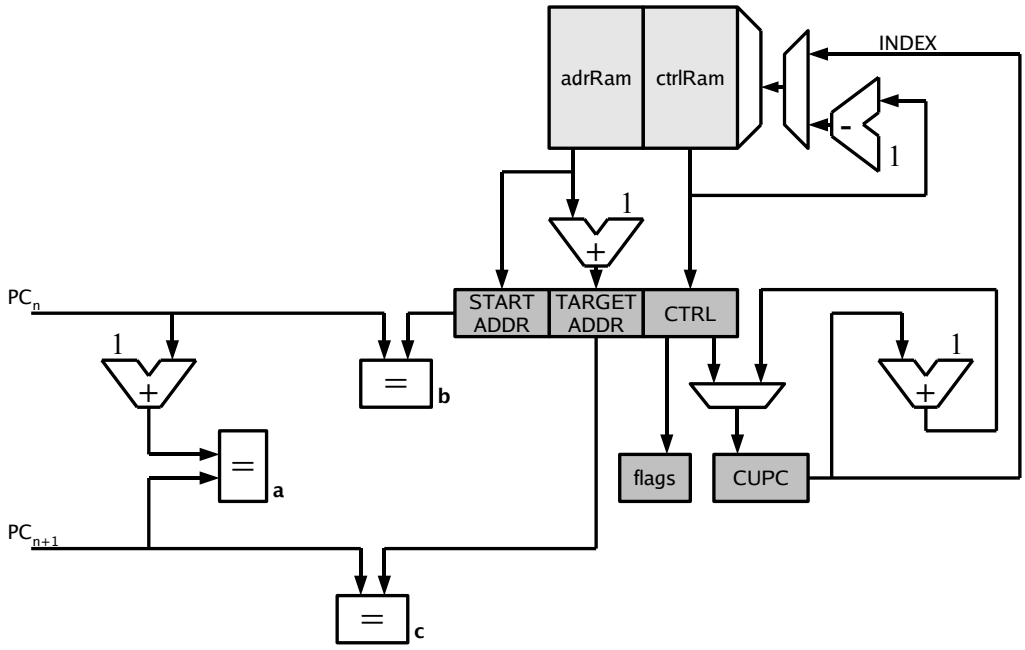


Figure 4.10: Architecture for control flow checking using the CF method. The two RAMs, the comparators and the *CUPC* are shown.

Usually, the processors stall in many wait cycles due to cache misses or pipeline stalls. However, an additional CPU wait cycle will only be produced for each basic block which consists of only one single instruction and if no other wait cycles which are not caused by the checker unit exists. So, it is not even sure that a basic block with only one instruction will cause an additional wait cycle when applying the CF method.

Activation and Deactivation of the Checker Unit

Besides the flags which indicate the different CFIs or basis block ends, also flags which control the checker unit itself can be stored in the *ctrlRam*. So, the checker unit can be activated or deactivated based on specific program addresses (see Figure 4.11). This can be done by storing the checking start or end address in the *sAdrRam* for the CFI or in the *adrRam* for the CF method and setting the checking start or end flag in the corresponding cell in the *ctrlRam*. These are additional entries in the checker memories and do not represent a checking point. However, it is further possible to use available checking point entries for the activation or deactivation. If the program flow reaches the start address, the checker unit will be activated, or, if the checking end address is reached, the checker unit deactivates itself. Furthermore, not only global activation or deactivation can be achieved by setting these flags. Parts

of the program flow, e.g., non-critical sections or sections which cannot be checked due to not supported indirect jumps, might be excluded from the checking process by setting the checking start and end flags.

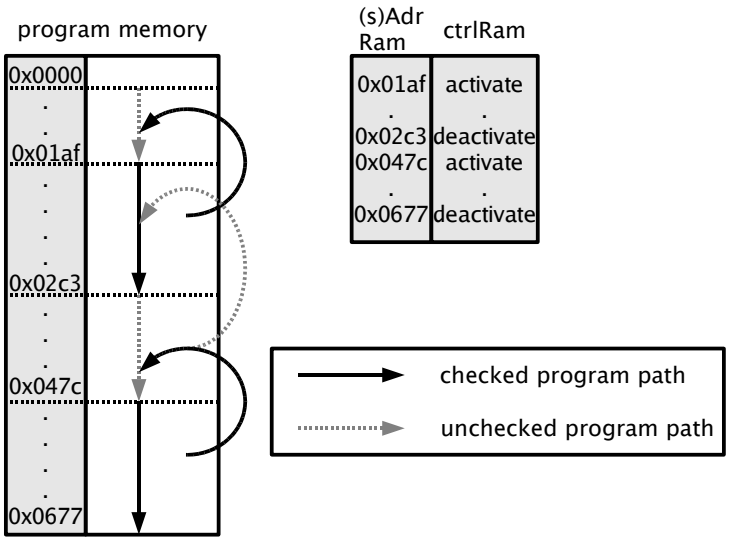


Figure 4.11: The checker unit can be activated or deactivated on certain program addresses. If a specified address is reached, the checker unit enables or disables itself.

4.4.2 Handling Indirect Jumps and Branches

Approaches to extend the proposed direct checking architectures for checking indirect jumps are discussed in this section. First, an extension for checking indirect returns from subroutine is presented. In the second part, architectures for checking general indirect jumps are shown.

Checking Returns from a Subroutine

The major use of indirect jumps occurs in the form of returns from a subroutine, see, e.g., Table 4.1. By executing a *return from subroutine* instruction, the program counter jumps to the next address after the instruction from where the subroutine was called. The return address is typically stored in a CPU register, so the return instruction is a special indirect jump. Returns can be verified also in our approach by introducing an additional *hardware stack*. Upon a call (direct or indirect), the return address is stored in the stack. Once the return instruction is executed, the correctness of the target address can be verified.

To integrate the return stack extension into the basic architecture (see Figure 4.12), we need the return stack itself, and an additional comparator *d*. Note that the return extension can be used for both the CFI and CF basic architecture. The addresses of return instructions are stored in the *sAdrRam* for the CFI method, or in the *adrRam* for the CF method, together with the entries for checking direct jumps and branches. An additional flag in the ctrlRam, the *RETURN* flag, identifies these entries as the addresses of return instructions. The field in the *jAdrRam* for the CFI method, and the field for the following CUPC (index) for both methods is left empty.

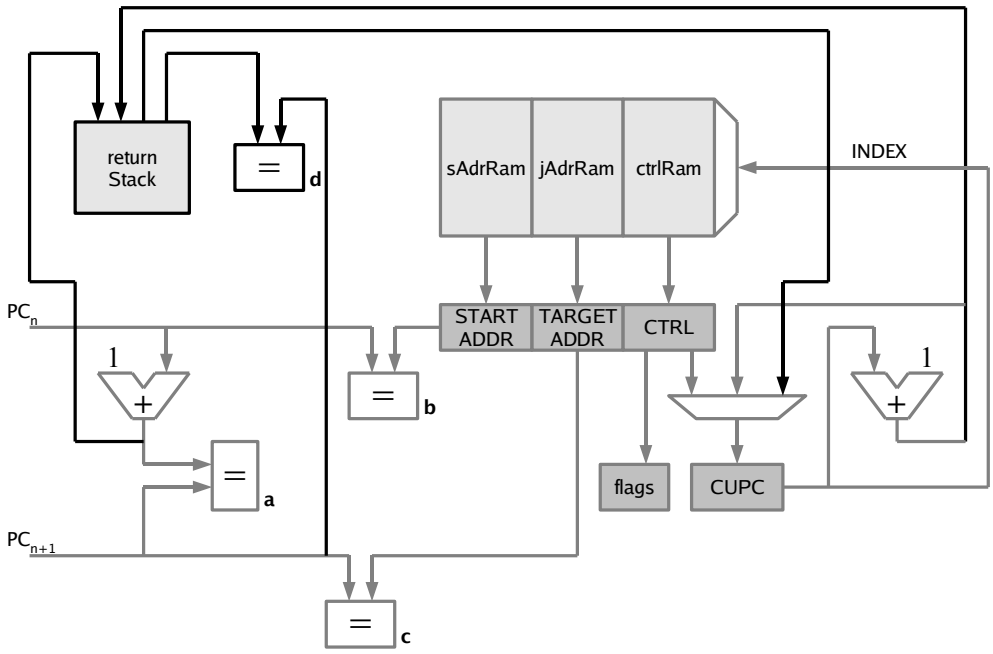


Figure 4.12: The integration of the return stack extensions to the basis architecture (in this figure the CFI architecture). Upon a call, the next instruction address and the next CUPC must be stored on the stack. The return address can be verified by the comparator *d*. Furthermore, the CUPC stored in the stack must write back to the CUPC register to keep the checker unit in sync.

Upon a call which is identified by the *CALL* flag, the next instruction-address and the next CUPC value is stored in the stack. The comparator *b* and the *RETURN* flag in the *CTRL* register identify a return instruction. With the help of comparator *d*, the checker unit can verify the back-jump to the return destination. Finally, the CUPC must be updated to the next checking point (next CFI instruction for the CFI method,

or the next BB end for the CF method) which is stored in the stack together with the return target address.

With this architecture, we are able to check all returns from subroutine, even if the subroutine is called from different locations in the program code. The depth of the stack is, however, dependent on the maximum allowed nesting of function calls. If the stack overflows, the checker unit is unable to check the following program flow.

Checking General Indirect Jumps

For checking general indirect jumps, the program code can be analyzed or simulated to detect all possible destinations of all indirect jumps (see Section 4.3.3). These destinations are marked as jump-able and can be stored in an additional Ram (*iJmpRam*) inside the checker unit. Also, the addresses of all indirect jump instructions are stored in the *sAdrRam* (for the CFI method) or *adrRam* (for the CF method). In the corresponding *ctrlRam*, these addresses are marked as indirect jumps with an additional flag (see Figure 4.13).

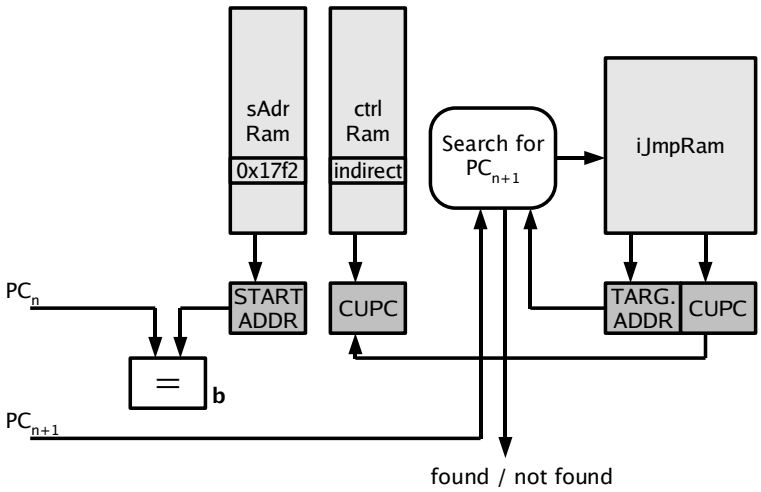


Figure 4.13: If an indirect jump instruction is reached, the control checker searches in the *iJmpRam* the jump destination. If the destination is found, the execution is correct.

If the program flow reaches an indirect jump, the checker unit searches the whole *iJmpRam* and compares the content with the following program counter address. If the address can be found, the execution is assumed allowed, and thus correct. In the other case, the checker unit reports an error. In addition to the destination address, the next value of the *CUPC* must be stored in the *iJmpRam* to synchronize the CUPC with the real program counter to detect the next checking point correctly. Searching

the destination address in the *iJmpRam* may take more than a single clock cycle. In the worst case, all Ram rows must be read and compared, and for each row, one clock cycle is needed. To speed up the search, a bisection method can be used if the addresses in the *iJmpRam* are stored successively.

If the search operation takes place, the execution of the CPU must be halted to enable a fast report of the error in order to correct it if necessary. The advantage of this method is the flexibility in the number of different jump destinations which are only limited by the number of rows of the memory *iJmpRam*. The disadvantages are the slow execution of indirect jumps and the resulting performance impact on the CPU.

Indirect jump analysis in [CHP97] shows that the number of indirect jump destinations is small in most cases. So, the *iJmpRam* can be organized such that all destinations of one indirect jump can be stored in one row (see Figure 4.14). A maximum number of stored destinations D_{max} in a row must be defined. For each jump destination, the corresponding CUPC value must also be stored in the *iJmpRam*. If an indirect jump has more destinations than D_{max}, only D_{max} destinations can be verified, or an additional column in the *iJmpRam* is inserted, where a pointer of the row with the next D_{max} destinations is stored. The address of the indirect jump instruction is stored in the *sAdrRam* or *adrRam* as in the first approach. Additionally to the indirect jump flag, the address of the row from the *iJmpRam* with the jump destinations is stored in the *ctrlRam*. If the program flow reaches an indirect jump, the state of the next program counter is compared with all destinations in one row of the *iJmpRam*. With parallel comparators, this can be done in a single clock cycle per row. If the next state of the program counter equals to one destination, the execution of the indirect jump is correct and CUPC is updated with the corresponding value form *iJmpRam*. The advantages of this method are the fast verification of the jump destinations and individual checking for each indirect jump. The disadvantage is a larger resource overhead due to parallel comparators and additional memory cells.

4.4.3 Handling Interrupts and Traps

A jump may also occur in case of a trap or an interrupt. If there is an exception during the execution of an instruction (e.g., a division by zero), a trap is triggered. Then, a jump to a fixed or predefined address where the trap table is stored, is typically induced. The different kinds of traps are distinguished in the trap table, and the right trap service routine is called. After the successful execution of the *trap service routine*, a back-jump to the instruction following the trap is initialized. Interrupts are handled similarly with a fixed or predefined address for the interrupt table.

It is important to note that an interrupt can occur asynchronously with respect to program execution. So, direct jumps to the base address for the trap and interrupt table are always allowed. Now, this base address for the trap and interrupt table can also be stored in the checker, and therefore, these jumps may be verified, too.

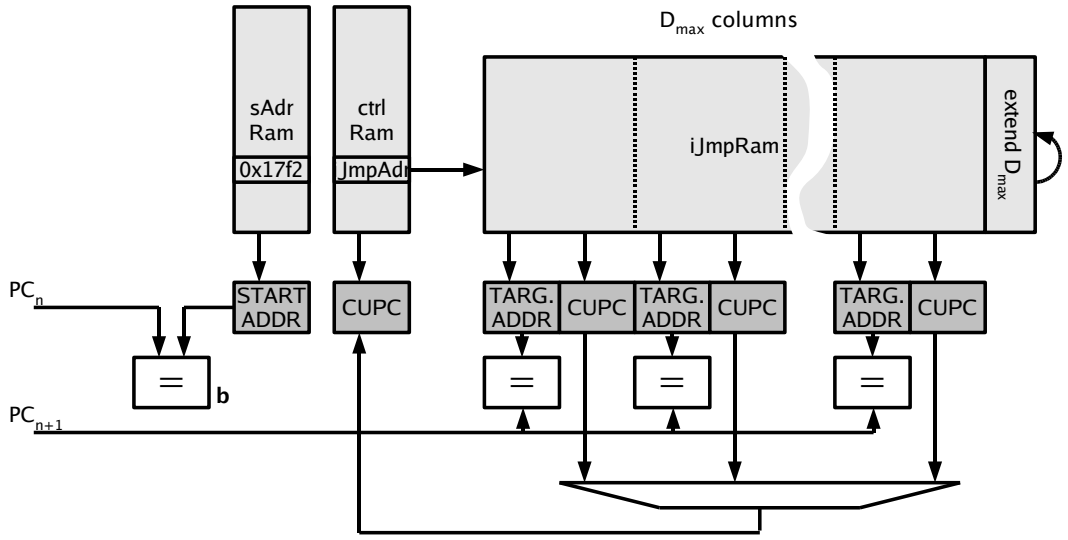


Figure 4.14: A faster architecture for checking indirect jumps. With parallel memories and comparators, the checker can simultaneously verify up to D_{max} jump destinations in one clock cycle.

Also, back-jumps may be checked, because the return address is automatically stored on the return stack (see Section 4.4.2). In some architectures, the base address for the trap and/or interrupt table can be modified by a special register (e.g., SPARC V8 architecture [SPA]). For these architectures, the checker unit must monitor this register and adapt this changes if the register is altered.

If the checker unit should be active during the execution of the trap or interrupt service routine, all CFIs of these routines must be analyzed and inserted into the checker memories. Unfortunately, these service routines are "hot spots" for indirect jumps which might be a problem if the current checker unit does not support these instructions.

Another possibility is to automatically deactivate the checker unit at an interrupt or trap and re-activate the unit at the back-jump. This can be done by monitoring the program counter. If a jump to the trap or interrupt table is initiated, the checker deactivates itself. The next instruction address of the program code and the corresponding *CUPC* value is stored inside the return stack. The checker unit monitors now the program counter until the back-jump address is reached. At this point, the checker activates itself. Another possibility is to use signals from the instruction pipeline which identify a trap or interrupt as well as the return from trap or interrupt. If the trap signal is set, the checker unit is deactivated and if the return from trap is set, the unit will be activated again. One problem of this architecture is that the back-jump target from the service routine must be at the same position or one instruction

word higher in the code where the trap or interrupt occurred. For example of *context switching* in *multi tasking operation systems*, this is not true.

Multi tasking operating systems usually use software interrupts to implement context switching. The currently executed task can be interrupted on every instruction, and another task becomes active. The problem with the checking unit is that the return address from interrupt is not the address before the interrupt. Should the checker unit work also in a multi tasking environment, then the context of the checker unit, the CUPC and the return stack, must be adapted to the new task.

This can be done by the operating system or autonomously by the checker unit. For storing the different checker unit contexts for each task, we need additional memory space. This can be a local memory on chip or the main external memory of the processor. The advantage of the local memory may be a fast switch between the contexts. Nevertheless, the local memory limits the number of currently executed tasks. The disadvantage of the main memory strategy may be a high latency penalty for the data transfer which results in a high switching time of tasks.

Another opportunity is to allow only one task to be checked at a time. The checker unit is disabled if the CPU executes another task and is enabled if the checked task is continued. Using this strategy, the interrupt for context switching in the checked task disables the checker unit and if the program flow jumps back to the checked task, the checker unit will be re-activated. The advantage is that the content of the CUPC and the return stack may not be changed at a context switch.

4.4.4 Checking Conditional Branches

So far, the correctness of the decision (for taking or not taking a branch) in case of conditional branches has not been considered yet. It was only checked whether a destination is one of the two possible targets according to the program specification.

The condition of a direct branch is usually evaluated in two phases or instructions in RISC architectures. In the first phase, two values are usually compared by a compare or subtract instruction (using the ALU) before the actual branch instruction is processed. The operands in the first phase are only know at runtime. The result of this operation affects the ALU flags (called *integer condition codes* (icc) in the SPARC ISA). Typically, there are four flags: n, z, v, c. Flag n is set if the result is negative, flag z is set if the result is zero, v is set if an overflow occurred on the calculation, and c is the carry flag. During the second phase, the actual branch instruction is executed. The condition encoded inside the instruction is evaluated using the ALU flags to decide if a branch should be taken or not. For example, a *branch on equal* (BE) instruction is taken if the flag z is set, or a *branch on signed greater* (BG) is taken if $f(z,n,v) = \textbf{not}(z \textbf{ or } (n \textbf{ xor } v))$ evaluates to true.

Regardless of whether a temporal or permanent hardware fault occurs in the first or in the second phase, the resulting branch target may be wrong. For example, all branches might be wrongly taken because of a permanent fault on the evaluation of

the branch condition. Errors which occur in the first phase can be detected by data path protection methods (see Section 4.5.5), while errors occurring in the second phase can neither be handled with the basic control flow checking or with data path protection methods. To check for these kinds of errors, conditions can be redundantly decoded from the branch instructions which are then evaluated with the result flags of the ALU or other modules (depending on the instruction). This additional decoding of a branch condition can be used to crosscheck the executed branch. This extension for the above control flow checking methods allows the checker unit to address only one destination address for the program counter. Figure 4.15 shows a possible architecture for this extension.

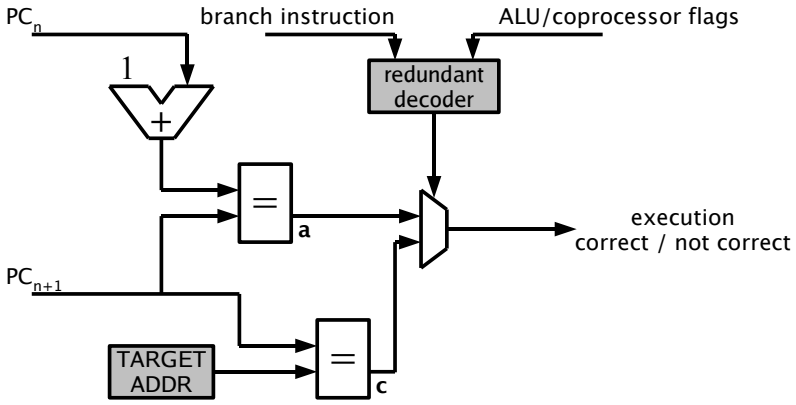


Figure 4.15: Redundant evaluation of the branching condition facilitates the detection of errors in the program counter logic and branch instruction decoder.

4.4.5 Instruction Integrity Checker

The instruction integrity checker (*IIC*) can be used in combination with the CF method to check all types of instructions, not only control flow instructions. A *CRC* (cyclic redundancy check) value is calculated for all instructions inside a basic block offline (at compile-time) and online inside the checker unit (see Section 4.3.2). The offline calculated CRCs are stored inside an additional memory (*crcRam*, or *iic* in Section 4.3.2) which extends the *adrRam* and *ctrlRam* (see Figure 4.16). For each basic block, we store now the end address, the index of the next basic block (for a jump or a taken branch), and additionally the CRC.

The checker unit calculates a CRC from the instructions during the execution of a basic block. At the last instruction of the basic block, the calculated CRC can be compared with the offline calculated CRC stored inside the *crcRam* (comparator

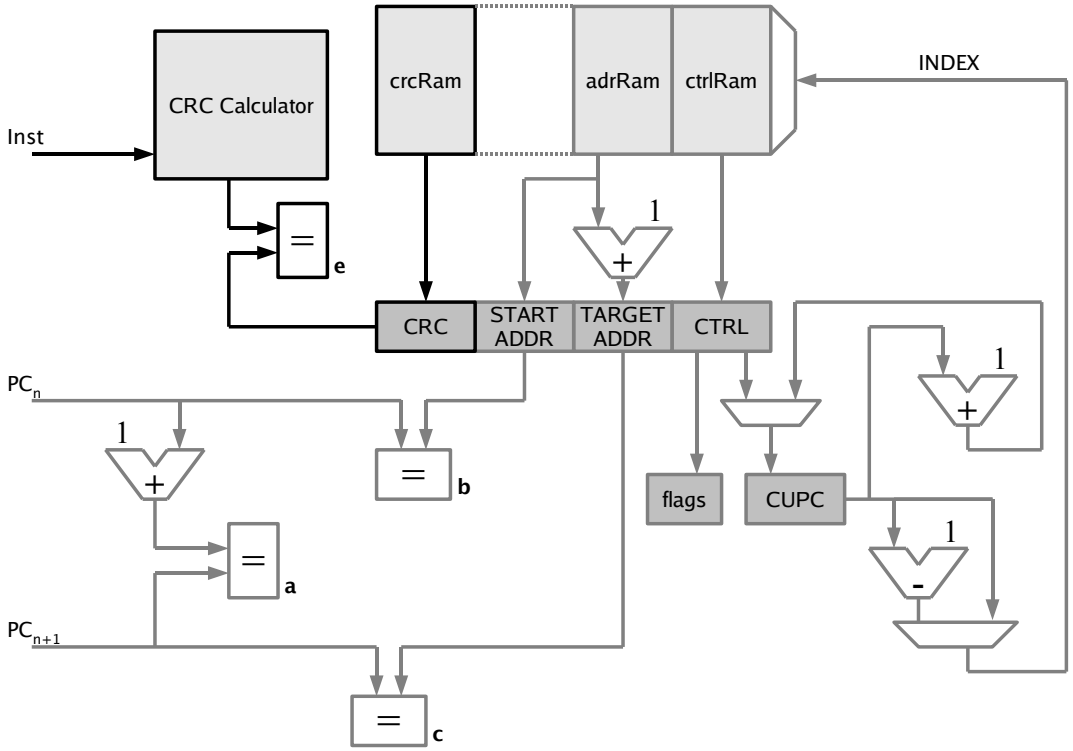


Figure 4.16: The basic CF architecture extended by an instruction integrity checker is shown. For each basic block, a CRC value is calculated at compile-time and stored in the *crcRam*. At runtime, a unit calculates the CRC value of the executed instructions (input Inst). At the end of a basic block, the value from the *crcRam* and the calculated value are compared (comparator *e*).

e). After a successful compare, the online calculation of the CRC must be reset to generate the correct value for the next basic block. If the CRCs are not equal, one or more bits are false in the instruction stream. This error can be signaled to the operating system by an interrupt or the system might be rebooted by a hardware reset. A re-execution is not possible, because we are able to detect only an error inside a basic block at the end of the block. The error detection latency would be too high to correct this error by re-execution.

The CRC value bit width is important for the robustness of the error detection against multiple bit errors. On the other hand, a large CRC value causes a higher area and memory overhead.

4.4.6 Repairing a Corrupt Control Flow by Re-Execution

If an erroneous control flow caused by a corrupt instruction is detected, the corresponding instruction must be annulled and re-executed. The problem is that the entrance of the erroneous instruction in the processor pipeline and the observable effect of the corrupted control flow do not happen at the same time. In fact, many clock cycles can elapse between both events. Furthermore, due to faults in the processor pipeline, the instruction might become erroneous inside the pipeline after it was fetched from the memory. In both cases, many other instructions could enter the processor pipeline after the corrupted instruction.

The first step after the detection of the erroneous control flow is to stall the responsible instruction and all following instructions in the pipeline at the corresponding pipeline stages. Many processor architectures support an annul flag which prevents the corresponding instructions from execution. This flag can be used for the erroneous as well as the following instruction in the pipeline. This should be done fast enough to prevent the erroneous control flow to alter the state of the processor, e.g, write registers, or initiate memory write accesses. Fortunately, these actions happen usually at the last pipeline stage. Therefore, the detection of the corrupted control flow should be as fast as possible which results that the checker unit should monitor the control flow on the first pipeline stage.

The second step is to insert the address of the corrupted instruction at the first pipeline stage which results in a re-fetch and re-execution of the instruction. To identify the responsible instruction and the corresponding instruction address, the information of the stored graph inside the checker memories can be used.

The example program in Figure 4.17 demonstrates the proposed re-execution technique. On the left side, let the branch instruction at address $0x4d$ be responsible for a wrong branch target ($0xde$). The checker unit cannot recognize this error before the corrupt instruction reaches the third pipeline stage. Therefore, the branch and the following two instructions are annulled by setting a corresponding annul flag A and the branch instruction address is forwarded to the pipeline by the checker unit to fetch the branch instruction again. During the second try shown on the right side, let the branch be executed correctly. Furthermore, the previously annulled instructions have now moved to later pipeline stages, but the annul bit prevents these instructions from execution.

To support this re-execution technique by the basic checker architectures, we need two additional outputs of the checker unit. The first one is the instruction address (*CPCREEXPC*) from where the re-execution should start. The second output (*CPCREEXCODE*) tells the processor pipeline how many instructions should be annulled. The *CPCREEXPC* signal is derived from the *STARTADDR* register, if the the erroneous instruction is an CFI. If not, the signal is taken from a history register, which stores the program counter value of the previous cycle (see Figure 4.18). The number of annulled instructions (*CPCREEXCODE*) can be derived from the checker flags

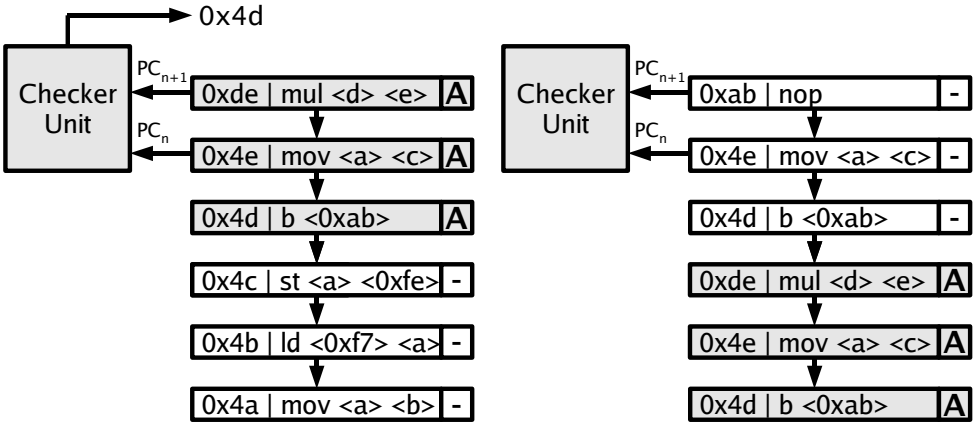


Figure 4.17: A processor pipeline with checker unit and an example program before (left side) and after (right side) the successful re-execution of an erroneous instruction to demonstrate this technique. Each instruction carries in each pipeline stage an additional annul flag which can be set (A) or not (-) to prevent this instruction from being executed. Note that the instruction set architecture used in this example has a *delayed control flow* concept. This means that the branch is executed after a delay of a single instruction (see Section 4.5.1 for the SPARC delayed control flow concept).

which denote the type of the next checking point instruction (CFI instruction or BB end) as well as the output of comparator *b* which identifies a checking point. Note that the re-execution technique can be adapted for the CF as well as for the CFI basic architecture.

Further, the processor pipeline must be extended to handle the re-execution from the *CPCREEXPC* address as well as the invalidation of the corresponding instructions. The number of instructions to be annulled depends on the erroneous instruction and the processor architecture and implementation.

4.4.7 Bus Interface

By introducing an additional *bus interface* from the checker memories to the system bus, the checker memory contents become accessible also from the processor side. This can be very useful for debugging or for updating the memory content, if a new software program is loaded into the system. The disadvantage of this concept is that also malicious programs might get access to the checker memories, and thus open a side channel for security attacks.

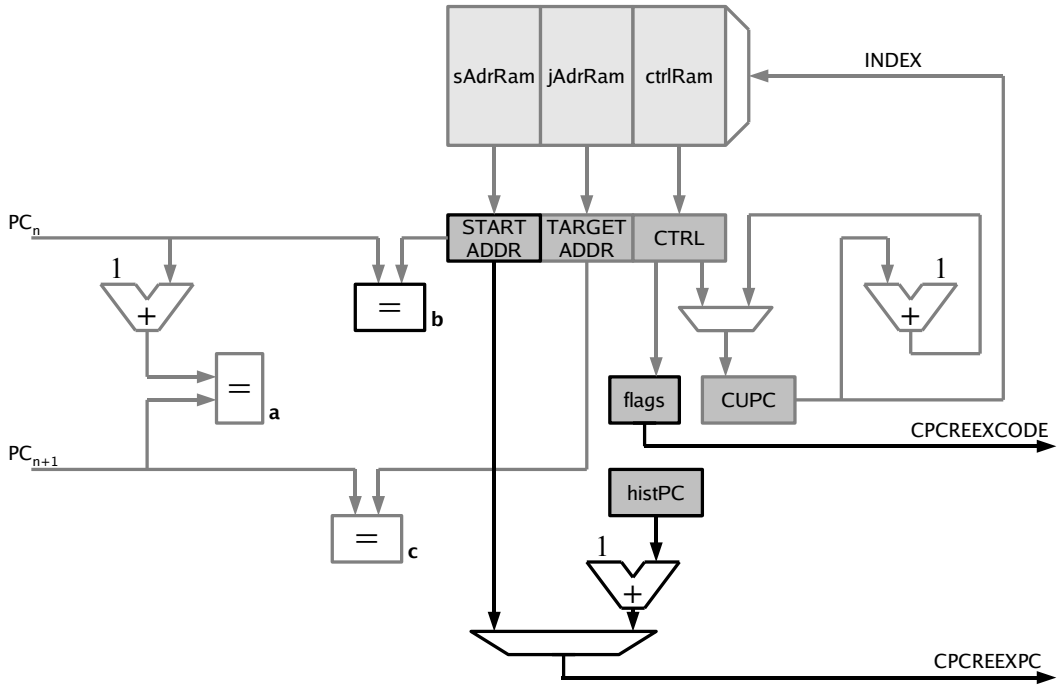


Figure 4.18: The basic CFI architecture with the re-execution extension. The two additional outputs (*CPCREXPC* and *CPCREEXCODE*) are shown. Note, the re-execution extensions work also with the CF basis architecture as well as with the return stack extension.

Another fact is that internal on-chip memory is very expensive (see Section 1.2.6). If the checker unit has a bus interface, the contents or part of the content of the checker memory may also be stored in the external system memory. Only the content for checking the current part of the program (e.g., the current function or a set of functions which are currently in use) may therefore be held in the internal checker memories. If the checker needs information which is not stored inside the local checker memories, the checker can generate a page-fault-like event to reload the checker memories with the needed contents. This concept of caching may reduce the amount of internal memory overhead significantly.

To have access to an existing system memory, the checker memories need a bus interface. The architecture and implementation of the bus interface depends highly on the used bus protocol.

4.4.8 IP Core Control Flow Checking

In Section 4.3.5, possibilities for identifying control path registers in general IP cores are shown. In this section, we concentrate on checking *finite state machines* (FSMs) in general and *counters* in control paths when specified by FSMs.

An FSM is defined as a 6-tuple $(\Sigma_M, \Gamma_M, S_M, s0_M, \delta_M, \omega_M)$, where Σ_M is the input alphabet, Γ_M is the output alphabet, S_M is a finite set of states, $s0_M$ is the initial state, δ_M is the state-transition function, and ω_M is the output function. The internal state $q_M \in S_M$ is stored in a state register. The transitions between the states are controlled by the state-transition function $\delta_M : S_M \times \Sigma_M \rightarrow S_M$ which uses the current state q_M and the current inputs $x_M \in \Sigma_M$ into the FSM to calculate the next state $\delta(q_M, x_M)$. This function is implemented using combinatorial logic. If a *Moore model* is used, the outputs Γ_M depend only on the states: $\omega_M : S_M \rightarrow \Gamma_M$. Whereas the output of a *Mealy machine* depends on the states and the inputs: $\omega_M : S_M \times \Sigma_M \rightarrow \Gamma_M$. Faults (e.g., SEU, SET) inside an FSM can lead to an erroneous state or to erroneous transitions between the states. The general IP core checking methods can check if a state q_M is valid, i.e., $q_M \in S_M$. Furthermore, it checks for a given next state q_{Mn} if this state is valid according to δ_M, i.e., $\exists x_M \in \Sigma_M : \delta_M(q_{Mn-1}, x_M) = q_{Mn}$. It does not check the condition and the timing depending on the inputs x_M when a state is entered or left.

A counter is a specific FSM, more precisely the implementation of a specific Moore automaton. The state q_M is given uniquely by the counter value. The states are sequentially ordered in a chain, which can be traversed in any direction. Here, we can also check the states (the counter range) and the transitions between the states (counting up/down).

Checking the State

An FSM has a finite number of states $q_M \in S_M$. These states are encoded in a specified manner into a state vector, which can be implemented and stored by a register. There exist many different encodings, e.g., gray, sequential, or 1-hot coding. Depending on the encoding, and the available bits for the state register, the number of presentable states can be higher than the actually used states. This means that not all possible states which can be represented by an n-bit register are valid.

To check if a state encoded into a register value is valid, we can use a combinatorial logic which has the state register q_M as input and an error signal as output. If the state is valid, i.e., $q_M \in S_M$, the output is '0', otherwise '1'. A straight forward implementation of this logic is a lookup table, where all invalid states are marked with a '1' as output. One problem with this technique is that an erroneous transition from one valid state to an other valid state is not detected. If the state vector has a normal binary encoding, the number of invalid state encoding is low which may lead to many undetected erroneous transitions.

The valid range for counter values can also be restricted. A checker unit can monitor the counter value and checks if this value is inside the specified range. This can also be done using a lookup table, where all invalid states are marked, but here an implementation with comparators is more resource-efficient.

Checking the Transitions

The transitions between the states can be checked by registering the current state q_{Mn} into a second state register with one clock cycle delay q_{Mn-1} (see Figure 4.19). Looking at both registers, we see the transition from the old to the new state: $q_{Mn-1} \rightarrow q_{Mn}$. Now, both registers can be used as an input for a combinatorial logic which determines if the transition is correct ($\exists x_M \in \Sigma_M : \delta_M(q_{Mn-1}, x_M) = q_{Mn}$) or not ($\nexists x_M \in \Sigma_M : \delta_M(q_{Mn-1}, x_M) = q_{Mn}$). Straight forward, this can also be done using a lookup table. Both inputs from the registers are connected together to become an new bit vector which has twice the size of the register. In the lookup table, all correct transitions as well as correct states for the new and the old value are marked with a '0' as an error free output. The rest is marked with a '1' which indicates an error.

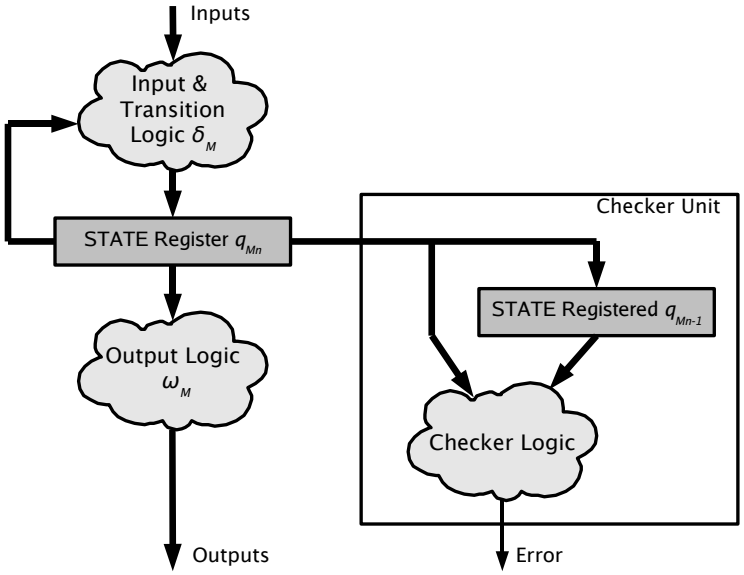


Figure 4.19: A finite state machine (FSM) is shown with a state register, transition and output logic. The checker unit on the right side is able to check the correctness of states as well as the correctness of state transitions. This can be done by registering the state register inside the checker unit to monitor the state change. If the checker logic detects an error, the corresponding signal goes high.

For checking the transitions of counters, the same technique can be used. However, we can see here that a lookup table implementation is very resource consuming. Implementing this function with a redundant counter saves more resources.

Implementation Issues

Implementing the checker logic with unoptimized lookup tables requires a very large resource overhead for larger state register width. For an n-bit state register, the maximal number of lookup table entries n_E for checking only the correct state is:

$$n_E = 2^n \tag{4.2}$$

For checking the correct states and the transitions the maximal number of required lookup table entries is:

$$n_E = 2^{2n} \tag{4.3}$$

For checking an 8-bit state register, we thus require at least 256 lookup table entries (1-bit correct/not correct) for checking the state and 65536 entries for checking the states and the transitions for correctness. This overhead is quite huge. But depending on the number of available used states there is also a large optimization potential. If we look at a *1-hot encoding*, we can check the state if we add all bits from the state register together and check if the result is one. This can be done for the 8-bit state register with a extremely low overhead than with a lookup table with 256 entries. Furthermore, the number of valid transitions is usually very low in contrast to all possible (valid + invalid) transitions. This has also a high optimization potential to bring the resource overhead down. Fortunately, synthesis tools are excellently suitable for optimizing this combinatorial problem.

One problem of the implementation is that the added checker unit represents redundant logic according to the synthesis tools which is usually removed due to area optimization. Also, the kind of state encoding as well as the proper encoding of each state of the FSM is usually decided by the synthesis tool. To handle these problems, the checker logic and the functional IP core shall be synthesized separately.

First, the IP core is synthesized without checker units. The different control paths of the IP core which should be checked can now be chosen and the corresponding state encoding scheme as well as the proper state coding can be taken from the synthesis report. Furthermore, a specification of the correct state transitions for the FSMs and/or the valid range for counters as well as the counter type (up, down, up/down) for generation of the checker unit is needed. For each checked control path a separate checker unit is instantiated. These checker units can now be optimized and translated into a netlist by a synthesis tool. Finally, in the IP core netlist, we must insert black box instances which ingest later the checker units and connect the state or counter

register to these black boxes as well as the error signal to an IP core output. The IP core and the checker units will be integrated in the following implementation steps.

In summary, this method and architecture can check the state and the state transitions of FSMs and counters embedded in general IP cores. Only errors which result in an erroneous state or take an erroneous transition can be detected. The robustness of an FSM can be increased by using a state encoding, e.g., 1-hot encoding, where errors lead to an erroneous state in most cases. The overhead depends on the register bit width as well as on the encoding type. By using different checking domains with different configurations (only checking states or states and transitions), checking the control paths in hardware IPs is modular and widely parametrical. Furthermore, a design flow to implement these methods was shown.

4.4.9 Fault Coverage

In the previous sections, the building blocks of an architecture which is able to check and correct the control flow of CPU programs for RISC processors were introduced. The cadre of this architecture is based on one of the two alternative basic modules for the CF and CFI method (see Section 4.4.1). These methods can be extended by the return stack and/or the other indirect checking architectures shown in Section 4.4.2 to also support the checking of indirect jumps as well as methods to support and check interrupts and traps (see Section 4.4.3).

Obviously, which different types of faults can be detected depends heavily on the proposed error detection method. *Fault coverage* refers to the percentage of faults which can be detected from a defined set of faults. With the instruction integrity checker (see Section 4.4.5) and the conditional branch checker extensions (see Section 4.4.4), we are able to detect more errors and faults than using the basic methods. Finally, the re-execution extension (see Section 4.4.6) gives us the possibility to correct errors which lead, if uncorrected, to an erroneous control flow.

The basic architecture of the CFI and CF method described in Section 4.4.1 is able to detect the following types of faults:

- All permanent and transient faults in instruction memory, memory bus, memory controller, instruction cache instruction register, and logic between different pipelined instruction registers, if the instruction affects the control flow.

- All permanent and transient faults which lead to errors in the program counter register, the logic between different pipelined program counter registers as well as the logic of the calculation of the next program counter.

- Security flaws which affect the control flow (e.g., stack and heap smashing attacks) with the return stack extension.

- Design faults in hardware which can be detected through diversity of the processor core and the checker unit, and in software caused by diversity between the compiled program and the checker unit entries (e.g., unauthorized software update).

Using the CF-method with the additional instruction integrity checker extension presented in Section 4.4.5, we can further detect the following types of faults besides those mentioned above:

- All permanent and transient faults leading to errors in the instruction memory, memory bus, memory controller, instruction cache instruction register, and logic between different pipelined instruction registers which appear before the pipeline step where the instruction integrity checker is instantiated.

Furthermore, using the conditional branch checker extensions (see Section 4.4.4), we are able to detect the following faults besides the previously mentioned faults above:

- All permanent and transient faults which lead to a false decoding of branch instructions which results in an erroneous decision of taking or not taking a branch.

The general IP checker method is able to detect faults in FSMs and counters which lead to invalid states or to erroneous, unspecified transitions. Note that the timing behavior of the transition and the conditions are not checked.

4.4.10 Overhead Discussion

To evaluate the control flow checking architectures, it is necessary to consider the required overhead. In this section, mainly the area and execution time overheads are in focus. The memory overhead for the different basic architectures were already discussed in Section 4.4.1. To have a technology independent area overhead discussion, we decided to use primitive components (see Section 1.2.6) to describe the overhead.

Overhead of the Basic Architectures

The overhead in terms of primitive components of the basis architecture of the CFI method (see Section 4.4.1) is:

- 1×30-bit adder,
- $1 \times n_{cupc}$-bit adder,
- 3×30-bit comparator,

- 11 flip-flops,

where n_{cupc} denotes to the CUPC width.

Additionally, three block memories and some resources for the control logic which cannot be measured in terms of primitive components are required. Also, the bit width of the adders and the flip-flops can alter slightly due to different memory sizes. An important hint of complexity is the number of checker memory entries which is dependent on the *index* and therefore the *CUPC* width (n_{cupc}) to address the memories. Furthermore, the *CUPC* width also has impact on the word width of the *ctrlRam*. The number of necessary checker memory entries depends on the checked program and the number of checked functions. For both methods, the number of necessary checker entries for the SPEC CINT2000 benchmark programs, if all functions are checked, is shown in Table 4.2 in Section 4.3.2.

The area overhead of the basic architecture using the CF method (see Section 4.4.1) is very similar to the values above. However, due to more complex control logic (two accesses to the RAM), the area overhead of the CF method is always higher than the overhead of the CFI method.

The resource requirements in terms of primitive components of the CF method are:

- 2×30-bit adder,
- $2 \times n_{cupc}$-bit adder,
- 3×30-bit comparators,
- 44 flip-flops.

Additionally, two local memories and additional resources for the control logic are required. Alike the CFI method, the bit width of the adders and the number of flip-flops can vary slightly due to different memory sizes.

The memory overhead of the CF method has already been discussed in Section 4.3.2. In summary, the CF method usually requires fewer memory resources for the same program than the CFI method.

If the program code contains basic blocks consisting of only one instruction, there might be an extra CPU time penalty. Usually, the processors stall in many wait cycles due to cache misses or pipeline stalls. However, an additional CPU wait cycle will only be produced for a basic block which consists of only one instruction and if no other wait cycles exist, which are not caused by the checker unit. Hence, it is not even sure that a basic block with only one instruction will actually cause an additional wait cycle if the CF method is used.

We have introduced several extensions for both basic methods of control flow checking. The overhead of these extensions is discussed next.

Overhead of the Return Stack

Returns from subroutine can be verified by introducing an additional hardware stack. Upon a call (direct or indirect), the return address may be stored on the stack. When the return instruction is executed, the target address can be verified.

The caused memory overhead depends on the depth of the stack. To store an instruction address, we need 30-bits for a 32-bit processor. The last 2 bits address the individual bytes inside an instruction and do not need to be stored on the stack. Additionally, we must store the corresponding CUPC value to denote the next checking point in case of a return instruction. Using a 32-bit processor, the memory overhead is therefore (30 bit+n_{cupc}) $\times$ d_{stack}, where d_{stack} is the stack depth.

Also, the return stack causes additional area overhead for control logic which depends on the implementation.

Overhead of Indirect Jump Extensions

The memory overhead of the extensions for checking indirect jumps as introduced in Section 4.4.2 depends on the number of the indirect jump memory (*iJmpRam*) entries and the width of the *CUPC* (checker unit program counter) register n_{cupc}. For each indirect jump which we would like to check in the code, we need one memory row in both methods. The total memory overhead for the first method for a 32-bit processor is therefore in bits:

$$(30\ bit + n_{cupc}) \times n_T \tag{4.4}$$

where n_T denotes to the number of all indirect jump targets.

For the second method in Section 4.4.2, the total memory overhead in bits is:

$$((30\ bit + n_{cupc}) \times D_{max} + n_{cupc}) \times n_{iJ} \tag{4.5}$$

where n_{iJ} denotes to the number of all indirect jumps.

The area overhead in terms of primitive components for the first method is therefore:

- min. $1 \times$ 30-bit comparator and
- logic for searching the indirect jump (e.g., bisection method).

The area overhead for the second method is:

- $D_{max} \times$ 30-bit comparators,
- $1 \times n_{cupc}$-bit D_{max} to 1 multiplexer, and

- control logic.

The execution time overhead for the first methods depends on the search algorithm. Using a bisection method, we need $n+1$ clock cycles for searching an indirect jump target among 2^n entries. During this time, the processor must be stalled if we want to have the possibility to re-execute an indirect jump if an error has occurred.

In the second method, we are able to check D_{max} targets during one single clock cycle. If the indirect jump has more then D_{max} possible targets, we produce an execution time overhead. The overhead in terms of clock cycles can be determined as

$$\left\lfloor \frac{n_t}{D_{max}} \right\rfloor, \tag{4.6}$$

where n_t is the number of possible targets of an indirect jump.

Overhead of the Interrupt and Trap Extension

The interrupt and trap extension (see Section 4.4.3) enhances the basic method to support interrupts and traps. The easiest way is to evaluate signals from the processor pipeline which indicate if an interrupt or a trap occurs or if the interrupt or traps service routine is left to continue the execution of the checked function (return from interrupt or trap). In this case, we only need some additional control logic to evaluate these signals to deactivate or activate the checker unit.

If also the jump to and from the service routine should be checked, we need a register where the address of the service routine is stored. Also a comparator is needed which identifies if a trap occurs. The back-jump can be verified using the return stack. Here, the trap signaling from the processor pipeline is not obligatory, but both techniques can be combined. The additional area overhead is therefore:

- 1×30-bit register,
- 1×30-bit comparator,
- control logic.

If the checker unit should check more than one thread in a multi threading operation system, we need additional memory to store the context of the checker unit for each thread. This memory space can be on the external memory or on chip local memory. For each thread, the context of the checker unit consists of the current CUPC, the current program counter, and the whole return stack.

Overhead of Checking Conditional Branch Extension

The area overhead of checking the conditional of direct branch instructions (see Section 4.4.4) consists of the redundant instruction decoder and a one bit multiplexer (see Figure 4.15 in Section 4.4.4). The overhead of the decoder depends highly on the used processor architecture and implementation.

Overhead of the Instruction Integrity Checker

The overhead of the instruction integrity checker extension (see Section 4.4.5) depends mainly on the bit width of the used CRC values. The CRC width is important for the robustness of the error detection against multiple bit errors. On the other hand, a large CRC value causes a higher area and memory overhead.

The area overhead in terms of primitive components is:

- $1 \times n_{crc}$ comparator and
- n_{crc} flip-flops.

Additionally, block ram and some logic for calculating the CRC and control logic is needed. The memory overhead evaluates to:

$$n_{crc} \times 2^{n_{cupc}} \tag{4.7}$$

This checker unit enhancement causes no further CPU time overhead.

Overhead of the Re-Execution Extension

With the re-execution extension (see Section 4.4.6), we are able to correct a corrupt control flow. To re-execute an erroneous instruction, we need the start point of re-execution (*CPCREEXPC*) as well as the number of instructions (*CPCREXCODE*) which must be annulled to prevent them from execution. The area overhead in terms of primitive components inside the checker unit to calculate both values is:

- $1 \times$ 30-bit adder (increment),
- $1 \times$ 30-bit register (hist register),
- $1 \times$ 30-bit 2 to 1 multiplexer, and
- control logic.

Furthermore, we need some additional logic in the processor pipeline to start the re-execution from the *CPCREEXPC* and also to annul the erroneous instruction as well as their following instructions.

Summarized Overhead Functions for Embedded Processor Control Flow Checking

In this section, all overhead functions of the introduced control flow checking architectures are summarized in one table (see Table 4.3). Here, n_{cupc} denotes the CUPC bit width. The CFI method can handle $2^{n_{cupc}}$ control flow instructions and the CF method can handle $2^{n_{cupc}}$ basic blocks. d_{stack} denotes the return stack depth, n_T denotes the number of different targets for all indirect jumps, D_{max} is number of different targets for one indirect jump for the indirect jump method 2 which can be checked within one clock cycle, n_{iJ} denotes to the number of all indirect jumps, n_t is the number of possible targets of an indirect jump, and finally n_{crc} is the CRC bit width. However, in Table 4.3 the overhead for control and other logic is omitted.

	Area Overhead				**Memory Overhead**	**CPU time Overhead**
Architecture	adders	comparators	flip-flops	multiplexers	bits	cycles
CFI Method	1×30-bit, $1 \times n_{cupc}$-bit	3×30-bit	$5 + n_{cupc}$		$(65 + n_{cupc}) \times 2^{n_{cupc}}$	none
CF Method	2×30-bit, $2 \times n_{cupc}$-bit	3×30-bit	$5 + n_{cupc} + 30$		$(35 + n_{cupc}) \times 2^{n_{cupc}}$	none
Return Stack					$(30 + n_{cupc}) \times d_{stack}$	none
Ind. Jump Method 1		min. 1×30-bit			$(30 + n_{cupc}) \times n_T$	$n+1$ for 2^n entries
Ind. Jump Method 2		$D_{max} \times$ 30-bit		$1 \times$ n_{cupc}-bit D_{max} to 1	$((30 + n_{cupc}) \times D_{max} + n_{cupc}) \times n_{iJ}$	$\left\lfloor \frac{n_t}{D_{max}} \right\rfloor$
Interrupt/ Traps		1×30-bit	1×30		none	none
Conditional Branch Ch.				1×2 to 1	none	none
Inst. Integrity Checker		$1 \times$ n_{crc}-bit	n_{crc}		$n_{crc} \times 2^{n_{cupc}}$	none
Re-Execution Architecture	1×30-bit		1×30	1×30-bit 2 to 1	none	depends on proc.

Table 4.3: Comparison of different overheads for each control flow checking method and its extensions for a 32-bit RISC processor.

In summary, CFI and CF are two alternative methods and the presented extensions can be combined with these methods and further extensions, resulting in a modular framework for control flow checking. One exception is the instruction integrity checker which can only be combined with the CF method.

4.5 Prototypical Implementation

In this section, the proposed control flow checker architectures for embedded processors are integrated into a given *Leon3 processor system* from Gaisler Research [Gaib] using the *SPARC V8* [SPA] instruction set. The two basic architectures for control flow checking (CFI and CF architecture) with the return stack, interrupt and trap handling, and the re-execution extension are implemented on an FPGA prototype.

First, the control flow instructions of the SPARC V8 instruction set are presented and some mannerism of the SPARC architecture are shown. Next, an overview of the Leon3 processor system is given. After the presentation of the processor platform, the integration of the control flow architectures and their extensions is shown. To analyze the compiled program code, we need an analyzer tool which generates the proper entries for the checker memories. In the next section, the integration and interaction with the data path protection technique developed by the project partner *TU Munich* (TUM) [BZS+06] is shown. Finally, for better understanding, some examples and experimental results from the simulation and implementation on an FPGA prototype are given.

4.5.1 The SPARC V8 Instruction Set Architecture

The *SPARC* (Scalable Processor Architecture) V8 is a 32-bit *RISC* (Reduced Instruction Set Complexity) *instruction set architecture* (ISA). Like other 32-bit RISC ISAs, each of the 72 different basic instructions are encoded in a single 32-bit word which means that all instructions have the same word length. The SPARC V8 architecture consists of an *integer unit* (IU) and an optional *floating point unit* (FPU) or a *co-processor* (CP). The FPU and CP have their own registers and are accessed by special floating point and co-processor instructions. Beside the 32-bit SPARC V8 ISA, also the 64-bit SPARC V9 ISA exists.

Instruction Overview

The instructions of the SPARC V8 ISA are encoded in 32-bit words. The format of each 32-bit word is depending on the type of instruction. In general, there are six different categories of instructions:

- **Data Transfer (Load/Store):** These instructions are able to access the memory. 16-bit, 32-bit, and 64-bit accesses are supported.
- **Arithmetic/Logical/Shift:** These instructions initiate an arithmetic, logical or shift operation.

- **Control Flow:** All instructions which affect the control flow. This category is described in detail later.
- **Read/Write Control Register:** Using these instructions, special control registers can be accessed.
- **Floating Point Operate:** This kind of instructions initiate a floating point operation, if a FPU is present. If not, a trap is generated, which gives the program or operating system the possibility to emulate floating point instructions in software.
- **Co-Processor Operate:** These instructions initiate a co-processor operation.

For a detailed description of each type of instruction, see the SPARC V8 manual [SPA].

Control Flow Instructions

The SPARC ISA consists of control flow instructions for direct jump and branches as well as indirect jumps. The target addresses of direct branches and jumps are calculated in a relative manner, i.e., a value which is stored inside the instruction is added to the current program counter in order to calculate the target address. The targets of the indirect jumps are register indirect, which means that the target is the program counter added with the result of an operation of two register values, or an operation with a register and a constant value.

The SPARC ISA uses a *delayed control flow* concept for almost all control flow instructions (except conditional traps). This means that the instruction following directly a control flow instruction is executed, and then, the control flow transfer is initiated. The effect of the control flow instruction is therefore delayed by one instruction. The instruction after a CFI is called delay instruction. Furthermore, all CFIs which use this delayed concept have a flag (annul flag or a) encoded into the instruction which indicates if the delay instruction should be executed or not. If the annul bit is not set ($a = 0$), the delayed instruction is executed. Whereas, if the annul bit is set ($a = 1$), the delay instruction is prevented from execution. This has the same effect as executing a non operation instruction (NOP). However, if the CFI is a conditional branch and the branch is taken, the delay instruction is always executed, independent of the annul flag. Figure 4.20 shows an example of this delayed control flow concept.

The different types of control flow instructions of the SPARC V8 architecture are described in the following:

Call and Link (CALL) The call and link instruction is a direct jump to an address which is encoded inside the instruction relatively to the current program counter.

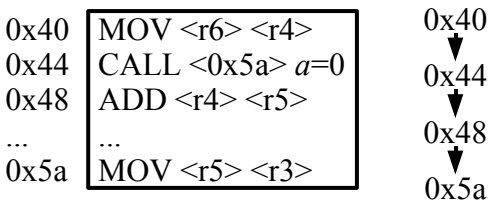


Figure 4.20: On the left side, a program memory snippet with a call instruction (CALL) where the annul flag is not set ($a = 0$) is depicted. If this program code is executed (right side), the delayed instruction ADD after the call CFI is executed before the control flow is transferred to the call target on address 0x5a.

The instruction has no annul flag, so that the following delay instruction is always executed. Furthermore, the current program counter is copied into a special register (r[15]). This instruction is usually used to call a subroutine.

Jump and Link (JMPL) The jump and link instruction is the common indirect jump instruction for the SPARC architecture. This unconditional jump exists in two versions. In the first version, the target address is calculated by an addition of two register values. The addresses of the registers are part of the instruction. In the second version, the target address is calculated by an addition of a register value and a constant value, stored inside the instruction. The current program counter is stored in a register which can be selected by the instruction. Register indirect calls and returns from subroutine are in fact also special jump and link instructions. For example, the RET (return from subroutine) and RETL (return from leaf subroutine) assembler instructions are only aliases for special JMPL instructions.

Branch on Integer Condition Codes (Bicc) The branch on integer condition codes instruction family consists of different branch instructions with different conditions which evaluate the *integer condition codes* (icc). The icc is a special register which is updated by an arithmetic operation. There exist four different icc flags: *n* is set if the result is negative; *z* is set if the result is zero; *v* is set if there was an overflow at the operation; *c* is set if the carry bit is set. The condition encoded in the branch instruction evaluates these bits and decides if the branch is taken or not. Some example Bicc instructions are: BE (branch on equal), BG (branch on greater), BLEU (branch on less or equal unsigned). There exist also two unconditional branch instructions: BA (branch always) and BN (branch never). If the branch is taken, first the delayed instruction is executed, and then the control flow is transferred to a target address which is stored inside the instruction relatively to the current program counter. The Bicc instructions have further an annul (*a*) bit, which only has effect if the branch is

not taken. One exception, however, is the BA instruction which also evaluates this bit to decide if the delay instruction is executed or not.

Branch on Floating Point Condition Codes (FBfcc) The branch on floating point condition codes instruction family is very similar to the Bicc instructions. The difference is that the floating point condition codes are evaluated instead of the integer condition codes. The floating point condition codes result from a floating point operation.

Branch on Co-Processor Condition Codes (CBccc) Like the FBfcc instructions, these instructions are very similar to the Bicc instructions. Here, the co-processor condition codes are evaluated.

Trap on Integer Condition Codes (Ticc) The trap on integer condition codes instruction can be used for programmed traps. Like the Bicc instruction, the integer condition codes are evaluated with the conditions programmed inside the instruction. Depending of this evaluation, a trap is generated or not. The conditions are the same as for the Bicc instructions. The target of this CFI is one of the 128 software trap routines. Which trap routine is called depends either on the addition of two register values or on an addition of one register value with a constant which is included in the instruction. More about traps is discussed later in this section. The Ticc instruction transfers the control flow immediately without executing a delay instruction. Like the call instruction, the current program counter is stored in a register, if the trap is taken.

Return from Trap (RETT) The return from trap instruction is an indirect jump which is used for the back-jump from the trap routine to the instruction where the trap has occurred. The format and behavior of this instruction is very similar to the JMPL instruction. However, this instruction further leaves the supervisor mode and enables traps.

Registers

In the SPARC V8 ISA, general purpose and control and status registers are available. All registers have a word length of 32-bit. The general purpose registers, denoted *r*, consist of 8 global registers and a *register window* consisting of 24 registers. The global registers are always available, whereas the register window is dynamic.

The register window is divided into three groups: 8 *in* registers, 8 *local* registers, and 8 *out* registers. The register window is only a visible part of a greater register file. With special instructions, the register window can be shifted, so that another part of the register file is visible. With the SAVE instruction, the window is shifted 16 registers to the right, whereas on a RESTORE instruction, the window is shifted

16 registers to the left. The register window position is given by the *current window pointer* (CWP). Hence, the register window has 24 registers, and the different window positions overlap (see Figure 4.21). The *in* and *out* registers are shared with the neighborhood window positions. After a SAVE instruction, the current *out* registers are the new *in* register, whereas at a RESTORE instruction, the current *in* register are the new *out* registers. The number of different window positions is depending on the implementation. Up to 32 window position are supported by the SPARC V8 ISA which corresponds to a total number of 520 general purpose registers.

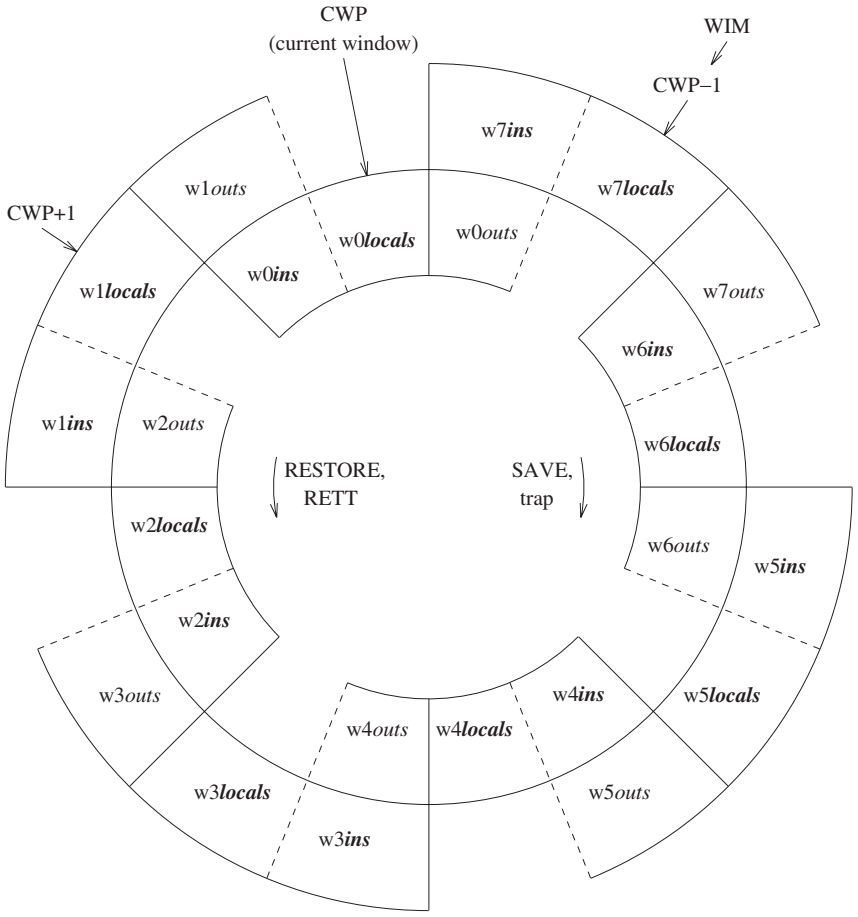


Figure 4.21: The SPARC register window concept. At a given time, only a subset of the registers are visible (*ins, locals, outs*). With the instructions SAVE and RESTORE, the visible area of the register file can be shifted [SPA].

The windowed register concept is mostly used by calls and returns from subroutine. For each nested subroutine, a window position is assigned. The input and output variables, like parameters, the program counter, or return values can be handled effectively by using the *in* and *out* registers. This extremely reduces the amount of

memory accesses for calls and returns in comparison to, e.g., the x86 architecture, where all these variables must be copied into the memory stack on a call and need to be restored from memory on a return. However, if the windowed registers overflow, a trap is triggered which copies the content into the memory and enables an unlimited nesting depth.

A further advantage of this concept is the improved robustness against buffer overflows attacks, because of the return address is mostly stored into registers which can only be addressed directly. This enormously hinders code injection attacks and thus improves the security of the system. A study about exploits for the SPARC architecture [Sch08] shows that stack smashing with buffer overflows can only be exploited with a high effort to execute malicious code.

Traps

A trap of the SPARC ISA is a *vectored transfer* of control to *trap service routines*. The behavior is like an unexpected subroutine call. A vectored transfer of control means that a *trap table* is used which has a certain base address which can be altered with a specific control register. The target of a trap is therefore the first 20 bits of the trap table base address, following 8 bits to encode the different types of traps (*tt* field) and 4 zero bits. The trap table has also 256 (2^8) entries whereas every entry has space for a maximum of 4 instructions ($2^4 = 16/4$ bytes per instruction). In this space usually the call to the service routine is initiated which is finished by a RETT instruction. This means that in case of a trap the control flow jumps directly to the correct trap table entry which handles this type of trap.

Traps can be generated by exceptions during the execution, by external interrupt events, or by software using the *Ticc* instruction. Each type of exception or interrupt uses a special *tt* encoding to jump to the correct service routine. For software traps, the *tt* field is encoded in the Ticc instruction which enables us to call different service routines.

4.5.2 An Overview of the Leon3 Processor Architecture

The *Leon3* processor from *Gaisler Research* [Gaib] is an implementation of the 32-bit SPARC V8 architecture. The processor core is included in the *GRLIB* library with many other IP cores which are suitable to build a complete SoC. The communication between the different IP cores inherits an *advanced high performance bus* (AHB) which is defined in the *AMBA* (advanced micro controller bus architecture) specification [ARM99]. The complete GRLIB, including the highly configurable Leon3 core and the AHB, is an open source VHDL implementation and can be downloaded from the Gaisler website. After the configuration of the SoC, it is possible to synthesize and implement the system for different ASIC or FPGA target technologies.

The Leon3 core consists of an integer unit (IU3), data and instruction caches, the register file, a debug port, the AMBA bus interface, and an optional FPU. The integer unit is implemented with a 7 stage pipeline including data and instruction cache interfaces as well as the register file interface. The pipeline consists of the following stages [Gaib] (see also Figure 4.22):

- **FE (Instruction Fetch):** In this stage, the instruction address is forwarded to the instruction cache and the instruction is fetched from the cache.
- **DE (Instruction Decode):** In this stage, the instructions are decoded and the target addresses are calculated for direct CFIs (CALL and Bicc).
- **RA (Register Access):** The operands are read from the register file or from internal data bypasses.
- **EX (Execute):** The ALU performs arithmetic, logical, or shift operations. Furthermore, the target addresses for indirect CFIs (JMPL and RETT) are calculated.
- **ME (Memory):** The memory over the data cache is accessed. For write operations, the data calculated by the EX stage is written to the data cache.
- **XC (Exception):** Traps and interrupts are resolved. For memory read operation, the data inquired from the ME stage are aligned.
- **WR (Write Back):** Any result from the EX or XC stage is written back to the register file.

Before the fetch stage, the next instruction address is calculated (see upper left corner in Figure 4.22). The next address can be an increment of the current address of the fetch stage, if no CFI is executed. The relative target address of a direct CFI is decoded in the DE stage and added to the current DE stage program counter before the result is fed back to the address calculation multiplexer. The target address of the indirect JMPL or RETT CFI is calculated in the EX stage. The target addresses of traps are available in the WE stage. Both target addresses are also fed back to the address calculation multiplexer before entering the pipeline. Furthermore, the pipeline can be stalled, e.g., for cache misses or register interlocks. In this case, no new address is fed into the pipeline for several clock cycles.

4.5.3 Integration of the Control Flow Checker Architecture

We prototyped and analyzed the architectures implementing the CFI and CF method (see Section 4.4.1) for the open source SPARC CPU *Leon3* from Gaisler Research [Gaib] on a *Virtex-4* FPGA from Xilinx. The checker can monitor direct branches,

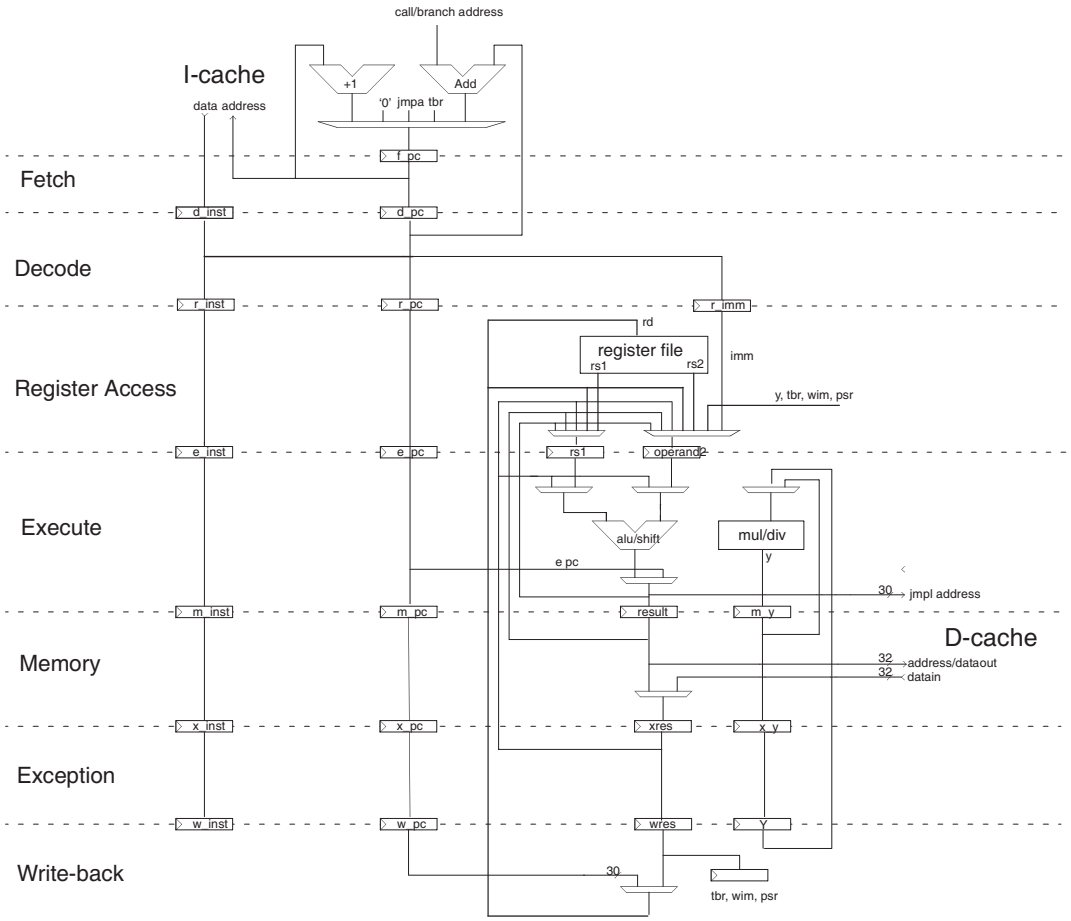


Figure 4.22: The Leon3 integer unit pipeline. On the left side, the control paths are shown with the calculation of the next instruction address in the upper left corner. On the right side, the data paths with the interface to the data cache are shown [Gaib].

jumps and calls as well as indirect returns and also has the ability to re-execute a corrupted jump or branch instruction by fetching it again from the memory. Other features of the checker are the support of the activate and deactivate procedures described in Section 4.4.1. The complete implementation for control flow checking of the Leon3 core consists of the proposed architectures of checking of direct jumps and branches (Section 4.4.1), the return stack (Section 4.4.2) and the repair mechanism (Section 4.4.6). To minimize the resource overhead, some features can be disabled (see Section 4.5.8). Indirect jumps which are not returns, are not supported so far, but many application programs or routines in embedded systems have none of these instructions, or should avoid their use.

Implementation Overview

The implementation consists of a control path checker module (*cpc_basicReEx*) and the logic for re-execution in the integer pipeline of the Leon3 processor (*iu3*) (see Figure 4.23). The different modular architectures of control flow checking are implemented into this checker module. The interface of this unit is kept generic, so the unit can in principle be attached to any embedded RISC processor. For the Leon3 core, the checker module is attached to the integer pipeline.

Figure 4.23 shows the interface between the integer unit (*iu3*) of the Leon3 core and the checker module. Also, an interface to the on-chip AMBA bus is shown which enables the access to the checker memories from the processor and debugger side. Furthermore, Figure 4.24 shows the connection of the checker module interface to the integer pipeline, which is part of the *iu3* module.

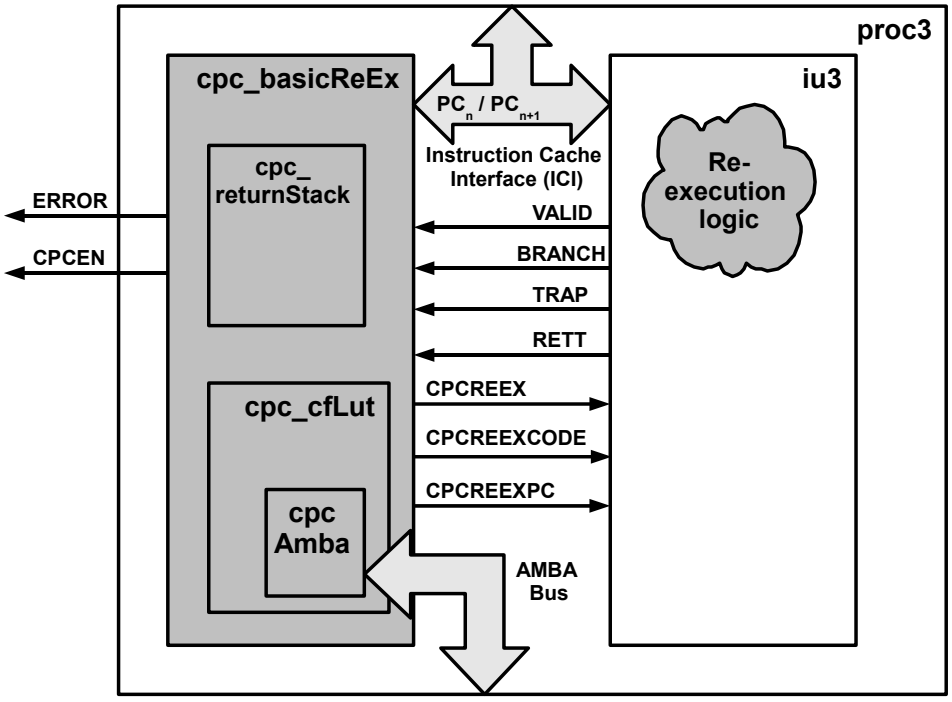


Figure 4.23: An overview of the control flow checker unit with the interface to the Leon3 integer unit (iu3). Each module (box) represents a VHDL module. All additional modules for the control flow checker are shown in gray.

The current program counter value (PC_n) and the next program counter value (PC_{n+1}) are taken from the first pipeline stages of the *Leon3* core (see Figure 4.24). The current program counter for the checker is the program counter in the decode

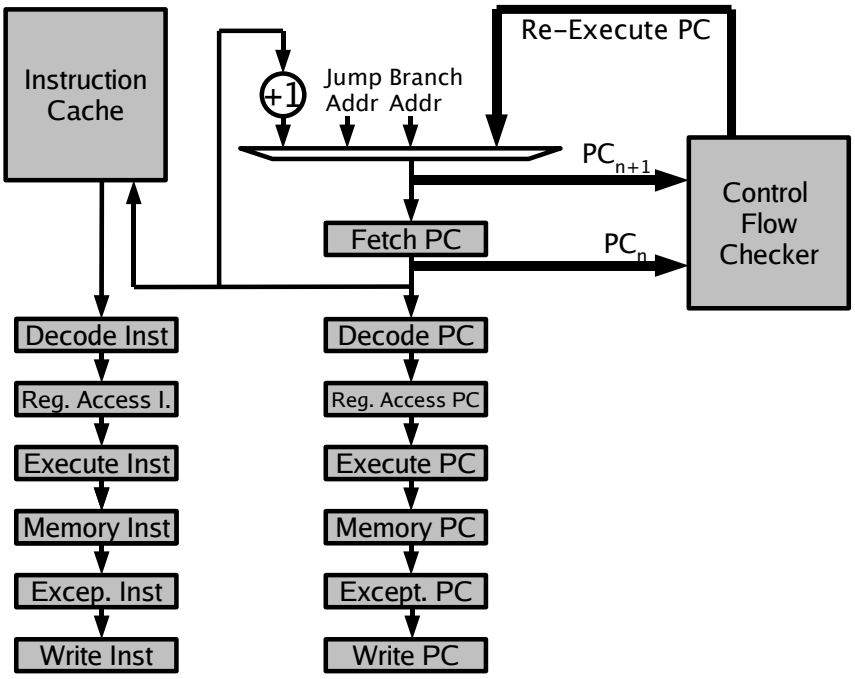


Figure 4.24: The checker unit is placed between the first two pipeline stages of the *Leon3* core [Gaib]. All bold lines denote new signal paths needed for monitoring and re-execution of jump and branch instructions.

pipeline step and the next program counter is the program counter of the fetch pipeline step. For re-executing a jump or branch, the re-execution program counter from the checker unit is fed to the program counter generation (a step prior to the fetch step), and the erroneous instructions are annulled, so the incorrect instructions are not executed and no registers or memories are written.

Checker Module (cpc_basicReEx)

The general control flow checker (*cpc_basicReEx*) consists of the control flow lookup table (*cfLut*), the return stack (*returnStack*) and the checker logic. Additionally, an AMBA slave interface is included.

The interface of the checker module is depicted in Figure 4.23. The *VALID* signal should be only active if the *CURRPC* and *NEXTPC* values are valid. For each executed instruction, the *VALID* signal should be high for exactly one clock cycle. *CURRPC* is the current program counter and *NEXTPC* the next program counter. Both signals are taken from the instruction cache interface. The *NEXTPC* is the program counter after the fetch stage; *CURRPC* denotes the program counter before the

fetch stage. *TRAP* signals the control path checker that a trap has occurred. The *RETT* (return from trap) signals that the trap routine has finished.

If the checker unit detects an error in the control flow, the *ERROR* signal is asserted. The *CPCEN* signal is active when the checker unit is active. For the re-execution procedure, we need the following signals:

- *CPCREX* signals that the re-execution is in progress.
- *CPCREXCODE* signals the type of erroneous instruction. This indicates how many instructions must be invalidated.
- *CPCREEXPC* is the program counter where the re-execution starts.

This general checker module is able to implement either the CFI or the CF method. The methods differ in the way of storing the information from the compiled code into memory structures inside the checker which is implemented into the *cfLUT* module. The top level module of the checker module and several extensions, such as the activate/deactivate feature, the return stack, or the re-execution possibility are the same and can be used for both methods. For switching the direct control flow method, only the *cfLUT* module must be exchanged.

Inside the control flow checker module, four comparators exist (see in Figure 4.25 *a-d*). Comparator *a* checks if the next program counter is incremented by one in the case if no control flow instruction is executed or a branch is not taken. If the executed program reaches a control flow transfer point (the end of a basic block in the CF method, or a control flow instruction in the CFI method), the result of comparator *b* is true. In this case, the next program counter can be checked either with comparator *c* in the case of a direct jump (call or taken branch), comparator *d* in case of an indirect return, or comparator *a* in the case of a non-taken branch. The instruction type of the control flow transfer point is encoded in the control flags, which are part of the entry of the control memory (see Figure 4.25 and 4.26). The type of control flow transfer instruction decides which results of the comparators must be true to ensure a correct control flow (see Table 4.4). For the Leon3 architecture, the delay slots of direct jumps and branches must be considered. This can be done by delaying the result of the *b* comparator for one instruction (*VALID* signal). Furthermore, the index of the next control flow transfer point is included in the control memory entry. In the example in Figure 4.26, 9 bits for the next CUPC are used which corresponds to a maximum of 512 checker entries inside these memories. If larger memories are used, the next CUPC value, and therefore the word length of the control memory, must be extended to the corresponding bit length.

Besides the type of instruction flags, there is an activation and a deactivation flag (see Figure 4.26). These flags can be used for the global activation/deactivation of the checker as well as local activation/deactivation if, for example, a function should be excluded from checking.

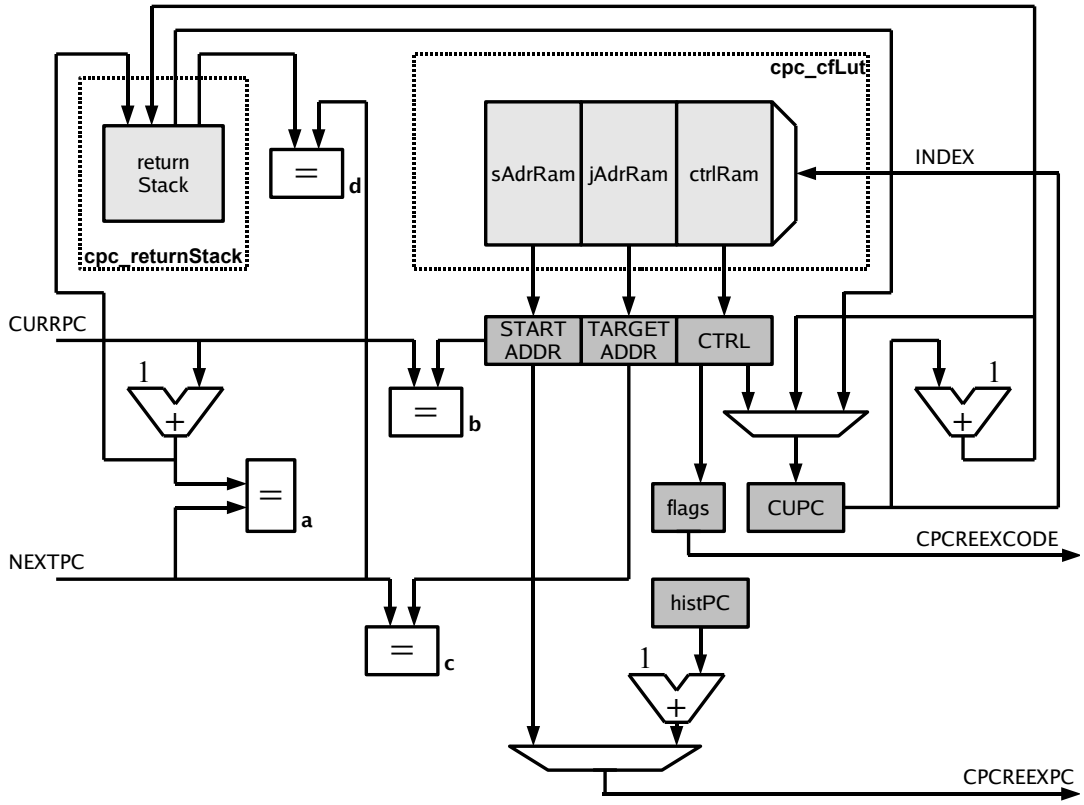


Figure 4.25: The architecture of the general module cpc_basicReEx with comparators and RAMs. The return stack and re-execution extensions are already integrated within the basic architecture. The *cpc_cfLut* shows the architecture for the CFI method.

Instruction	*ctrlRam* flag	Comparators which must be true
non CFI		*a*
branch (**Bicc**)	**B**	*b* and (*a* or *c*)
call (**CALL**)	**C**	*b* and *c*
return (**RET**)	**R**	*b* and *d*
end of BB with non CFI	**E**	*b* and *a*

Table 4.4: Depending on the current instruction, different comparators must be true to signal a correct program flow. Also, the different control memory flags are shown.

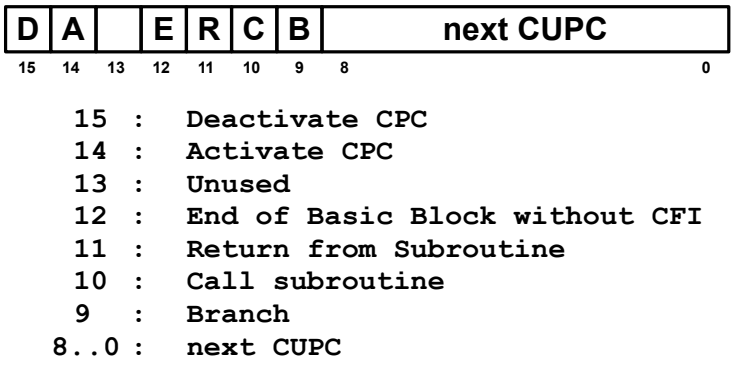


Figure 4.26: Control memory (*ctrlRam*) entry. The flags in the front denote the kind of the control flow transfer point. The last 9 bits represents the index for the next control flow point in case of a call or branch instruction.

If a trap occurs during the execution, which is signaled by the *TRAP* input, the checker unit deactivates itself and the checker will be activated again by leaving the trap routine (*RETT* signal).

Furthermore, logic and registers for the re-execution procedure are included in the top level of the control path checker module. If the re-execution is triggered by a control transfer point, the correct start address is taken as the re-execution start address. If not, the value of a history register is taken which saves the last executed program counter (see Figure 4.25).

Control Flow LUT (cpc_cfLut)

The only difference between the architecture of the CF and CFI method is in the control flow LUT which implements the information of the control flow graph or the control flow instruction graph. For both methods, the *cfLut* has the same interface, but different implementations. The LUT has also an AMBA bus interface to allow for access to the LUT from the processor side.

The interface consists of the following signals: *STARTADDR* is the address of the next control flow transfer point and *TARGETADDR* is the corresponding target address if the instruction is a direct jump or branch. *CTRL* is the control vector with the control flags and the index of the next control flow point if the jump is executed (e.g., call or branch taken), as depicted in Figure 4.26. *INDEX* is the address of the checker memories and the next values are fetched when the *NEXTVALUE* signal is active. We need the *VALID* signal for the CF method, because the CF method needs two clock cycles to fetch or calculate all values from the RAMs.

The implementation of the control flow LUT is depicted for the CFI method in Figure 4.25 and for the CF method in Figure 4.27.

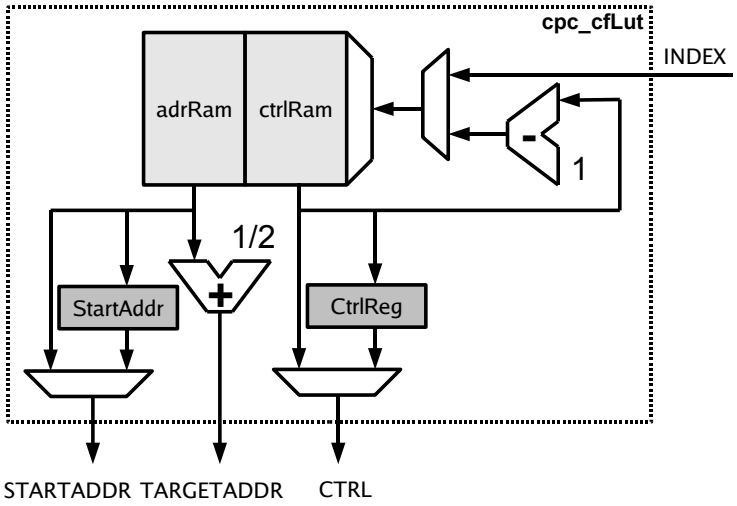


Figure 4.27: The *cpc_cfLut* module for the CF method. Here, we need two times access to the *adrRam* per basic block. The calculation of the *TARGETADDR* depends on the type of the instruction at the end of the previous basic block. If this instruction is a control flow instruction, we have to add two instruction positions to the address, due to the additional delay instruction.

Return Stack (cpc_returnStack)

The return stack saves the current program counter incremented by one (PC_{n+1}) and the CUPC is also incremented by one if a call is executed and the checker unit is active. This is done when the call control flag (C) is high and the current program counter is equal to *STARTADDR* (comparator *b* in Figure 4.25).

If a return instruction is reached (current program counter is equal to *STARTADDR* and the return control flag (R) is set), the entry is popped off the stack. The CUPC is overwritten with the value from the stack and next program counter is checked against the program counter value from the stack (comparator *d* in Figure 4.25). Currently, a stack with 32 entries is implemented.

Modification of the Leon3 Integer Unit (iu3)

Besides the separate control path checker module, the VHDL files implementing the Leon3 integer unit are modified. The Leon3 integer unit needs three additional inputs for the re-execution procedure: *CPCREEX*, *CPCREEXCODE*, and *CPCREEXPC*. *CPCREEX* is active if an error is detected and the re-execution procedure is initiated. In this case, the address of the erroneous instruction *CPCREEXPC* is forwarded to the fetch stage. The result is a re-fetch of the instruction which causes the error.

The *CPCREEXCODE* denotes the kind of erroneous instruction. A non control flow instruction is encoded with “00”, a direct call or branch is encoded with “10” and a return from subroutine is encoded with “01”. If the re-executed instruction is a direct call or branch, the current register window pointer *CWP* from the decode stage must be restored from the access stage so to ensure that the *CWP* has the same value before the erroneous instruction was executed. Also the integer condition codes *icc* must be restored. In case of an erroneous return from subroutine instruction, the *CWP* must be restored from the execute stage.

At last, the erroneous and the following instructions must be invalidated. This can be done by setting the *annul* bit at the corresponding stage. In case of a direct call or branch, the *annul* bit is set for the decode, register access, and execute stage and in case of an indirect return from subroutine, the *annul* bit for decode, register access, execute, memory and exception stage must be set. If the erroneous instruction was not a control flow instruction, only the decode stage must be invalidated.

4.5.4 A Tool for Program Analysis

To prepare an application for control flow checking, the compiled code is analyzed by a program called “AISTool” which decodes the instructions and searches for control flow instructions. This tool is able to generate the content of the checker memories for the CF and the CFI *cfLut* module. The functions of the analyzed program which should be checked (user functions) can be specified in a separate file which is committed to the analyzer program. The program first extracts all functions and searches for the user functions.

For the CFI method, all CFIs of the user functions are extracted and their addresses are stored in the *sAdrRam* initialization file and the destination address, except for the return instruction, is stored in the *jAdrRam* initialization file. For the CF method, the analyzed program is first structured into basic blocks. The basic block end addresses are stored in the *adrRam* initialization file. For both methods, the corresponding initialization file for the control Ram (*ctrlRam*) is generated. If a user function is left by a call to a non user function, the checker will be deactivated and activated again on the back-jump. This is done by setting the corresponding activate (*A*) and deactivate (*D*) flags in the control memory. Furthermore, the call to the first defined user function (usually `main()`) which is the starting point for the checker is searched and at this call, the checker unit will be activated. The index of this call must be ’0’, because after global reset, the value at address ’0’ is loaded from the checker memories. Finally, the memory initialization files and, optionally, a graph (CFG or CFIG) of the analyzed functions in the *Graphviz* `.dot` format [AR] are written. The analyzed application program remains completely unchanged.

The memory initialization files can be used for the synthesis of the checker unit, or for Xilinx FPGAs, the content of the rams can be initialized directly in the bitfile using the Xilinx tool *“data2mem”*. We also implemented an *AMBA* bus interface

for the checker unit to have access to the checker memories from the processor side. Using the bus interface, the memories can also be initialized at runtime over the memory bus or the processor (see Figure 4.28).

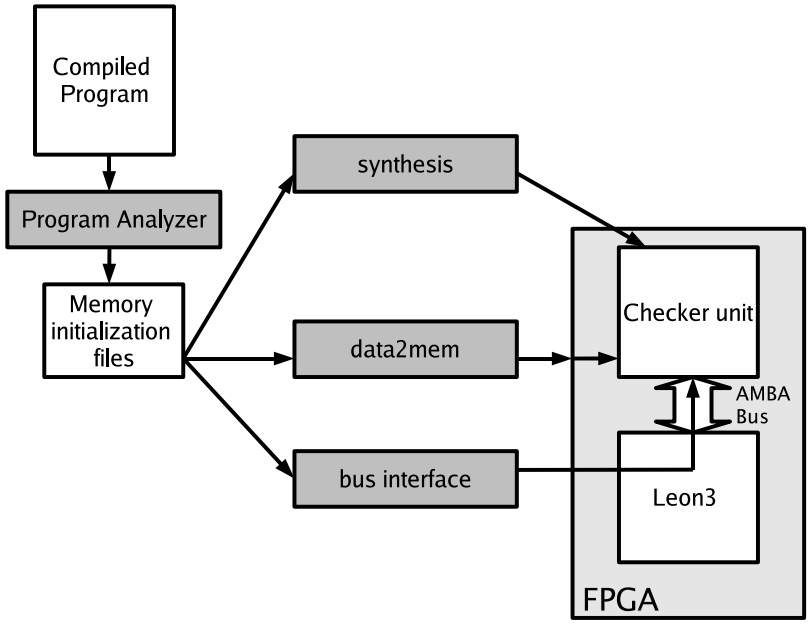


Figure 4.28: From the compiled code, the program analyzer extracts all branches, calls, and return from subroutines and generates the memory initialization files for the checker memories. These memories can be initialized during synthesis, later in the bitfile with the *data2mem* tool, or at runtime using a bus interface.

4.5.5 Interaction between Control Flow Checking and Data Path Protection

The goal of the AIS project partner TUM was to investigate techniques for data path protection where single event effects in the data path can be detected and corrected by means of a micro rollback [BZS+06]. The fault detection is done by using *shadow registers* as introduced by Nicolaidis [Nic99]. In this concept, the original main register and an additional shadow register have the same input, but the shadow register is clocked with a delayed clock. The result is that the input signal is sampled at two different points. The delay time of the shadow register is chosen in a way that the time delay between the two sample points is greater than the largest duration of a single event perturbation (glitch). If a particle impact and therefore a single transient effect occurs, only one of both registers is affected. By comparing the value of both

registers, these effects can be detected. Furthermore, this technique allows also to detect a single event upset which occurs in one of these registers as well as timing errors. To detect errors in the data path, all pipeline registers of the Leon3 integer unit are extended with this second shadow register.

If an error is detected, the latest error-free register values of the pipeline should be restored. This can be done at low latency, by introducing a third category of registers, so-called *history registers* [BZS+06]. A history register samples the main register value with one clock cycle delay. If an error occurs, the history registers carry the latest correct state which can be used for the micro rollback. This is done by restarting the operation from the history registers which are now used as inputs for the following pipeline step. Figure 4.29 shows the replacement of a pipeline register with the error-resilient triple register set. For correcting an error, two additional clock cycles are needed for detecting and correcting the error.

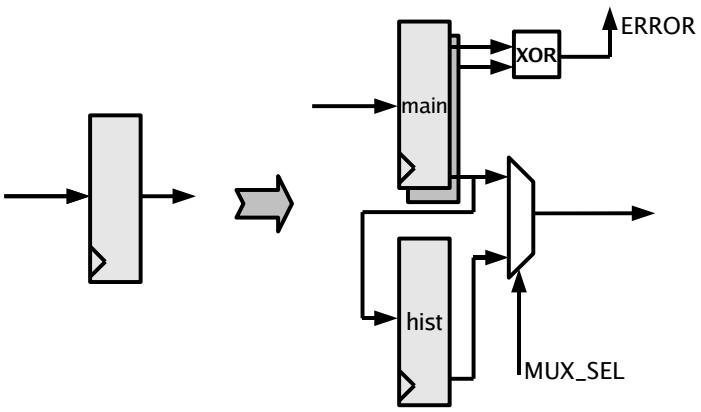


Figure 4.29: This figure shows the replacement of a single pipeline register with an error resilient register set, consisting of the main and shadow register (above) as well as the history register (below). By comparing the main and shadow registers SEEs which manifests in errors can be detected. Using the history register, the latest correct state can be recovered [BZS+06, Koh08].

The data path protection method serves mainly to detect and correct transient single event effects and timing errors which happen at the register pipeline, whereas the control flow checking methods can detect all errors which affect the control flow. To these errors belong also permanent errors as well as errors in cache and memory structures and software buffer overflows. The intersection of the error coverage of both techniques is small, and therefore, a combination of both techniques makes sense and complement each other which leads to a robust control- and data path-protected processor core.

However, both techniques cannot easily be combined without modification, because the error correction of one method affects the error detection of the other method. For example, consider a micro rollback of the data path protection caused by an error. The currently executed instruction is stalled and executed again, which is identified by the control flow checker as a control flow error. It is obvious that the method must communicate its correction action to the complementary method. Here, we cover only the control flow checking method. The question is how the control flow checker deals with a micro rollback of the data path protection.

The easiest way is to deactivate the checker on a micro rollback and activate the checker again after the reconstruction of a correct state. This can be easily done by evaluating a rollback signal from the data path protection control unit. The disadvantage is that during the rollback, the control flow checker unit is deactivated and the control paths of the processor are unprotected.

The second possibility is to check also the control flow of the micro rollback. As mentioned before, the micro rollback consists of two steps. The first step is the detection and freeze step. After successful detection of a data path error, the pipeline is stalled and the data path protection feeds the value of the history register of the fetch step back to the program counter generation step in order to start the micro rollback (see Figure 4.30). From the control flow checker view, the PC_{n+1} input is set back to the PC_{n-1} value. To get this value, we can use the registered value from the PC_n input. Actually, if the control flow re-execute feature is implemented, we already have such a register (*histPC*) that is used for the re-execution program counter value. To check the first rollback step, a new comparator f is introduced, which checks the correct new next program counter value in order to start the rollback [Koh08] (see Figure 4.31).

The next step is the retry step, where the input of all pipeline stages is switched to the history registers and as a result, the state of the data pipeline is rolled back to the state before the error occurs. From the control flow checker view, the last instruction is executed again. The internal state of the control flow checker has also changed, hence, the state of this checker unit must also be rolled back. This can be done by introducing history registers for the registers used inside the checker unit (see Figure 4.31). For checking the control flow of the retried instruction, the values from these history registers must be used. After successfully verifying the rollback, the control flow checking unit is switched back to the normal registers just as the other registers of the data path pipeline.

If an (control path) error occurs during the micro rollback, the control flow checker is able to correct this error by initializing a re-execution. If the error happens in the detection/freeze step, the re-execution starts from the *histPC* register. However, if the error is inside this register, we have a problem which cannot be solved using this architecture. A possible solution is to protect this register with an *error correcting code* (ECC) or provide an additional redundant register. If the error occurs during the second rollback step, the normal re-execution procedure of the control flow checker

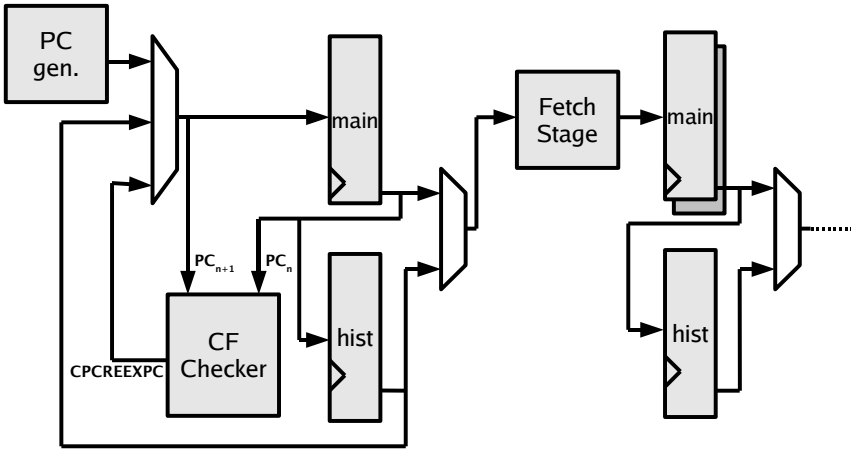


Figure 4.30: The integration of control flow checking and data path protection into the first pipeline stages of the Leon3 processor. The PC_n and the PC_{n+1} signals as well as the $CPCREEXPC$ of the control flow checker and the history and shadow registers of the data path protection are shown. In the first phase of the micro rollback, the PC from the first history register is fed back to the PC generation multiplexer. In the second phase, the history register is taken as input for the following pipeline stage. Here, only the fetch stage is shown. It can also be shown that the data path protection can only detect errors using the shadow register concept from the fetch and the following stage. The program counter generation step is protected by the control flow checker.

unit is used which, depending of the type of erroneous instruction, re-executes one to several instructions.

The standard control flow checking unit cannot check the conditions of branches. The checker allows to take and not take the branch. Using the additional data path protection facilities, the *integer condition codes* (icc) are protected. The final gap for checking also the conditions of branches thus closes the conditional branch checking extension, introduced in Section 4.4.4. Using all these techniques together, conditional branches can be fully checked for correctness.

In summary, the integration of both concepts increases the reliability and also the security (keyword: *buffer overflows*) of the Leon3 processor core significantly.

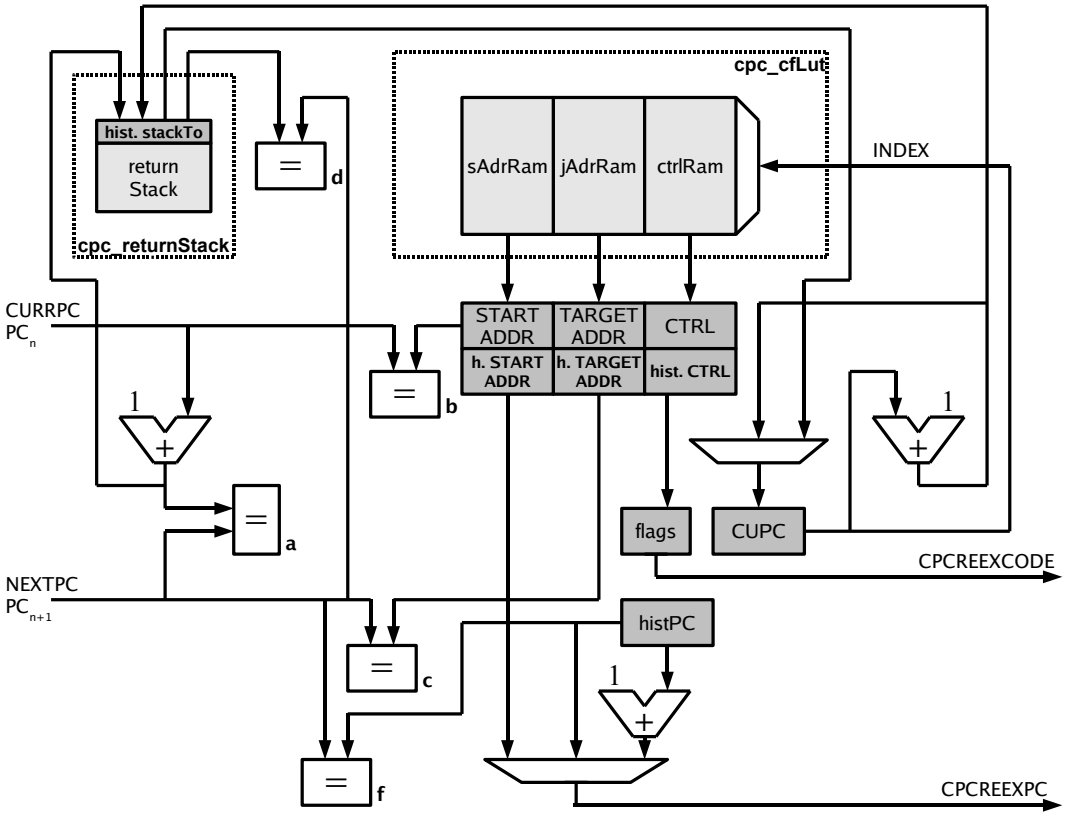


Figure 4.31: The architecture of the control flow checker with rollback protection for the data path protection technique. All internal state registers are duplicated by history registers. Comparator f checks the first rollback phase. If control flow errors during the rollback are detected, the checker can initialize a re-execution (see *CPCREEXPC*).

4.5.6 Example

To test the proposed concept and architectures for control flow checking, a real example is provided in this section. The simple example program is given in Listing 4.3 and a snippet of the resulting compiled assembler code for the SPARC V8 architecture is shown in Listing 4.4. The numbers on the left side denote to the corresponding instruction addresses. Additionally, for better understanding, the corresponding indexes of the CFI and CF method are shown as comments behind the instruction.

The compiled code is processed by the program analyzer tool *AISTool* for the CFI and the CF method. The program analyzer creates the initialization files for the checker memories and an optional graph. The created content of the checker memories for the CFI method is depicted in Figure 4.32 on the right side. Furthermore, the

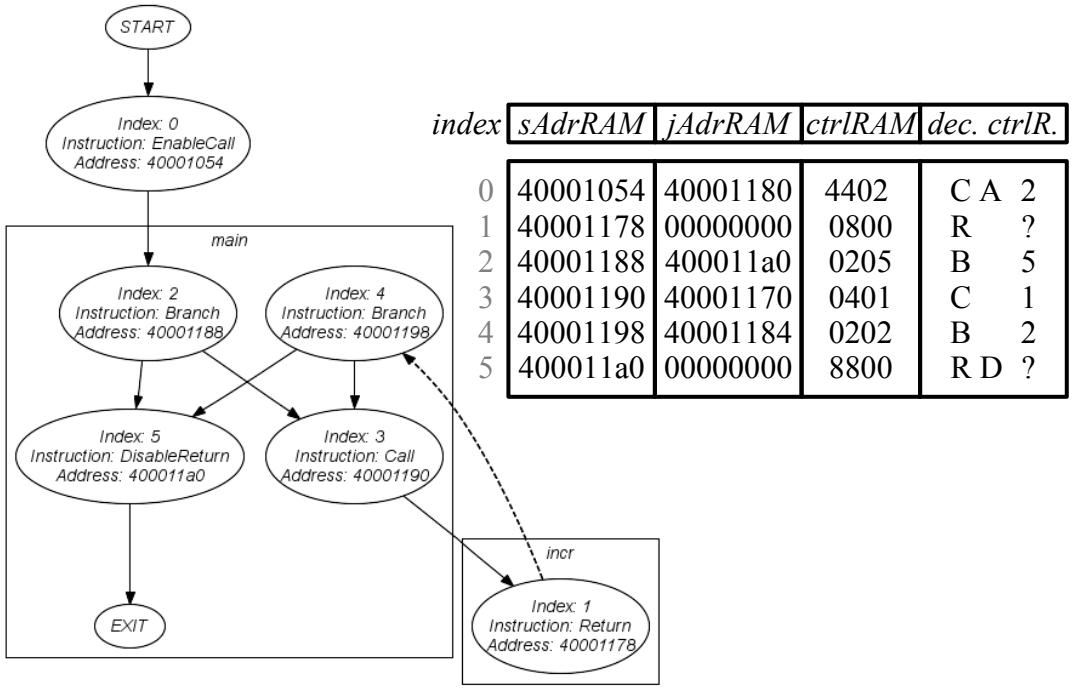
Listing 4.3 Example program written in C

```
1  int inc = 0;
2
3  void incr () {
4    inc++;
5  }
6
7  int main (int argc, char** argv) {
8    while (inc < 3) {
9      incr();
10   }
11 }
```

Listing 4.4 The compiled example program for the SPARC V8 architecture

```
1  40001054:  call 40001180 <main>    !CFI index: 0, CF index: 0
2  40001058:  nop
3  ...
4  4000116c:  restore                              !CF index: 1
5
6  40001170 <incr>:
7  40001170:  save %sp, -104, %sp
8  40001174:  inc  %g1
9  40001178:  ret                     !CFI index: 1, CF index: 2
10 4000117c:  restore
11
12 40001180 <main>:
13 40001180:  save %sp, -104, %sp                  !CF index: 3
14 40001184:  cmp  %g1, 2
15 40001188:  bg   400011a0           !CFI index: 2, CF index: 4
16 4000118c:  nop
17 40001190:  call 40001170 <incr>    !CFI index: 3, CF index: 5
18 40001194:  nop
19 40001198:  b    40001184           !CFI index: 4, CF index: 6
20 4000119c:  nop
21 400011a0:  ret                     !CFI index: 5, CF index: 7
22 400011a4:  restore
```

ctrlRam column is decoded in the last column for better understanding. The description of the flags are shown in Figure 4.26. On the left side, the CFIG is shown which is originally generated by the program analyzer tool. Figure 4.33 shows the memory content and the corresponding CFG for the CF method.



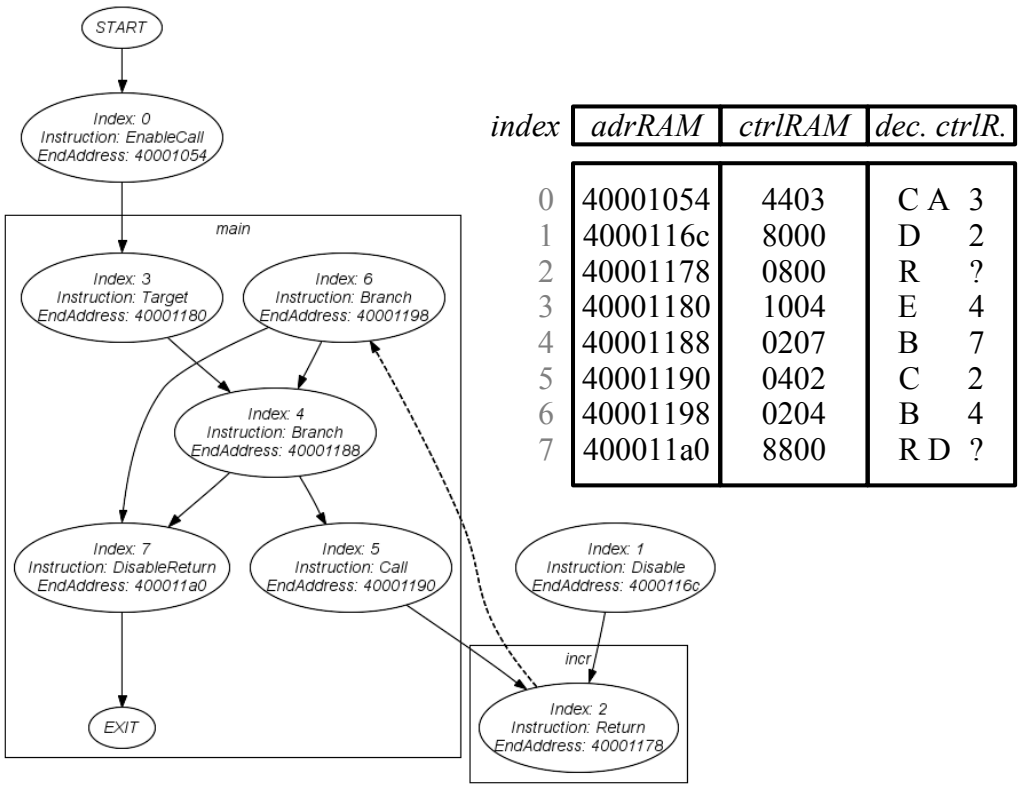
index	sAdrRAM	jAdrRAM	ctrlRAM	dec. ctrlR.
0	40001054	40001180	4402	C A 2
1	40001178	00000000	0800	R ?
2	40001188	400011a0	0205	B 5
3	40001190	40001170	0401	C 1
4	40001198	40001184	0202	B 2
5	400011a0	00000000	8800	R D ?

Figure 4.32: Results after processing the example program in Listing 4.4 with the program analyzer tool *AISTool*. On the left side, the CFIG graph, generated from the *AISTool* is shown. On the right side, the content of the checker memories are depicted. The last column shows the decoded *ctrlRAM* column with the flags and the decoded next CUPC index.

In the following, an example of error detection and correction is shown. This is done by using the CFI method with the example program from Listing 4.4 and the corresponding checker entries from Figure 4.32. Note that the CF method produces very similar results.

Figure 4.34 shows a waveform plot. In the first rows, the current program counter (*PC*) and the next program counter are shown (*nPC*). The checker unit checks the *call* instruction for calling the `incr()` subroutine. The CUPC points to the row in the checker memories where the call address as well as the call target address is stored (see the *from* and *to* rows). Furthermore, the *ctrl* register (ctrl content 0401) indicates the type of instruction (*call*) and the next CUPC index (1).

- In the second cycle, the program flow reaches the call ($PC = 1190$). The comparator *b* (*PC_eq_from*) denotes that a checking point is reached. Due to the



index	adrRAM	ctrlRAM	dec. ctrlR.
0	40001054	4403	C A 3
1	4000116c	8000	D 2
2	40001178	0800	R ?
3	40001180	1004	E 4
4	40001188	0207	B 7
5	40001190	0402	C 2
6	40001198	0204	B 4
7	400011a0	8800	R D ?

Figure 4.33: The results after processing the example program in Listing 4.4 with the program analyzer tool *AISTool* for the CF method. On the left side the generated CFG is shown and on the right side the corresponding checker memory content.

delay slot concept of the SPARC architecture, the call is executed in the next cycle.

- In the third cycle, the call is first executed correctly ($nPC = 1170$), but then an error occurs. The correct next program counter address 1170 is falsified to 3170. The error is detected by the checker unit because the comparator c (*nPC_eq_to*) is not longer true.

- In the forth cycle, the detected error is indicated by the error signal and the re-execution procedure is initialized. The starting point of the re-execution is the call instruction (address 1190) which is shown by the *reexPC* signal. The *reex-code* signal indicates that the falsified instruction was a call, and therefore, the latest three instructions in the pipeline should be annulled. This is done by annulling the decode (*d.annull*), the register access (*a.annull*), and the execution (*e.annull*) pipeline stages.

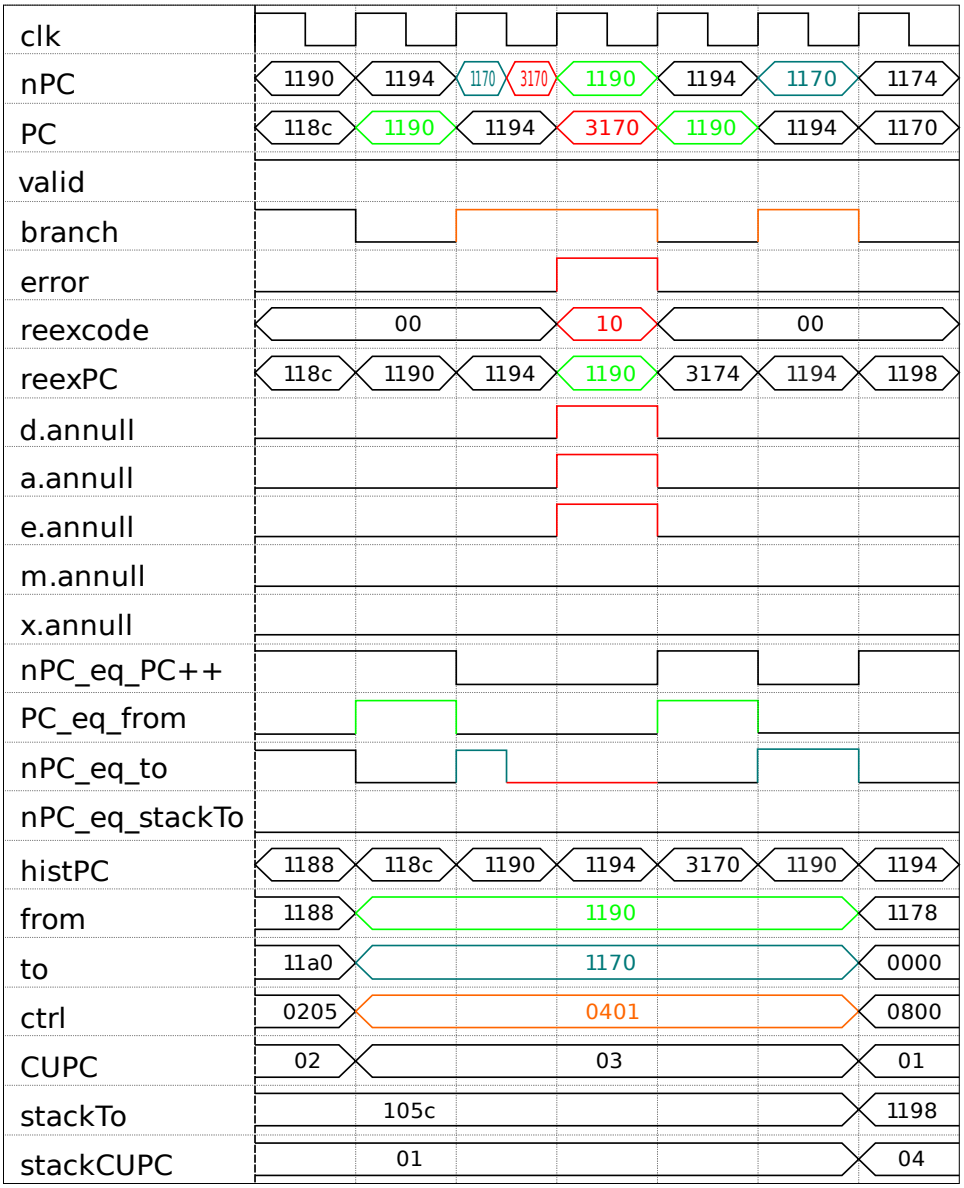


Figure 4.34: An example waveform showing the execution of the example program from Listing 4.4. During the execution of the *call* instruction on address 1190, an error occurs. The checker unit detects this error and starts the re-execution procedure which corrects the error [Koh08].

- In the fifth cycle, the call is executed again.
- In the sixth cycle, the jump is performed without error (comparator *c* is high).

After successfully executing the call, the values for the next checking point, the return instruction with index 1 is loaded into the register *from*, *to*, and *ctrl*. Furthermore, it is also shown that the back-jump address for the return is put on the stack ($stackTo = 1198$).

4.5.7 Simulation and Verification

In this section, we will discuss the verification of our control flow checker using simulation. For testing the handling of faults and errors with our control path checker, we need a fault model and methods for fault and error injection. The fault model and the different kinds of fault and error injection are discussed in Section 4.2. For the verification, only intentional fault injections are used.

Fault and Error Injection

Now, we present the implementation of different kinds of fault and error injection methods as introduced in Section 4.2.

Permanent Memory Errors: The memory content of the SDRAM is stored for simulation in `*.srec` files (`sram.srec`, `sdram.srec`, and `prom.srec`). The VHDL model of the external SDRAM reads the content from these files during the simulation. To inject permanent memory errors for simulation, we can alter the `*.srec` file. The checker memories of the checker unit which store the control flow graph or control flow instruction graph, are generated using *Xilinx Core Generator* (coregen) [Xilg]. The content of these checker memories is stored in `*.mif` files for simulation. Here, we can also alter these files to inject permanent memory errors.

To generate permanent memory errors on the FPGA implementation, we can further use the *in-circuit debugger* of the Leon3 processor. Using this debugger, we are able to change the contents of all memory cells which are writable and mapped to the address space of the processor over the AMBA bus interface. Since this applies to the main memory of the processor system and the checker memories, it is easy to alter values and memory contents in this way.

Transient on Chip Processor Faults: There are two different ways to inject transient faults for simulation. The first method is to use a simulator built-in command to set a signal to a specific value. In *Modelsim 6.3e*, the command is called *force*.

```
VSIM> force -freeze signame forceval forcetime -cancel releasetime
```

The signal *signame* is forced to the value *forceval* at *forcetime* until *releasetime*. The *freeze* parameter indicates that the original signal value is overwritten with *forceval*.

The second method to inject transient faults into the processor system is to use a fault injection unit instantiated in the integer pipeline of the Leon3. The injection unit can inject multiple faults in the control path on the same signals as the simulation command method (`rin.d.pc`). The faults are triggered by a *PRNG* (see Section 4.2 and [Koh08]). This method is able to inject *single event transitions* (SET) on a (pseudo-)random time with (pseudo-)random duration. This is done using the VHDL statements `wait for` and `transport after`. Because of the usage of these statements, the method is only working in simulation.

A synthesizable injection unit is coupled with the clock frequency. This restricts us to use a free time pattern for fault injection. We inject faults on the same signals as in the fault injection methods above. For triggering the injection, we use a *5-bit* counter. If the counter overflows and the integer pipeline of the Leon3 executes an instruction, a fault is injected. The processor executes approximately one instruction every 5 clock cycles in average. But this pattern is very irregular due to pipeline stalls caused by cache misses, branches, bus arbitration and so on. The combination of both, the counter overflow and the irregular pattern of execution of instructions, generates a (pseudo-)random distribution of injected faults.

Testbench and Verification

To verify the correctness of the checker unit by simulation, we are using the Leon3 *debug log*. With the debug log enabled, the Leon3 simulation model prints the address and name of every executed instruction on the ModelSim console. The output of the console can be logged into a file. If the debug log with and without fault or error injection is the same, the checker unit passes the test for the used test program.

For the complete verification, we used several test programs. These test programs are *pbm* and *qsort* from the Mibench benchmark [GRE+01] and the *turbo decoder* from AIS project partner TU Kaiserslautern (adapted from [BGT93]). We are using the counter fault injection method, because of this method is synthesizable. We can use this method for simulation and for an FPGA implementation. On the FPGA implementation, we check the output which is transferred over the RS232 link to the PC and compare it with the run without error injection.

Table 4.5 shows the simulation results of the test programs. Each program is executed twice, one run without and one run with error injection. The number of executed instructions and the overall clock cycles are measured. With these values, the *CPI* (Clocks per Instruction) value can be calculated. Furthermore, the number of injected errors are depicted. It can been shown that the CPI value increases with enabled

fault injection which indicates the performance overhead due to the re-execution of erroneous instructions (see also Figure 4.35).

test program	error inject.	no. checker entries	inj. errors (IER)	executed inst.	clock cycles	CPI
pbm		45	0 (0)	524925	844043	1.61
	X	45	12097 ($\approx 10^{-2}$)	524925	884071	1.68
qsort		24	0 (0)	444086	857969	1.93
	X	24	353 ($\approx 10^{-4}$)	444086	865393	1.95
turbo		113	0 (0)	3415530	5323733	1.56
	X	113	72516 ($\approx 10^{-2}$)	3415530	5585277	1.64

Table 4.5: This table shows the simulation results for the CFI method. Each test program is executed with and without error injection. The number of injected errors and the corresponding *injection error rate* (IER) in errors per clock cycles are depicted. Furthermore, the executed instructions, the used clock cycles and the CPI (Clocks per Instructions) are shown.

The number of needed clock cycles for re-execution of an erroneous control flow depends on the currently executed instruction. For a simple program counter increment (no control flow instruction), we are able to correct the error in one additional clock cycle, whereas the correction of an erroneous return instruction needs five clock cycles. Furthermore, cache misses due to falsified branch or jump targets have also an impact on the latency.

4.5.8 Synthesis and Implementation

The Leon3 processor and the different versions of the checker unit are synthesized using the *Xilinx XST* synthesis tool of the *ISE 8.2* framework and implemented on a *Xilinx Virtex-4* FPGA. In the following, an overhead analysis is provided for different versions of the checker which supports different jump instructions and error detection features and thus, result in different area overheads.

The smallest version of the checker (version *A*) implements the CFI basic method which can only monitor direct jumps or branches. No indirect jumps are supported and are therefore not allowed in the code, but it is allowed that indirect jumps can occur in the unchecked code. This includes also returns from subroutine. So, this technique can only be used for a single procedure or function. But many of these procedures and functions can be checked if the checker unit is de-activated at calls and returns, and activated inside each function.

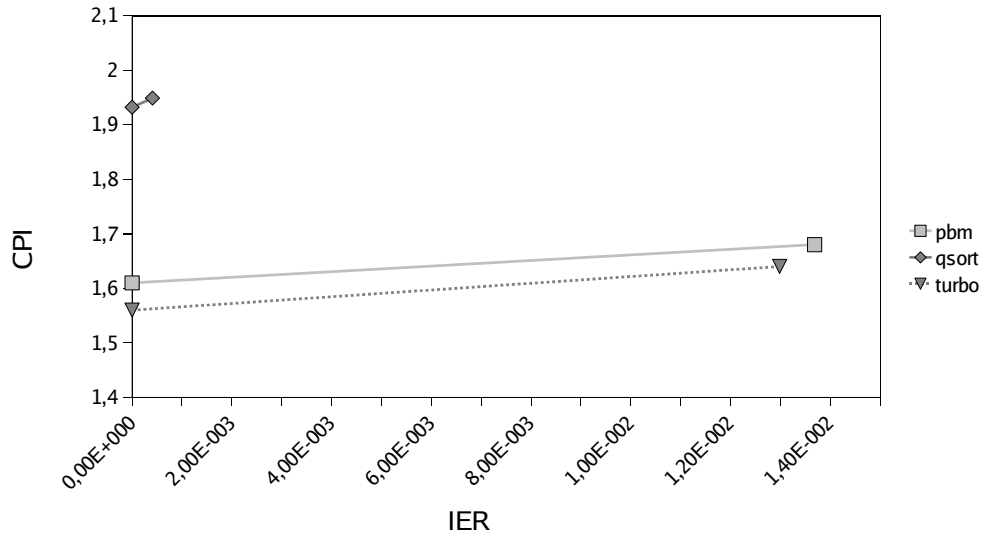


Figure 4.35: Plot of the number of cycles per instruction (CPI) over the injected error rate (IER) according to the results reported in Table 4.5 for the three test programs.

The second version (version *B*) has an additional 32-entry return stack extension. With this version, we can also monitor calls and returns, so most application programs can be fully monitored.

The last version (version *C*) has the additional capability of repairing an incorrect control flow by re-execution (re-execution extension).

Table 4.6 shows the measured resource overheads of these three versions. The results show that the area overhead for logic (in terms of lookup tables and flip-flops) is very small compared to the amount of resources needed by the processor core. If more control flow instructions shall be monitored and more checker memory entries are needed, only the overhead of necessary block rams (BRAMs) increases. Using the CF methods instead of the CFI, the area overhead increases slightly for the same maximum number of checker memory entries whereas the memory overhead decreases. Note that typically, the CF method needs more checker entries than the CFI method for the same program. The bus interface for the *AMBA* bus creates an additional area overhead of *68 LUTs* and *5 flip-flops* which can be subtracted from the values above if the bus interface is not needed.

The overhead values above consider the control flow checker implementations only. If these methods should be combined with the data path protection technologies as described in Section 4.5.5, the area overhead increases. For example, a CFI method in version *C* configuration (with return stack and re-execution extension) which is able to check also the micro rollback from the data path protection needs

Overhead	Leon3	CFI Method Version *A*		Version *B*		Version *C*		CF Method diff. to CFI	
checker memories with 512 entries									
LUTs	13554	325	2.40%	491	3.62%	579	4.27%	+51	+0.38%
flip-flops	3476	69	1.99%	74	2.13%	106	3.05%	+48	+1.38%
BRAMs	50	3	6%	3	6%	3	6%	-1	-2%
checker memories with 1024 entries									
LUTs	13554	329	2.43%	494	3.64%	582	4.29%	+54	+0.40%
flip-flops	3476	71	2.04%	76	2.19%	108	3.11%	+49	+1.41%
BRAMs	50	5	10%	5	10%	5	10%	-2	-4%
checker memories with 2048 entries									
LUTs	13554	335	2.47%	499	3.68%	575	4.24%	+57	+0.42%
flip-flops	3476	73	2.10%	78	2.24%	110	3.16%	+50	+1.44%
BRAMs	50	8	16%	8	16%	8	16%	-3	-6%
checker memories with 4096 entries									
LUTs	13554	343	2.53%	507	3.74%	595	4.39%	+60	+0.44%
flip-flops	3476	75	2.16%	80	2.30%	112	3.22%	+51	+1.47%
BRAMs	50	10	20%	10	20%	10	20%	-4	-8%

Table 4.6: Resource overheads in terms of FPGA primitives for different checker unit versions (including the bus interface) with respect to a *Leon3* core without PCI and Ethernet. The area (4-input LUTs) and memory overheads (flip-flops and BRAMs) of the full version *C* and the reduced versions (*A* and *B*) and for different checker memory sizes are shown in absolute numbers and in percent of the resources needed by the *Leon3* core. The difference between the CF and CFI method is depicted in the last columns. Note that the difference is independent of the used version (*A*, *B*, or *C*).

561 4-input LUTs and *202 flip-flops* [Koh08]. Of course, the data path protection itself causes also an additional area overhead.

If we implement also the error injection unit, we must reduce the clock frequency to 25 MHz. The reason is that we inject errors at the falling edge of the clock. Due to this reason, we have only half of a clock cycle time to meet the timing constraints for these paths which are additionally extended by logic of the error injection unit. Without the error injection unit, we have no timing degeneration compared to the original Leon3 core.

4.6 Case Study: Turbo Decoder

In this section, we present another case study for control flow checking on a demonstrator platform consisting of an FPGA board and a *turbo en/decoding* example application. This demonstrator combines the joint integrative work of the different project partners of the AIS project to show the increase of reliability as well as the corresponding overheads. Furthermore, a deeper look into the control flow checking part is given which shows also some error detection and correction scenarios.

4.6.1 The AIS Demonstrator

The AIS demonstrator consists of the FPGA board *Leon3-GR-CPCI-XC4V* from Gaisler [Gaia] and an example application, a rapid prototyping turbo de/encoder application delivered by the project partner TU Kaiserslautern [MW08]. The FPGA board consists of a Xilinx *Virtex-4 XC4LX100* FPGA, 256MB SDRAM, and compact PCI, Ethernet, JTAG, and RS232 as interfaces (see Figure 4.36). The FPGA is large enough to include at least four Leon3 [Gaib] CPUs.

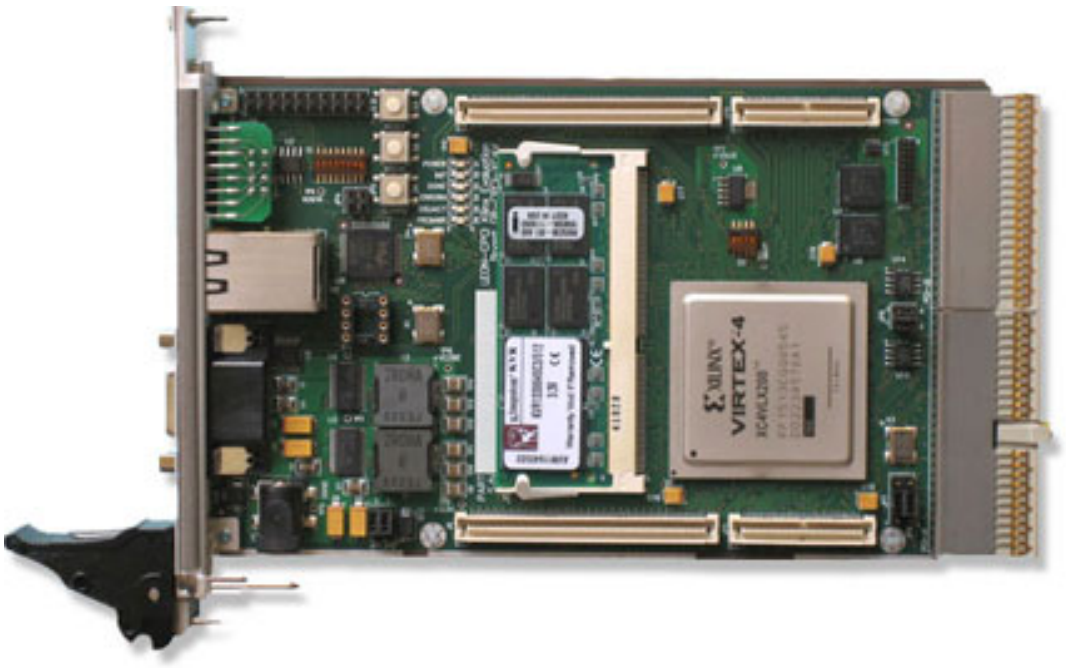


Figure 4.36: The Gaisler *Leon3-GR-CPCI-XC4V* FPGA-board [Gaia] used as AIS demonstrator.

We use the SDRAM as a shared main memory for the Leon3 *MPSoC* (Multi Processor Sytem on Chip). The SDRAM is connected with a memory controller from

Gaisler, implemented inside the FPGA, to the AMBA-Bus system which is the central communication infrastructure for all FPGA cores including the Leon3 CPUs. Furthermore, we use the RS232 interface for the debugging link to the Gaisler Leon3 debugging and monitor software *grmon*. Over the JTAG interface, we can configure and debug the FPGA with the Xilinx software *Chipscope* and *Impact*.

The application is a turbo en/decoder rapid prototyping system (see Figure 4.37a). The system consists of different modules of a data transmission chain. From a data generator, the data is channel-encoded by a turbo encoder and *binary phase-shift keying* (BPSK)-modulated. After that, the data passes through the *AWGN* (Additive White Gaussian Noise) channel. Inside the AWGN channel simulation, the signal subject to noise so that errors which occur on a real AWGN channel transmission can be simulated. Finally, the received data is demodulated and is handed over to the turbo decoder. The decoded data is finally compared to the generated data and the emerged errors are monitored.

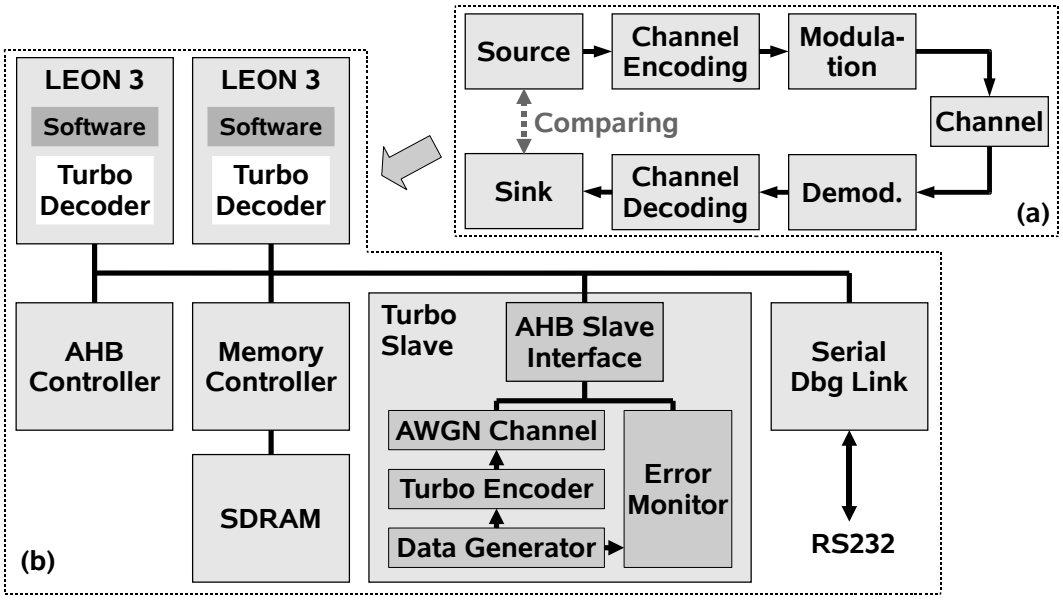


Figure 4.37: On the upper right side (a) the turbo en/decoding application provided by AIS project partner TU Kaiserslautern [MW08, MWB+10] is shown. The data flow from the generator over all different modules to the channel decoding is depicted. Finally, the received data is compared with the original one, and the emerged errors are recorded. On the left side (b), the demonstrator system architecture with the mapped components is shown. The turbo decoding part is running in software on different Leon3 CPUs, whereas the other components are implemented in hardware and are accessible over an AMBA Advanced High-performance Bus (AHB) interface.

This system is mapped into our demonstrator architecture (see Figure 4.37b). The data generator, turbo encoder, AWGN channel, and the error monitor are implemented as a hardware module inside the FPGA. The channel and the error monitor are connected to the AMBA bus. The turbo decode module is implemented in software running on one or more Leon3 CPUs. The data to and from the turbo decoder is transmitted over the AMBA Bus. Additionally, the main memory (SDRAM) is connected to the AMBA Bus over the memory controller.

With this architecture, we are able to evaluate and demonstrate the autonomous units developed for the AIS project. We can inject faults into the AMBA bus communication and the control and data paths of the Leon3 CPUs.

4.6.2 Control Flow Checking Contribution

For the control flow protected Leon3 CPU, we implemented the VHDL model of the CFI method with the return stack, re-execution, and bus interface extensions. To emulate control path errors, we used a synthesizable error injection unit based on a *5-bit* counter as introduced in Section 4.5.7. The number of detected and corrected errors is recorded by an error counter which is accessible from software over a memory-mapped register. Also, the fault injection unit can be switched on or off by software, by setting a certain bit in a control register. The status of the fault injection unit is shown in the output of the software which is transferred over a RS232 link and can be displayed using a serial terminal program. Also, the read values from the error counter register as well as a clock cycle and executed instructions counter register are displayed.

Additionally, the program counter signals from the first pipeline steps as well as the *CPCREEX* (see Section 4.5.3) signal which shows that a re-execution procedure is in progress, are recorded and can be displayed with the Xilinx debugging tool *Chipscope*.

Measuring the executed instructions and clock cycles when the turbo decoding software is executed shows us that the average CPI (clocks per instructions) increases when the error injection is switched on due to additional clock cycles for correcting errors than without error injection (see Table 4.7). An additional latency of 4.7 clock cycles on average for each injected error is measured.

Figure 4.38 shows screenshots from Chipscope. In the upper trace (a), a fault is injected at a non control flow instruction. The normal sequence from `0x40001F68` over `0x40001F6C` to `0x40001F70` is interrupted by an error of the program counter (`0x40009F6C`). The checker unit has detected this error and re-executes the erroneous instruction (`0x40001F6C`).

In the trace below (b), the error is the wrong target (`0x40009898` instead of `0x40001898`) of a branch (`0x40001EA8`). In this case, the error is also detected and the branch is re-executed. Note that `0x40001EAC` is the delay instruction.

operation	clock cycles $\times 10^6$	instruct. $\times 10^6$	inj. errors $\times 10^6$ (IER)	CPI
no error inj.	2575	1386	0 (0)	1.86
error inj.	3078	1565	37.430 ($\approx 10^{-2}$)	1.97

Table 4.7: Number of executed clock cycles and instructions of the Turbo decoding software to show the additional latency of the CPU correction methods. Also, the clock cycles per instructions (CPI) are shown.

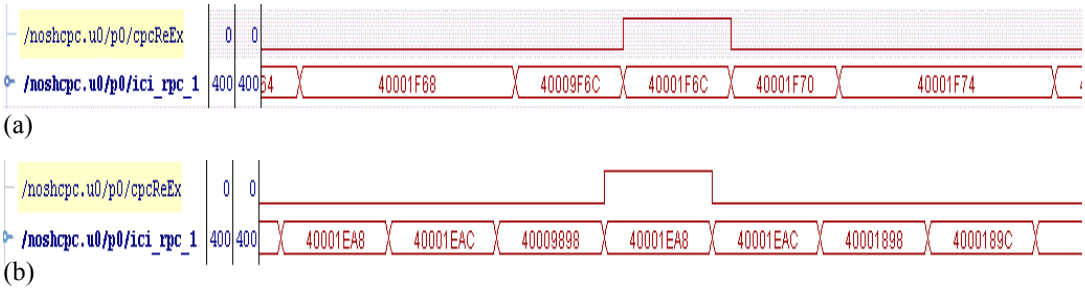


Figure 4.38: These recorded signal traces from *Chipscope* show error detection and correction. In the upper trace (a) the correction of a non control flow instruction is shown. The successive linear execution is disturbed by an error at the instruction address `0x40001F6C` and after successful detection, the erroneous instruction is re-executed. The trace below (b) shows an error at a branch instruction with the following re-execution of the branch.

4.7 Summary

We have introduced a systematic methodology for autonomous control flow checking for embedded RISC CPUs which can monitor and correct the control flow. Different methods and architectures were introduced which cover all different types of instructions as well as many faults and errors in control paths of an embedded processor. The cadre of the architecture are the basic methods (CFI / CF method). Based thereupon, the proposed architectures are *modular*, so it is easy to remove and add features like the return stack, to get a lower area overhead or support more types of instructions. A huge repertoire of different extensions which enhance the error detection possibilities, supporting different types of control flow instructions, enable re-execution features, and simplify the debugging of the checker unit are introduced which have the possibility to increase the reliability as well as security of a processor enormously. Furthermore, techniques for checking general IP cores are introduced which can in particular detect single event effects and permanent faults.

Implementations of some of these embedded processor checking modules were provided which can monitor and correct direct jumps and branches as well as returns from subroutine. Experimental results show that the additional hardware overhead is quite small. In particular, lookup tables and flip-flops overhead amount to an overhead of less than 5% of the CPU core requirements in all cases. So, the actual overhead results from the additional memory needed to monitor the control flow instructions. According to Michel et al. [MLS91], the Cerberus-16 processor has a memory overhead of 34-85%, whereas the WDP approach has an overhead of 24-61% (see Section 2.4.2). Our control flow checking approach has a memory overhead of only 15-30% according results reported in Table 4.2. Please note that these values are only comparable with care due to different processor architectures and benchmarks. Furthermore, the values from Table 4.2 do not include the overhead for the *instruction integrity checker* (IIC). When also considering the IIC, the memory overhead of our method is increased by approximate 5-10%.

A modular concept for generation of generic micro-programmable checker units has been proposed, so the area overhead can be further reduced by removing some modular components. The detection of errors happens during the execution of the erroneous instruction, so we have the possibility to react immediately and prevent any incorrect instruction from being executed. Furthermore, an incorrect jump or branch instruction can be re-fetched and re-executed. With this technique, we therefore have no performance impact on the CPU in the error-free case and the compiled program code remains also unchanged.

In our implementations, an independent program counter *CUPC* with its own state machine and own micro-code with own micro-instructions (*ctrlRam*) was introduced. The checker micro-program code is based on the extracted branch and jump instructions from the program code. This reduced code covers only direct branches or jumps without the instructions between two branch points. With this technique, we enhanced the CPU with a reduced second independent program counter and instruction unit at minimum additional hardware cost and full control of the program flow. The approach of Arora et al. [ARRJ06] has some similarities with our approach. The advantages of ours, however, is that we are a) more flexible in using memories to store the control flow (instruction) graph instead of synthesizing a dedicated finite state machine (FSM) in logic for each program to be executed. Moreover, we have b) no performance impact in the error-free case, and c) our checker unit is simpler and thus requires less resources. Compared to all existing approaches, however, the main advantage is the tight integration into the processor, which enables us to detect errors very fast and before the error manifests into the register of the processor which often might be too late to circumvent failures or attacks. This allows us also to immediately correct an error by a simple re-execution of the last instructions.

Finally, using the bus interface of the checker unit, the contents or part of the content of the checker memories might also be stored in the system memory instead of dedicated memories. Only the content for checking the current part of the program

(e.g., the current function or a set of functions which are current in use) may be held in the local checker memories. If the checker needs information which is not stored inside the checker memories, the checker can generate a page-fault-like event to signal the operating system to reload the these memories with the needed contents. This concept of caching can reduce the memory overhead significantly.

5 Conclusions

This dissertation provides an overview of *security and reliability issues* for IP cores in embedded systems. Potential faults and attacks, which cause common security and reliability problems, were classified. Major contributions of this thesis are novel techniques for *IP protection* of cores and *control flow checking* for RISC processor cores.

The justification of the existence of such methods which clearly cause additional overhead and costs, are the ongoing technical advances of the underlying semiconductor technology. These advances lead, on the one hand, to more complex systems, consisting of billions of transistors. On the other hand, the individual transistors are becoming more and more unreliable in new technology generations due to increased process variation, accelerated aging, and increased sensitivity for single event effects. Current design methodologies like *worst-case design flow* or *proprietary interfaces* cannot keep up with this development. New design methodologies are needed to handle these challenges for systems using billions of transistors, which include the massive reuse of designs and cores. *Interface standardization* of IP cores will be a huge topic in the future, which will provide solutions, but also cause problems. Standardized IP core interfaces will boost the growth of the already vast market for reusable IP cores for ASIC and FPGA technologies enormously. Clearly, the question of confidentiality and protection against unlicensed usage of IP cores is becoming more and more important. Our presented *watermarking techniques* can be a solution to this problem for several IP cores. Furthermore, these complex systems consisting of manifold kinds of IP cores, such as processors, bus systems, interface cores, hardware accelerators, stand on shaky ground. The decreased reliability of the individual transistors leads to the loss of faith that the underlying semiconductor technology,

combined with the corresponding verified design flow are error-free. The well known worst case design flow is not longer applicable. Future designs and systems must deal with the unreliable foundation they are built upon. They have to autonomously react to environmental changes, and monitor faults. In other words, future designs must take care of the underlying resources by using them resource-aware. However, this resource-aware autonomous methodology needs new design paradigms which results in a redesign of existing cores. To circumvent the need to redesign, existing cores must be extended by additional elements to implement this autonomous behavior. The *control flow checking* methodology proposed in this thesis provides one of these units which is able to monitor faults and errors caused by the underlying unreliable technology and which can correct them autonomously.

Summary

A brief summary of the three parts of this dissertation is given as follows:

- **Overview of Security and Reliability Issues for Embedded Systems:** In Chapter 1, *security* and *reliability* problems for embedded systems are analyzed and described. Basic definitions in the area of *security*, and *dependability* are given. A taxonomy of faults and attacks in embedded systems is presented and the severity of these issues is emphasized, in particular for future technology generations. In Chapter 2, related work on techniques to increase security and reliability is presented. In focus are techniques for IP protection and countermeasures for code injection attacks which corresponds to security issues, as well as common techniques for increasing the reliability of IP cores in embedded systems. Finally, *control flow checking* techniques for embedded processors are discussed which combine security and reliability issues.

- **Watermarking and Identification Techniques for FPGA IP Cores:** In Chapter 3, novel IP core *watermarking* and *identification* techniques for FPGA designs are presented. These techniques were developed in order to detect an unlicensed usage of IP cores. First, a *general watermarking model*, introduced by Li et al. [LMS06], is extended and adapted to watermarking and identification of IP cores used in the electronic design automation flow. Techniques for identification of HDL and netlist cores as well as approaches for watermarking of netlist and bitfile cores are proposed. All these methods have in common that only the FPGA bitfile is needed from the company suspected of IP fraud. Moreover, novel watermarking techniques that verify the authorship by monitor the power consumption, so-called *power watermarking*, are introduced. Starting from a basic approach, many different encoding and decoding methods are analyzed. Furthermore, different multiplexing techniques for concurrent sending of different watermarked cores, combined in the same FPGA,

are investigated. Finally, the effectiveness, but also the additional costs are confirmed by experimental results.

- **Control Flow Checking for Embedded RISC Processors:** In Chapter 4, new methods, architectures, and implementations for *autonomous control flow checking* in embedded RISC processors are described. First, an overview of the project *Autonomous Integrated Systems* (AIS) and some initial remarks for fault injection are given. After that, methods and architectures for control flow checking of all types of control flow instructions in embedded RISC cores and common IP cores are introduced. For direct jumps and branches, two alternative methods are proposed which form a basis architecture. This basis architecture can be extended towards support of indirect jumps, detecting more types of errors, or error correction measures. An example implementation of the basis control flow checker for RISC processors as well as many extensions is shown for the *Leon3 SPARC V8* core. Moreover, the interaction with a data path protection scheme and the corresponding overheads are elaborated. Finally, a case study consisting of an error protected *multi-processor rapid prototyping platform* which implements a turbo encoding/decoding system is demonstrated.

Future Directions

Security and reliability issues in embedded systems are becoming more and more important with future generations of semiconductor technology. Therefore, the need for methods and concepts as presented in this dissertation, will increase. However, there are several extensions to these methods and new techniques which worth to investigate in the future.

In the area of *IP protection*, several links are still missing to complete the chain for *identification of HDL cores* in FPGA bitfiles. The identification of netlist cores in the FPGA bitfile and the comparison of different netlists in order to decide if they were generated from the same source is possible. Future work might bring both methods together to identify HDL cores in bitfiles. Another interesting technique is to verify watermarks using *electromagnetic radiation*. The advantages over power watermarking techniques are an easy measurement without soldering wires onto the PCB board, and the raster scanning over the FPGA area which provides additional geometric location information. *Power watermarking techniques* can be further improved by investigation of new encoding and decoding methods in order to be more resistant against noise and disturbances. Furthermore, these techniques can be used to transfer general data from the cores out of the FPGA. *Power communication* might be useful if a communication possibility must be added later in the development and no further dedicated pins of the FPGA or wires on the board are available. Possible applications include *monitoring* core status information or *debugging*. With the methods presented in this dissertation, we achieve data transfer rates of up to 500

kbit/s. Finally, all these watermarking and identification techniques can be bundled into a *framework* which can be tightly integrated with the design flow in development environments.

Future research in the context of *control flow checking* techniques for embedded RISC processors might consider the minimization of the memory overhead, full support for indirect jumps and support of other processor architectures. The memory overhead can be reduced by introducing hierarchical checker structures as, for example, in [ARRJ06], and compression methods. Using *hierarchical checkers*, relative addresses can be used and combined with memory compression algorithms and thus, the memory overhead can be drastically decreased. By using caching techniques where the checker content is stored in the system memory and loaded into a cache-like on-chip memory for checking the current part or function, the amount of expensive on-chip memory can be further reduced. The reloading of the cache may be done by the operation system or autonomously by the checker unit. The approaches for checking indirect jumps can also benefit from the memory overhead reduction techniques. Furthermore, the identification of valid indirect jump targets is also a potential area of future research. Finally, our control flow checking approaches can be adapted to other types of processor architectures. In particular *single instruction, multiple data* (SIMD) architectures and multi-cores which execute the program synchronously in parallel are suitable for our checking approach. On these architectures, only one checker unit is needed which reduces the relative hardware overhead.

Finally, the IP core watermarking techniques are applicable only to FPGA targets and design flow, whereas the reliability methods presented in this dissertation are focused on ASIC designs. The combination and adaption of both so to increase the reliability of FPGA designs might also be an interesting future area of research.

A

German Part

Methoden zur Verbesserung der Sicherheit und Zuverlässigkeit von eingebetteten IP-Cores in FPGA- und ASIC-Designs

Zusammenfassung

Die vorliegende Arbeit gibt einen Überblick über *Sicherheits- und Zuverlässigkeitsaspekte* in eingebetteten Systemen. Hierbei wurden mögliche Fehlerquellen und Sicherheitsangriffe, sowie geeignete Gegenmaßnahmen aufgezeigt. Speziell wurden neue Techniken zum *Schutze geistigen Eigentums* (engl. *Intellectual Property* – IP) von Schaltungsblöcken sowie *Kontrollflussüberwachungsmaßnahmen* für RISC Prozessoren untersucht.

Der Einsatz solcher Methoden, die zusätzliche Ressourcen benötigen und damit Mehrkosten verursachen, ist notwendig und gerechtfertigt durch die Fortschritte in der Halbleitertechnologie. Diese Fortschritte erlauben komplexere Systeme, die einerseits aus Milliarden von Transistoren bestehen können, jedoch andererseits mit jeder neuen Technologiegeneration immer unzuverlässiger werden. Gründe hierfür sind höhere Variationen in der Herstellung, eine schnellere Alterung, sowie eine höhere Anfälligkeit gegenüber sogenannten *Single-Event-Effects*. Heutige Entwurfsstrategien wie der *Worst-Case Entwicklungsfluss* oder *proprietäre Schnittstellen* können mit dieser Entwicklung nicht mithalten. Um zum Beispiel Schaltungen zu beherrschen, die aus Milliarden von Transistoren bestehen, müssen neue Entwurfsstrategien angewandt werden wie die massive Wiederverwendung von gekauften oder selbst geschriebenen Schaltungsblöcken. Die Standardisierung der Schnittstellen von Schaltungsblöcken, die zwar Lösungen bringt aber auch Probleme verursachen kann, wird eine Herausforderung sein. Das Wachstum des Markts von wiederverwendbaren Schaltungsblöcken für die FPGA oder ASIC Technologie wird durch die Standardisierung der Schnittstellen enorm beschleunigt. Natürlich wird dadurch auch die Frage nach dem Schutz von geistigem Eigentum immer wichtiger. Die in dieser Arbeit vorgestellten *Wasserzeichenmethoden* können für viele Schaltungsblöcke eine Antwort auf diese Frage sein. Andererseits stehen diese komplexen Systeme – bestehend aus vielseitigen Schaltungsblöcken wie Prozessoren, Bussen, Schnittstellenschaltungen, Hardwarebeschleunigern, usw. – auf wackligen Beinen. Dass die grundlegende Halbleitertechnologie kombiniert mit dem entsprechenden verifizierten Entwurfsfluss fehlerfrei ist, kann durch die abnehmende Zuverlässigkeit der einzelnen Transistoren nicht mehr sichergestellt werden. Der bekannte Worst-Case Entwurfsfluss ist in Zukunft nicht mehr anwendbar. Zukünftige Systeme müssen sich mit der unzuverlässigen Halbleitertechnologie auf der sie aufbauen auseinandersetzen und autonom auf Änderungen der Umgebungsbedingungen sowie Fehler reagieren. Mit an-

deren Worten: Sie müssen auf die zugrundeliegende Halbleitertechnologie Rücksicht nehmen, indem sie diese ressourcengewahr verwenden. Diese neue autonome ressourcengewahre Benutzung verlangt aber auch neuen Entwurfsparadigmen, welche üblicherweise einen Neuentwurf existierender Schaltungen nach sich zieht. Um dies zu verhindern, können auch existierende Schaltungen mit neuen Elementen, welche dieses autonome Verhalten bereit stellen, erweitert werden. Die in dieser Arbeit vorgestellte *Kontrollflussüberwachungseinheit* ist eine solche autonome Einheit, die dem entsprechenden Prozessor die Möglichkeit gibt, Fehler der zugrundeliegenden Halbleitertechnologie zu erkennen, und diese selbstständig zu korrigieren.

Kapitelübersicht

Im Folgenden wird eine kurze Zusammenfassung der Kapitel gegeben:

- **Sicherheits- und Zuverlässigkeitsaspekte in eingebetteten Systemen:** In Kapitel 1 werden aktuelle *Sicherheits-* und *Zuverlässigkeitsprobleme* aufgezeigt, die in heutigen eingebetteten Systemen vorkommen. Dazu werden auch grundlegende Begriffe und Definitionen aus dem *Sicherheits-* und *Verlässlichkeitsbereich*, sowie mögliche Fehler und Sicherheitsangriffe vorgestellt. Die Wichtigkeit dieser Aspekte, sowie die Notwendigkeit von Gegenmaßnahmen wird dadurch unterstrichen. Diese Maßnahmen werden in Kapitel 2 vorgestellt. Im Mittelpunkt standen insbesondere Maßnahmen zum Schutz geistigen Eigentums, Gegenmaßnahmen für sogenannte *Code-Injection-Angriffe* zur Erhöhung der Sicherheit, sowie Maßnahmen um die Zuverlässigkeit von eingebetteten Schaltungen zu verbessern. Des Weiteren werden *Kontrollflussüberwachungsmethoden* vorgestellt, die gleichzeitig Sicherheits- als auch Zuverlässigkeitsaspekte vereinen.

- **Wasserzeichen und Identifikationsmethoden für FPGA-Schaltungen:** Kapitel 3 beschäftigt sich mit neuen *Wasserzeichen-* und *Identifikationsmethoden* zum Schutz geistigen Eigentums von FPGA-Schaltungsblöcken. Diese Methoden können eine nicht lizenzierte Verwendung des Schaltungsblocks aufdecken. Zuerst wird ein von Li und anderen [LMS06] entwickeltes *allgemeines Modell für Wasserzeichenmethoden* um Identifikationsvefahren für elektrische Schaltungen erweitert. Danach werden Identifikationsmethoden für sogenannte Netzlisten- und HDL-Schaltungsblöcke, sowie Wasserzeichenmethoden für Netzlisten- und Bitfile-Schaltungsblöcke vorgestellt. Alle diese Verfahren benötigen nur das FPGA-Bitfile aus dem Produkt der Firma, die im Verdacht steht, Schaltungsblöcke anderer illegal einzusetzen. Des Weiteren wird ein neuer Ansatz vorgestellt, bei dem das Wasserzeichen durch den Energieverbrauch des FPGAs ausgelesen und verifiziert wird – das sogenannte *Power Watermarking*. Zuerst wird eine Basismethode dieses Verfahrens vorgestellt, die um

verschiedene Kodierungs- und Dekodierungsansätze erweitert wird. Zusätzlich werden noch verschiedene Multiplexverfahren präsentiert, die ein gleichzeitiges Senden von mehreren Wasserzeichen verschiedener Schaltungsblöcke in einem FPGA erlauben. Zum Schluss werden die Kosten und die Funktionsweise aller Methoden durch Experimente an Beispielschaltungen verifiziert.

- **Kontrollflussüberwachung für eingebettete RISC Prozessoren:** In Kapitel 4 werden Methoden, Architekturen und Implementierungen für *autonome Kontrollflussüberwachung* in eingebetteten Prozessoren vorgestellt. Diese Verfahren werden im Rahmen des Projekts *Autonome Integrierte Systeme* (AIS) untersucht, welches zu Beginn erörtert wird. Zusätzlich werden grundlegende Aspekte zur Fehlerinjektion analysiert. Danach werden die Methoden und Architekturen für die Kontrollflussüberwachung vorgestellt, welche alle Arten von Kontrollflussbefehlen eines eingebetteten RISC Prozessors abdecken. Dabei werden allgemeine Schaltungsblöcke nicht außer Acht gelassen. Zwei alternative Verfahren zur Überwachung von direkten Sprüngen und Verzweigungen bilden die Basisarchitekturen, welche durch zusätzliche Module zur Überwachung von indirekten Sprüngen, Erkennung von anderen Arten von Fehlern oder Fehlerkorrekturmaßnahmen erweitert werden können. Eine beispielhafte Implementierung der Basisarchitekturen und einige Erweiterungen wurde für den SPARC V8 Prozessor Leon3 realisiert. Zusätzlich wurde eine Datenpfadüberwachung in unsere Verfahren integriert und der zusätzliche Ressourcenaufwand untersucht. Zum Schluss wird eine Fallstudie präsentiert, die aus einem *Multiprozessorsystem* besteht, welches um unsere Fehlererkennungs- und Korrekturverfahren erweitert wurde, und auf dem eine Kodierungs- und Dekodierungsanwendung nach dem Turbo-Verfahren ausgeführt wird.

Ausblick auf weiterführende Themen

Auch mit zukünftigen Halbleitergenerationen werden Sicherheits- und Zuverlässigkeitsaspekte in eingebetteten Systemen immer wichtiger. Damit wird auch die Notwendigkeit der in dieser Arbeit entwickelten Methoden und Konzepte steigen. Jedoch ist die Arbeit an diesen Methoden keinesfalls abgeschlossen und zukünftige Erweiterungen und neue Techniken bieten genug Raum für weitere Forschungen.

Auf dem Forschungsgebiet des *Schutz geistigen Eigentums* von FPGA-Schaltungsblöcken fehlen noch einige Glieder der vollständigen Kette um HDL-Schaltungsblöcke in einem FPGA-Bitfile ohne Wasserzeichen zu identifizieren. Die Identifikation von Netzlistenschaltungen in FGPA-Bitfiles, sowie die Entscheidung, ob mehrere Netzlistenschaltungen von dem gleichen HDL-Schaltungsblock generiert wurden, ist jetzt schon möglich. In zukünftigen Arbeiten können diese beide Verfahren vereinigt werden, um HDL-Schaltungblöcke in FPGA-Bitfiles zu identifizieren. Das Auslesen und Verifizieren von Wasserzeichen mittels *elektro-magnetischer Abstrahlung* ist

auch ein interessantes zukünftiges Forschungsgebiet. Vorteile gegenüber dem Auslesen über die Spannungsversorgung sind unter anderem das einfache Messen ohne aufwändige Lötarbeiten an der Platine und die Möglichkeit eines Abtasten der Abstrahlung an verschiedenen Stellen über der FPGA-Fläche, welches zusätzliche geometrische Lokalisierungsinformationen liefert. Aber auch das *Power Watermarking Verfahren* lässt sich mit neuen Kodierungs- und Dekodierungsmethoden erweitern, um im Falle von Störungen noch bessere Dekodierungsergebnisse zu bekommen. Darüber hinaus lassen sich diese Verfahren auch zur Übertragung allgemeiner Daten aus dem FPGA einsetzen. Diese sogenannte *Power Communication* könnte dann in Fällen hilfreich sein, wenn diese Kommunikationsmöglichkeit erst spät in der Entwicklung eingesetzt werden muss und keine weiteren dafür vorgesehene Pins oder Leitungen auf der Platine vorhanden sind. Mögliche Anwendungen wären z.B. das Übertragen von *Überwachungsinformationen* des Schaltungblocks oder das *Debuggen* und *Testen*. Mit den in dieser Arbeit vorgestellten Verfahren lassen sich Übertragungsraten bis zu 500 kbit/s realisieren. Schließlich lassen sich alle diese Wasserzeichen- und Identifikationsverfahren in einem *Framework* zusammenfassen, welches in den bestehenden Entwicklungsumgebungen integriert werden kann.

Weitere Forschungsrichtungen für die *Kontrollflussüberwachung* von eingebetteten RISC-Prozessoren sind die Minimierung des Speicheraufwands, die komplette Unterstützung von indirekten Sprüngen, sowie die Unterstützung von mehr Prozessorarchitekturen. Der Speicheraufwand kann durch hierarchische Überwachungsstrukturen, wie z.B. in [ARRJ06] und durch Kompressionsmethoden verringert werden. *Hierarchische Überwachungseinheiten* verwenden relative Speicheradressen, welche zusätzlich durch Kompressionsverfahren weit weniger Speicher benötigen als die heutigen Methoden. Des Weiteren kann durch die Implementierung des Cache-Verfahrens der Bedarf des teuren On-chip-Speichers verringert werden. In diesem Verfahren werden die Informationen, die zur Überwachung benötigt werden, im Systemspeicher abgelegt und zum Überwachen einer gerade ausgeführten Funktion oder Teil des Programms in einen Cache-artigen On-chip-Speicher geladen. Das Wiederauffüllen des Caches mit neuen Daten kann durch das Betriebssystem oder selbstständig von der Überwachungseinheit veranlasst werden. Auch die Verfahren zur Überwachung von indirekten Sprüngen können von den Speicherreduktionstechniken profitieren. Die automatische Identifizierung von indirekten Sprungzielen im Progammcode ist auch ein mögliches zukünftiges Forschungsgebiet. Schließlich sollte auch die Portierung der hier vorgestellten Überwachungseinheit auf anderen Prozessorarchitekturen untersucht werden. Vor allem *Single-Instruction, Multiple-Data* (SIMD) Architekturen und Multiprozessorsysteme, bei denen das Programm synchron parallel ausgeführt wird, eignen sich für unseren Überwachungsansatz, da in diesen Architekturen nur eine Überwachungseinheit mehrere Ausführungseinheiten kontrollieren kann.

Die in dieser Arbeit vorgestellten Wasserzeichenverfahren sind für den FPGA-Entwurfsfluss entwickelt worden, während die Methoden zur Verbesserung der Zuverlässigkeit eher für den ASIC-Entwurf ausgelegt sind. Eine Kombination und Anpassung dieser Methoden um die Zuverlässigkeit von FPGA-Schaltungen zu verbessern könnte auch ein interessantes zukünftiges Forschungsgebiet sein.

Bibliography

[AAN00] Lorena Anghel, Dan Alexandrescu, and Michael Nicolaidis. Evaluation of a Soft Error Tolerance Technique based on Time and/or Space Redundancy. In *Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on*, pages 237–242, 2000.

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In *CHES '02: 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 29–45, London, UK, 2003. Springer-Verlag.

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity: Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):1–40, 2009.

[ABMF04] Todd Austin, David Blaauw, Trevor Mudge, and Krisztián Flautner. Making Typical Silicon Matter with Razor. *Computer*, 37(3):57–65, 2004.

[ADM04] Sobeeh Almukhaizim, Petros Drineas, and Yiorgos Makris. Concurrent Error Detection for Combinational and Sequential Logic via Output Compaction. In *ISQED '04: Proceedings of the 5th International Symposium on Quality Electronic Design*, pages 459–464, Washington, DC, USA, 2004. IEEE Computer Society.

[AGM+71] Algirdas Anthony Avizienis, GC Gilley, Francis Parkash Mathur, David Allen Rennels, JA Rohr, and DK Rubin. The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers*, 100(20):1312–1321, 1971.

[AHTA04] Amr T. Abdel-Hamid, Sofiéne Tahar, and El Mostapha Aboulhamid. A Survey on IP Watermarking Techniques. *Design Automation for Embedded Systems*, 9(3):211–227, 2004.

[Ajl95] Cheryl Ajluni. Two new Imaging Techniques Promise to Improve IC Defect Identification. *Electronic Design*, 43(14):37–38, 1995.

[AK96] Ross Anderson and Markus Kuhn. Tamper Resistance: A Cautionary Note. In *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–11, Berkeley, CA, USA, 1996. USENIX Association.

[Ale96] Aleph One. Smashing the Stack for Fun and Profit. *Phrack magazine*, 49(7), 1996.

[All00] VSI Alliance. Intellectual Property Protection White Paper: Schemes, Alternatives and Discussion Version 1.1. *Issued by Intellectual Property Protection Development Working Group, Ver*, 1.1, 2000.

[All07] Business Software Alliance. Fifth Annual BSA and IDC Global Software Piracy Study. Technical report, 2007.

[ALR01] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. *Technical Report Series – University of Newcastle upon Tyne Computing Science*, 2001.

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE transactions on dependable and secure computing*, pages 11–33, 2004.

[Alta] Altera. Error Detection and Recovery Using CRC in Altera FPGA Devices. *URL:* `http://www.altera.com/literature/an/an357.pdf`.

[Altb] Altera. FPGA Design Security Solution Using MAX II Devices. *URL:* `http://www.altera.com/literature/wp/wp_m2dsgn.pdf`.

[And72] James P. Anderson. Computer Security Technology Planning Study, 1972.

[ANKA99] Z. Alkhalifa, V. S. Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):627–641, 1999.

[AR] AT&T-Research. Graphviz - Graph Visualization Software. *URL:* `http://graphviz.org/`.

[ARM99] ARM. AMBA specification (rev 2.0). *ARM Limited*, 1999.

[ARRJ06] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Hardware-assisted Run-time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(12):1295–1308, 2006.

[ARS04] André Adelsbach, Markus Rohe, and Ahmad-Reza Sadeghi. Overcoming the Obstacles of Zero-knowledge Watermark Detection. In *MM&Sec '04: Proceedings of the 2004 workshop on Multimedia and security*, pages 46–55, New York, NY, USA, 2004. ACM.

[Aus95] Kenneth Austin. Data Security Arrangements for Semiconductor Programmable Devices, February 7 1995. US Patent 5,388,157.

[Aus99] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[BA02] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-signal VLSI Circuits*. Kluwer Academic, 2002.

[BAFS05] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized Instruction Set Emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[Bar] Scott Barrick. Designing Around an Encrypted Netlist: Is The Pain Worth the Gain? *D&R Industry Articles. URL:* `http://www.design-reuse.com/articles/18205/encrypted-netlist.html`.

[Bar81] Joel F. Bartlett. A NonStop Kernel. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, pages 22–29, New York, NY, USA, 1981. ACM.

[Bau05] Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005.

[Bau08] Florian Baueregger. Identifikation von signierten Schaltungen anhand von Leistungsverbrauchsmessungen. Dilpomarbeit, Department of Computer Science 12, University of Erlangen-Nuremberg, January 2008.

[BBB+97] S. Buchner, M. Baze, D. Brown, D. McMorrow, J. Melinger, SFA Inc, and MD Largo. Comparison of Error Rates in Combinational and Sequential Logic. *IEEE Transactions on Nuclear Science*, 44(6 Part 1):2209–2216, 1997.

[BBC05] BBC. 1986: Coal Mine Canaries made Redundant. *URL: http://news.bbc.co.uk/onthisday/hi/dates/stories/december/30/newsid_2547000/2547587.stm*, 2005.

[BDH+98] Feng Bao, Robert H. Deng, Yongfei Han, Albert B. Jeng, A. Desai Narasimhalu, and Teow-Hin Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 115–124, London, UK, 1998. Springer-Verlag.

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[BEA06] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural Support for Software-based Protection. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51, New York, NY, USA, 2006. ACM.

[BGB06] Lilian Bossuet, Guy Gogniat, and Wayne Burleson. Dynamically Configurable Security for SRAM FPGA Bitstreams. *International Journal of Embedded Systems*, 2(1):73–85, 2006.

[BGT93] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. In *Proc. 1993 International Conference on Communications (ICC '93)*, pages 1064–1070, Geneva, Switzerland, May 1993.

[BGXC07] Fujun Bai, Zhiqiang Gao, Yi Xu, and Xueyu Cai. A Watermarking Technique for Hard IP Protection in Full-custom IC Design. In *International Conference on Communications, Circuits and Systems (ICCCAS 2007)*, pages 1177–1180, 2007.

[BK00] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack Magazine*, 2000.

[BKIL03] Saurabh Bagchi, Zbigniew Kalbarczyk, Ravishankar Iyer, and Y. Levendel. Design and Evaluation of Preemptive Control Signature (PECOS) Checking. *IEEE Transactions on Computers*, 2003.

[BLW+01] Saurabh Bagchi, Y Liu, Keith Whisnant, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Y. Levendel, and Larry Votta. A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 225–234, Washington, DC, USA, 2001. IEEE Computer Society.

[BM07] Jan A. Bergstra and C. A. Middelburg. Instruction Sequences with Indirect Jumps. *Electronic Report PRG0709, Programming Research Group, University of Amsterdam*, 2007.

[Bor05] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software-Practice Experience*, 30(7):775–802, 2000.

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions go Bad: Generalizing Return-oriented Programming to RISC. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, New York, NY, USA, 2008. ACM.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 513–525, London, UK, 1997. Springer-Verlag.

[BSO07] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(2):8, 2007.

[BSOS04] Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems*

and Networks, page 51, Washington, DC, USA, 2004. IEEE Computer Society.

[BSR09] Daniel Baldin, Katharina Stahl, and Franz Rammig. Ergebnisbericht über Selbstorganisierende Betriebssysteme für autonome Integrierte Schaltungen. *AIS Meilensteinbericht UPB-HNI-Q12*, 2009.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Runtime Defense against Stack Smashing Attacks. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, USA, 2000. USENIX Association.

[BTH96] Laurence Boney, Ahmed H. Tewfik, and Khaled N. Hamdy. Digital Watermarks for Audio Signals. In *International Conference on Multimedia Computing and Systems*, pages 473–480, 1996.

[BWWA06] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-Based Transparent and Comprehensive Control-Flow Error Detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[BZS+06] Abdelmajid Bouajila, Johannes Zeppenfeld, Walter Stechele, Andreas Herkersdorf, Andreas Bernauer, Oliver Bringmann, and Wolfgang Rosenstiel. Organic Computing at the System on Chip Level. In *Proceedings of the IFIP International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC 2006)*. Springer, October 2006.

[BZSH09] Abdelmajid Bouajila, Johannes Zeppenfeld, Walter Stechele, and Andreas Herkersdorf. Multi-bit Soft-and Timing Error Detection for CPU Pipelines. In *Proceedings of edaWorkshop 09*. VDE VERLAG GmbH, 2009.

[CBB+01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[CBD+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, 1999.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[CD00] Roy Chapman and Tariq S. Durrani. IP Protection of DSP Algorithms for System on Chip Implementation. *IEEE Transactions on Signal Processing*, 48(3):854–861, 2000.

[CE99] Cristina Cifuentes and Mike Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, pages 192–199, Washington, DC, USA, 1999. IEEE Computer Society.

[CH01] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: A Compile-time Solution to Buffer Overflow Attacks. In *International Conference on Distributed Computing Systems*, volume 21, pages 409–420. IEEE Computer Society; 1999, 2001.

[Cha75] K. Mani Chandy. A Survey of Analytic Models of Rollback and Recovery Stratergies. *Computer*, 8(5):40–47, 1975.

[Cha98] Edoardo Charbon. Hierarchical Watermarking in IC Design. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 295–298, 1998.

[CHP97] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target Prediction for Indirect Jumps. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 25(2):274–283, 1997.

[CK06] Nathaniel Couture and Kenneth B. Kent. Periodic Licensing of FPGA based Intellectual Property. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 234–234, New York, NY, USA, 2006. ACM.

[CNV96] T. Calin, Michael Nicolaidis, and Raoul Velazco. Upset Hardened Memory Design for Submicron CMOS Technology. *IEEE Transactions on Nuclear Science*, 43(6 Part 1):2874–2878, 1996.

[Con99] Matt Conover. w00w00 on Heap Overflows. *URL:* `http://www.w00w00.org/files/articles/heaptut.txt`, 1, 1999.

[CPG+06] Encarnacion Castillo, Luis Parrilla, Antonio Garcia, Antonio Loris, and Uwe Meyer-Baese. IPP Watermarking Technique for IP Core Protection on FPL Devices. In *International Conference on Field Pro-*

grammable Logic and Applications, 2006. FPL'06, pages 487–492, 2006.

[CPG+08] Encarnacion Castillo, Luis Parrilla, Antonio Garcia, Uwe Meyer-Baese, Guillermo Botella, and Antonio Lloris. Automated Signature Insertion in Combinational Logic Patterns for HDL IP Core Protection. In *4th Southern Conference on Programmable Logic, 2008*, pages 183–186, 2008.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, Berkeley, CA, USA, 1998. USENIX Association.

[CPW74] J. R. Connet, E. J. Pasternak, and B. D. Wagner. Software Defenses in Real-time Control Systems. *Digest of papers*, page 94, 1974.

[CRH90] P. J. Clarke, A. K. Ray, and C. A. Hogarth. Electromigration–A Tutorial Introduction. *International Journal of Electronics*, 69(3):333–338, 1990.

[Cro79] Dwight L. Crook. Method of Determining Reliability Screens for Time Dependent Dielectric Breakdown. *Reliability Physics Symposium, 1979. 17th Annual*, pages 1–7, 1979.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[CSW89] Fan R. K. Chung, Jawad A. Salehi, and Victor K. Wei. Optical Orthogonal Codes: Design, Analysis and Applications. *IEEE Transactions on Information Theory*, 35(3):595–604, 1989.

[Dau06] Andrew Dauman. An Open IP Encryption Flow Permits Industry-wide Interoperability. *Synopsys, Inc. White Paper*, June 2006.

[Des97] Solar Designer. Non-executable Stack Patch. *URL:* `http://www.openwall.com/linux/`, 1997.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[Dig] Digilent. Spartan-3 Starter Board. *URL:* `http://www.digilentinc.com/`.

[DMW98] J. H. Daniel, D. F. Moore, and J. F. Walker. Focused Ion Beams for Microfabrication. *Engineering Science and Education Journal*, 7(2):53–56, 1998.

[Dob03] Igor Dobrovitski. Exploit for CVS Double free() for Linux pserver. *Neohapsis Archives (`http://www.security-express.com/archives/fulldisclosure/2003-q1/0545.html`)*, 2003.

[Dri09] Saar Drimer. Security for Volatile FPGAs. November 2009.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *ACM SIGPLAN Notices*, 38(5):155–167, 2003.

[DS90] X. Delord and Gabriele Saucier. Control Flow Checking in Pipelined RISC Microprocessors: the Motorola MC88100 Case Study. In *Proceedings of the Euromicro'90 Workshop on Real Time*, pages 162–169, 1990.

[DS91] X. Delord and Gabriele Saucier. Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Multiprocessors. In *Proceedings of the IEEE International Test Conference on Test*, pages 936–945, Washington, DC, USA, 1991. IEEE Computer Society.

[DS99] Debatosh Debnath and Tsutomu Sasao. Fast Boolean Matching under Permutation using Representative. In *Proceedings of the ASP-DAC'99 Asia and South Pacific Design Automation Conference, 1999*, pages 359–362, 1999.

[DSG05] Nij Dorairaj, Eric Shiflet, and Mark Goosman. PlanAhead Software as a Platform for Partial Reconfiguration. *Xilinx XCELL Journal, Art*, 55:68–71, 2005.

[DTP+09] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, Wei-Hsiang Ma, Sudherssen Kalaiselvan, Kevin Lai, David M. Bull, and David T. Blaauw. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):32–48, 2009.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[EKD+03] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[EL02] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[Erl07] Úlfar Erlingsson. Low-level Software Security: Attacks and Defenses. *Foundations of Security Analysis and Design IV*, 4677:92, 2007.

[ES84] James B. Eifert and John Paul Shen. Processor Monitoring Using Asynchronous Signatured Instruction Streams. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,'Highlights from Twenty-Five Years', Reprinted from FTGS-14 1984*, pages 394–399, 1984.

[Est] Chip Estimate. ChipEstimate.com. *URL:* `http://www.chipestimate.com/`.

[Fed01] Federal Information Processing Standards Publication 197. Announcing the Advanced Encryption Standard (AES). *URL:* `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[FKF+03] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol – Version 3.0. *URL:* `http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt`, 1996.

[Fra] Fraunhofer IISB. FIB Focused Ion Beam - Anwendungsbeispiele, Spezifikationen und Prinzip. *URL:* `http://www.iisb.fraunhofer.de/de/arb_geb/technologie_an_fib.htm`.

[Fra95] Matthew Franklin. A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. In *DFT '95: Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, page 207, Washington, DC, USA, 1995. IEEE Computer Society.

[FS01] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 55–66, Berkeley, CA, USA, 2001. USENIX Association.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[FT03] Y. C. Fan and H. W. Tsao. Watermarking for Intellectual Property Protection. *Electronics Letters*, 39(18):1316–1318, 2003.

[Gaia] Gaisler Research. GR-CPCI-XC4V LEON Compact-PCI Development board. *URL:* `http://www.gaisler.com`.

[Gaib] Gaisler Research. LEON3 SPARC V8 Processor core. *URL:* `http://www.gaisler.com`.

[Gai94] Jiri Gaisler. Concurrent Error-detection and Modular Fault-tolerance in a 32-bit Processing Core for Embedded Space Flight Applications. In *Twenty-Fourth International Symposium on Fault-Tolerant Computing FTCS, 1994*, pages 128–130. IEEE Computer Society Press, 1994.

[Gai02] Jiri Gaisler. A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 409–415, Washington, DC, USA, 2002. IEEE Computer Society.

[GCvDD02] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160, New York, NY, USA, 2002. ACM.

[GDWL92] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[Ger91] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[GHB07] Chloe Guerin, Vincent Huard, and Alain Bravaix. The Energy-Driven Hot-Carrier Degradation Modes of nMOSFETs. *IEEE Transaction on Device and Materials Reliability*, 7(2):225–235, 2007.

[GHJM05] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A Type-safe Dialect of C. *C/C++ Users Journal*, 23(1):112–139, 2005.

[GKST07] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. FPGA Intrinsic PUFs and Their Use for IP Protection. In *CHES '07: Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, pages 63–80, Berlin, Heidelberg, 2007. Springer-Verlag.

[GO98] Anup K. Ghosh and Tom O'Connor. Analyzing Programs for Vulnerability to Buffer Overrun Attacks. In *Proceedings of the 21st National Information Systems Security Conference, Crystal City, VA*, pages 274–382, 1998.

[Gor96] Michael B. Gordy. GA.M: A Matlab Routine for Function Maximization using a Genetic Algorithm. `ftp://all.repec.org/RePEc/cod/html/Matlab/gordy.m`, 1996.

[Gra] Mentor Graphics. Precision Synthesis. *URL:* `http://www.mentor.com/products/fpga/synthesis/precision_rtl/upload/PrecisionFamilyAug2007.pdf`.

[GRE+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[GRRV03] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Soft-Error Detection Using Control Flow Assertions. In *DFT '03: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588, Washington, DC, USA, 2003. IEEE Computer Society.

[GRRV05] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Improved Software-based Processor Control-flow Errors Detection Technique. In *Reliability and maintainability symposium*, pages 583–589, 2005.

[GSB+04] Matthew J. Gadlage, Ronald D. Schrimpf, Joseph M. Benedetto, Paul H. Eaton, David G. Mavis, Mike Sibley, Keith Avery, and Thomas L. Turflinger. Single Event Transient Pulse Widths in Digital Microcircuits. *IEEE Transactions on Nuclear Science*, 51(6 Part 2):3285–3290, 2004.

[GSS64] Izrail Moissevich Gel'fand, Georgi E. Shilov, and Eugene Saletan. *Generalized Functions*. Academic Press New York, 1964.

[GSVP03] Mohamed A. Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault Recovery for Chip Multiprocessors. *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, 31(2):98–109, 2003.

[GV05] Mohamed A. Gomaa and T. N. Vijaykumar. Opportunistic Transient-Fault Detection. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 172–183, Washington, DC, USA, 2005. IEEE Computer Society.

[GWA79] C. S. Guenzer, E. A. Wolicki, and R. G. Allas. Single Event Upset of Dynamic RAMs by Neutrons and Protons. *IEEE Transactions on Nuclear Science*, 26(6):5048–5052, 1979.

[Hag] Hagai Bar-El, Discretix Technologies Ltd. Known Attacks Against Smartcards. *URL:* `http://www.discretix.com/PDF/KnownAttacksAgainstSmartcards.pdf`.

[Har65] Michael A. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill, 1965.

[HB03] Eric Haugh and Matt Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, volume 2. Citeseer, 2003.

[HBF07] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In *Proceedings of the Conference on RFID Security*. Citeseer, 2007.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[HJ92] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, volume 136, 1992.

[HKW+03] Peter Hazucha, Tanay Karnik, Steven Walstra, Bradley Bloechel, James Tschanz, Jose Maiz, Krishnamurthy Soumyanath, Greg Dermer, Siva Narendra, Vivek De, and Shekhar Borkar. Measurements and analysis of SER tolerant latch in a 90 nm dual-Vt CMOS process. In *Proceedings of the IEEE Custom Integrated Circuits Conference, 2003.*, pages 617–620, 2003.

[HP99] Inki Hong and Miodrag Potkonjak. Behavioral Synthesis Techniques for Intellectual Property Protection. In *Design Automation Conference*, pages 849–854, 1999.

[Hüt03] Markus Hütter. *Logic Synthesis with Complex Gates*. dissertation, 2003.

[IBM] IBM. Rational Purify. *URL:* `http://www-01.ibm.com/software/awdtools/purify/`.

[IFI90] IFIP WG. Dependability: Basic Concepts and Terminology: in English, French, German, Italian and Japanese: IFIP WG 10.4. *Dependable Computing and Fault Tolerance*, 1990.

[ISO05] ISO JTC. 1/SC 27: Information Technology–Security Techniques–Code of Practice for Information Security Management. 2005.

[ITR05] ITRS. International Technology Roadmap for Semiconductors, 2005 Edition, URL: `http://www.itrs.net/Links/2005ITRS/Home2005.htm`. Technical report, 2005.

[ITR07] ITRS. International Technology Roadmap for Semiconductors, 2007 Edition, URL: `http://www.itrs.net/Links/2007ITRS/Home2007.htm`. Technical report, 2007.

[ITS91] ITSEC. Information Technology Security Evaluation Criteria (ITSEC). *Office for Official Publications of the European Communities*, 1991.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs. *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference*

on USENIX Annual Technical Conference, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[JMKP07] José A. Joao, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Dynamic Predication of Indirect Jumps. *IEEE Computer Architecture Letter*, 6(2):25–28, 2007.

[Joh00] Allan H. Johnston. Scaling and Technology Issues for Soft Error Rates. In *4th Annual Research Conference on Reliability, Stanford University*, 2000.

[JWSK06] Nikhil Joshi, Kaijie Wu, Jayachandran Sundararajan, and Ramesh Karri. Concurrent Error Detection for Involutional Functions with applications in Fault Tolerant Cryptographic Hardware Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1163–1169, 2006.

[JYPQ03] Adarsh K. Jain, Lin Yuan, Pushkin R. Pari, and Gang Qu. Zero Overhead Watermarking Technique for FPGA Designs. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 147–152. ACM Press, 2003.

[KBT08] Dirk Koch, Christian Beckhoff, and Jürgen Teich. ReCoBus-Builder - a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, pages 119–124, Heidelberg, Germany, September 2008.

[KC08] Kris Kaspersky and Alice Chang. Remote Code Execution through Intel CPU Bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*, 2008.

[KE91] David R. Kaeli and Philip G. Emma. Branch History Table Prediction of Moving Target Branches due to Subroutine Returns. *SIGARCH Computer Architecture News*, 19(3):34–42, 1991.

[Kea01] Tom Kean. Secure Configuration of a Field Programmable Gate Array. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 259–260, Washington, DC, USA, 2001. IEEE Computer Society.

[Kes00] David Kessner. Copy Protection for SRAM based FPGA Designs. *Application Note, Free IP Project, URL:* `http://web.archive.org/web/20031010002149/http://free-ip.com/copyprotection.html`, 2000.

[KHPC98] Darko Kirovski, Yean-Yow Hwang, Miodrag Potkonjak, and Jason Cong. Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 194–198, New York, NY, USA, 1998. ACM.

[KHT08] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient Reconfigurable On-Chip Buses for FPGAs. In *16th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)*, pages 287–290. IEEE Computer Society, April 2008.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.

[KK99] Oliver Kömmerling and Markus G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 9–20, 1999.

[KK07] Israel Koren and C. Mani Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[KLMR04] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new Dimension in Embedded System Design. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 753–760, New York, NY, USA, 2004. ACM. Moderator-Ravi, Srivaths.

[KLMS+98] Andrew Byun Kahng, John Lach, William Henry Mangione-Smith, Stefanus Mantik, Igor Leonidovich Markov, Miodrag M. Potkonjak, Paul Askeland Tucker, Huijuan Wang, and Gregory Wolfe. Watermarking Techniques for Intellectual Property Protection. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 776–781, New York, NY, USA, 1998. ACM.

[KLMS+01] Andrew Byun Kahng, John Lach, William Henry Mangione-Smith, Stefanus Mantik, Igor Leonidovich Markov, Miodrag M. Potkonjak, Paul Askeland Tucker, Huijuan Wang, and Gregory Wolfe. Constraint-Based Watermarking Techniques for Design IP Protection. *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20(10):1236–1252, 2001.

[klo99] klog. The Frame Pointer Overwrite. *Phrack magazine*, 55(9), 1999.

[KMM+98] Andrew Byun Kahng, Stefanus Mantik, Igor Leonidovich Markov, Miodrag M. Potkonjak, Paul Askeland Tucker, Huijuan Wang, and Gregory Wolfe. Robust IP Watermarking Methodologies for Physical Design. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 782–787, New York, NY, USA, 1998. ACM.

[KMM08] Tom Kean, David McLaren, and Carol Marsh. Verifying the Authenticity of Chip Designs with the DesignTag System. In *HOST '08: Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 59–64, Washington, DC, USA, 2008. IEEE Computer Society.

[KNKA96] Ghani A. Kanawati, V. S. Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Evaluation of Integrated System-level Checks for on-line Error Detection. In *IPDS '96: Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, pages 292–301, Washington, DC, USA, 1996. IEEE Computer Society.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.

[Koh08] Christian Kohn. Integration of Concepts for Assuring Correct Program Execution at Control and Data Path Level for Embedded RISC-Processors. Projektarbeit, Department of Computer Science 12, University of Erlangen-Nuremberg, October 2008.

[Kop97] Hermann Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.

[Kot06] Arun Kottolli. The Economics of Structured- and Standard-cell-ASIC Designs. *EDN Magazine*, 51(6):61–68, 2006.

[KP98] Darko Kirovski and Miodrag Potkonjak. Intellectual Property Protection Using Watermarking Partial Scan Chains For Sequential Logic Test Generation. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[KR06] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM.

[Kre03] Andreas Krennmair. ContraPolice: A libc Extension for Protecting Applications from Heap-smashing Attacks, 2003.

[KT05] M. Moiz Khan and Spyros Tragoudas. Rewiring for Watermarking Digital Circuit Netlists. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1132–1137, 2005.

[Lal01] Parag K. Lala, editor. *Self-checking and Fault-tolerant Digital Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[LB94] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software-Practice and Experience*, 24(2):197–218, 1994.

[LBD+04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, 2004.

[LBMC94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[LC02] Kyung-suk Lhee and Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88, Berkeley, CA, USA, 2002. USENIX Association.

[LC06] Qiming Li and Ee-Chien Chang. Zero-knowledge Watermark Detection Resistant to Ambiguity Attacks. In *MMSec '06: Proceedings of the 8th workshop on Multimedia and security*, pages 158–163, New York, NY, USA, 2006. ACM.

[LD03] Tri Van Le and Yvo Desmedt. Cryptanalysis of UCLA Watermarking Schemes for Intellectual Property Protection. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 213–225, London, UK, 2003. Springer-Verlag.

[LKMS04] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. *Security in Pervasive Computing*, pages 237–252, 2004.

[LLG+04] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srini Devadas. A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications. In *Proceedings of the IEEE VLSI Circuits Symposium*, pages 176–179. Citeseer, 2004.

[LLG+05] Daihyun Lim, Jae W. Lee, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srini Devadas. Extracting Secret Keys from Integrated Circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 13(10):1200, 2005.

[LMS06] Qiming Li, Nasir Memon, and Husrev T. Sencar. Security Issues in Watermarking Applications – A Deeper Look. In *MCPS '06: Proceedings of the 4th ACM international workshop on Contents protection and security*, pages 23–28, New York, NY, USA, 2006. ACM.

[LMSP98] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Signature Hiding Techniques for FPGA Intellectual Property Protection. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 186–189, New York, NY, USA, 1998. ACM.

[LMSP99] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Robust FPGA Intellectual Property Protection through Multiple Small Watermarks. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 831–836, New York, NY, USA, 1999. ACM.

[LMSP01] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Fingerprinting Techniques for Field-Programmable Gate Array Intellectual Property Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 20, 2001.

[LTRN92] Jien-Chung Lo, Suchai Thanawastien, T. R. N. Rao, and Michael Nicolaidis. An SFS Berger Check Prediction ALU and its Application to Self-checking Processor Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(4):525–540, 1992.

[Lu82] David J. Lu. Watchdog Processors and Structural Integrity Checking. *IEEE Transaction on Computers*, 31(7):681–685, 1982.

[MAF+99] M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi. RAS Strategy for IBM S/390 G5 and G6. *IBM Journal of Research and Development*, 43(5):875–888, 1999.

[MAS+07] Mojtaba Mehrara, Mona Attariyan, Smitha Shyam, Kypros Constantinides, Valeria Bertacco, and Todd Austin. Low-cost Protection for SER Upsets and silicon Defects. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1146–1151, San Jose, CA, USA, 2007. EDA Consortium.

[MBS07] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[MBS08] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. *IEEE Micro-Institute of Electrical and Electronics Engineers*, 28(1):52–59, 2008.

[MdR99] Todd C. Miller and Theo de Raadt. strlcpy and strlcat: Consistent, Safe, String Copy and Concatenation. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 1999. USENIX Association.

[ME02] David G. Mavis and Paul H. Eaton. Soft Error Rate Mitigation Techniques for Modern Microcircuits. In *40th Annual Reliability Physics Symposium Proceedings*, pages 216–225, 2002.

[Mey71] John F. Meyer. Fault Tolerant Sequential Machines. *IEEE Transactions on Computers*, 20(10):1167–1177, 1971.

[MH91] Edgar Michel and Wolfgang Hohl. Concurrent Error Detection using Watchdog Processors in the Multiprocessor System MEMSY. In *Fault-tolerant computing systems: tests, diagnosis, fault treatment: 5th International GI/ITG/GMA Conference, Nürnberg, September 25-27, 1991: proceedings*, page 54. Springer, 1991.

[MHPS96] István Majzik, Wolfgang Hohl, András Pataricza, and Volker Sieh. Multiprocessor Checking using Watchdog Processors. *Computer Systems Science and Engineering*, 11(5):301–310, 1996.

[Mic03] Giovanni De Micheli. Designing Robust Systems with Uncertain Information. In *Asia and South Pacific Design Automation Conference (ASPDAC 03)*, 2003.

[MKGT92] Ghassem Miremadi, Johan Karlsson, Ulf Gunneflo, and Jan Torin. Two Software Techniques for On-line Error Detection. In *Digest of Papers,*

Twenty-Second International Symposium on Fault-Tolerant Computing. FTCS-22., pages 328–335, 1992.

[MKP09] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Techniques for Design and Implementation of Secure Reconfigurable PUFs. *ACM Transaction on Reconfigurable Technology Systems*, 2(1):1–33, 2009.

[MKSL03] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A Processor Architecture Defense against Buffer Overflow Attacks. In *Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, pages 243–250. Citeseer, 2003.

[MLS91] Thierry Michel, Régis Leveugle, and Gabriele Saucier. A New Approach to Control Flow Checking Without Program Modification. In *Digest of Papers of Twenty-First International Symposium of Fault-Tolerant Computing*, pages 334–343, 1991.

[MM88] Aamer Mahmood and Edward J. McCluskey. Concurrent Error Detection Using Watchdog Processors-A Survey. *IEEE Transaction on Computers*, 37(2):160–174, 1988.

[MM00] Subhasish Mitra and Edward J. McCluskey. Which Concurrent Error Detection Scheme to Choose? In *ITC '00: Proceedings of the 2000 IEEE International Test Conference*, pages 985–994, Washington, DC, USA, 2000. IEEE Computer Society.

[MN98] Lee D. McFearin and V. S. Sukumaran Nair. Control Flow Checking Using Assertions. *Dependable Computing and Fault Tolerant Systems*, 10:183–200, 1998.

[MP00] Seapahn Meguerdichian and Miodrag Potkonjak. Watermarking while Preserving the Critical Path. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 108–111, New York, NY, USA, 2000. ACM.

[MS91] Henrique Madeira and João G. Silva. On-line Signature Learning and Checking: Experimental Evaluation. In *CompEuro'91: Proceedings of the 5th Annual European Computer Conference of Advanced Computer Technology, Reliable Systems and Applications*, pages 642–646, 1991.

[MSZ+05] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust System Design with Built-In Soft-Error Resilience. *Computer*, 38(2):43–52, 2005.

[MV05] Matt Messier and John Viega. Safe C String Library v1.0.3. *URL: http://www.zork.org/safestr/*, 2005.

[MW04] Ritesh Mastipuram and Edwin C. Wee. Soft errors' impact on system reliability. *Electrical Design News*, 49:69–76, 2004.

[MW08] Matthias May and Norbert Wehn. Modelle für Demonstrator übergeben. *AIS Meilensteinbericht TUK-EMS-Q09*, 2008.

[MWB+10] Matthias May, Norbert Wehn, Abdelmajid Bouajila, Johannes Zeppenfeld, Walter Stechele, Andreas Herkersdorf, Daniel Ziener, and Jürgen Teich. A Rapid Prototyping System for Error-Resilient Multi-Processor Systems-on-Chip. In *Proceedings of DATE'10*, pages 375–380, March 2010.

[MZS+08] Subhasish Mitra, Ming Zhang, Norbert Seifert, T. M. Mak, and Kee Sup Kim. Soft error resilient system design through error correction. *IFIP International Federation for Information Processing*, 249:143–156, 2008.

[NAB03] Michael Nicolaidis, Nadir Achouri, and Slimane Boutobza. Dynamic Data-bit Memory Built-In Self- Repair. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 588, Washington, DC, USA, 2003. IEEE Computer Society.

[Nam82] Masood Namjoo. Techniques for Concurrent Testing of VLSI Processor Operation. In *Proceedings of International Test Conference*, pages 461–468, 1982.

[Nam83] Massod Namjoo. CERBERUS-16: An Architecture for a General Purpose Watchdog Processor. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 216–219, 1983.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[NDMF97] Michael Nicolaidis, Ricardo O. Duarte, Salvador Manich, and Joan Figueras. Fault-Secure Parity Prediction Arithmetic Operators. *IEEE Design & Test of Computers*, 14(2):60–71, 1997.

[Nic93] Michael Nicolaidis. Efficient Implementations of Self-checking Adders and ALUs. In *FTCS-23: Digest of Papers of the Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 586–595, 1993.

[Nic99] Michael Nicolaidis. Time Redundancy based Soft-error Tolerance to Rescue Nanometer Technologies. In *Proceedings of the 17th IEEE VLSI Test Symposium*, pages 86–94, 25-29 April 1999.

[NKIX04] Nithin M. Nakka, Zbigniew T. Kalbarczyk, Ravi K. Iyer, and J. Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 585–594, Washington, DC, USA, 2004. IEEE Computer Society.

[NMCB97] Jeffrey M. Nick, Brian B. Moore, Jen Yao Yao Chung, and Nicholas S. Bowen. S/390 Cluster Technology: Parallel Sysplex. *IBM Systems Journal*, 36(2):172–201, 1997.

[NNC+01] Naveen Narayan, Rexford D. Newbould, Jo Dale Carothers, Jefrey J. Rodriguez, and W. Timothy Holman. IP Protection for VLSI Designs via Watermarking of Routes. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, pages 406–410, 2001.

[NX06] Vijaykrishnan Narayanan and Yuan Xie. Reliability Concerns in Embedded System Designs. *Computer*, 39(1):118–120, 2006.

[OCK+75] Severo M. Ornstein, William R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and Frank E. Heart. Pluribus: A Reliable Multiprocessor. In *AFIPS '75: Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 551–559, New York, NY, USA, 1975. ACM.

[Oli01] Arlindo L. Oliveira. Techniques for the Creation of Digital Watermarks in Sequential Circuit Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1101–1117, 2001.

[OMM02] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Transaction on Computers*, 51(2):180–199, 2002.

[Opea] Opencores.org. Basic DES Crypto Core. *URL:* `http://www.opencores.org/project,basicdes`.

[Opeb] Opencores.org. Keyboardcontroller. *URL:* `http://www.opencores.org/project,keyboardcontroller`.

[Opec] Opencores.org. Opencores. *URL:* `http://www.opencores.org`.

[OR95] Joakim Ohlsson and Marcus Rimen. Implicit Signature Checking. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 218–227, Washington, DC, USA, 1995. IEEE Computer Society.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[OVB+06] Hilmi Özdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transaction on Computers*, 55(10):1271–1285, 2006.

[PAX03] PAX Team. Non Executable Data Pages. *URL:* `http://pax.grsecurity.net/docs/pax.txt`, 2003.

[PB04] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 02(4):20–27, 2004.

[PBA10] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-Based Attack of RSA Authentication. In *Proceedings of Design and Test in Europe (DATE'10)*, pages 855–860, March 2010.

[PMHH93] András Pataricza, István Majzik, Wolfgang Hohl, and Joachim Hönig. Watchdog Processors in Parallel Systems. *Microprocessing and Microprogramming*, 39(2-5):69–74, 1993.

[PS95] John G. Proakis and Masoud Salehi. *Digital Communications*. McGraw-Hill New York, 1995.

[PV01] Matthias Pflanz and Heinrich Theodor Vierhaus. Online Check and Recovery Techniques for Dependable Embedded Processors. *IEEE Micro*, 21(5):24–40, 2001.

[QP00] Gang Qu and Miodrag Potkonjak. Fingerprinting Intellectual Property using Constraint-addition. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 587–592, New York, NY, USA, 2000. ACM.

[QP03] Gang Qu and Miodrag Potkonjak. *Intellectual Property Protection in VLSI Designs*. Kluwer Academic Publisher, 2003.

[Qu02] Gang Qu. Publicly Detectable Watermarking for Intellectual Property Authentication in VLSI Design. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 21(11):1363–1367, 2002.

[Rag06] Roshan G. Ragel. *Architectural Support for Security and Reliability in Embedded Processors*. PhD thesis, University of New South Wales, Sydney, Australia, 2006.

[RAMSP99] Azra Rashid, Jeet Asher, William H. Mangione-Smith, and Miodrag Potkonj. Hierarchical Watermarking for Protection of DSP Filter Cores. In *Proceedings of the Custom Integrated Circuits Conference. Piscataway, NJ*, pages 39–45. IEEE Press, 1999.

[RBD+01] Rob A. Rutenbar, Max Baron, Thomas Daniel, Rajeev Jayaraman, Zvi Or-Bach, Jonathan Rose, and Carl Sechen. (When) will FPGAs kill ASICs? (panel session). In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 321–322, New York, NY, USA, 2001. ACM.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[Reu] Design & Reuse. Catalyst of Collaborative IP Based SoC Design. *URL:* `http://www.design-reuse.com/`.

[RF89] Thammavarapu R. N. Rao and Eiji Fujiwara. *Error-control Coding for Computer Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[RHF01] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual Use of Superscalar Datapath for Transient-fault Detection and Recovery. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224, Washington, DC, USA, 2001. IEEE Computer Society.

[Ric02] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *URL:* `http://www1.corest.com/files/files/11/StackGuardPaper.pdf`, 2002.

[Ric08] Robert Richardson. CSI Computer Crime and Security Survey. Technical report, 2008.

[Riv92a] Ronald Linn Rivest. RFC 1321: The MD-5 Message Digest Algorithm. In *Internet Activities Board*, 1992.

[Riv92b] Ronald Linn Rivest. The RC4 Encryption Algorithm. In *RSA Data Security, Inc.*, 1992.

[RKMV03] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time Detection of Heap-based Overflows. In *LISA '03: Proceedings of the 17th USENIX conference on Large Installation Systems Administraton*, pages 51–60, Berkeley, CA, USA, 2003. USENIX Association.

[RL04] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[RLC+07] Eduardo L. Rhod, Calisboa A. Lisbôa, L. Carro, Massimo Violante, and Matteo Sonza Reorda. A Non-intrusive On-line Control Flow Error Detection Technique for SoCs. In *IEEE Latin-Americam Test Workshop, LATW*, volume 8, 2007.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *ACM SIGARCH Computer Architecture News*, 28(2):25–36, 2000.

[Rot99] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 84–93, Washington, DC, USA, 1999. IEEE Computer Society.

[RP06] Roshan G. Ragel and Sri Parameswaran. Hardware Assisted Preemptive Control Flow Checking for Embedded Processors to Improve Reliability. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 100–105, New York, NY, USA, 2006. ACM.

[RRKH04] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in Embedded Systems: Design Challenges. *ACM Transaction on Embedded Computer Systems*, 3(3):461–491, 2004.

[RRP06] Vimal K. Reddy, Eric Rotenberg, and Sailashri Parthasarathy. Understanding Prediction-based Partial Redundant Threading for Low-overhead, High-coverage Fault Tolerance. *ACM SIGOPS Operating Systems Review*, 40(5):83–94, 2006.

[RSA78] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RSR00] Faisal Rashid, Kewal K. Saluja, and Parameswaran Ramanathan. Fault Tolerance through Re-Execution in Multiscalar Architecture. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 482–491, Washington, DC, USA, 2000. IEEE Computer Society.

[SAH] Bassel Soudan, Wael Adi, and Abdulrahman Hanoun. Enabling Secure Integration of Multiple IP Cores in the Same FPGA. *D&R Industry Articles. URL:* `http://www.design-reuse.com/articles/21638/secure-integration-ip-cores-fpga.html`.

[Sal89] Jawad A. Salehi. Code Division Multiple-access Techniques in Optical Fiber Networks – Part I: Fundamental Principles. *IEEE Transactions on Communications*, 37(8):824–833, 1989.

[San] Sanyo. Quality and Reliability Handbook Ver. 3. *URL:* `http://www.semiconductor-sanyo.com/reliability/index.htm`.

[SB89] Jawad A. Salehi and Charles A. Brackett. Code Division Multiple-access Techniques in Optical Fiber Networks – Part II: Systems Performance Analysis. *IEEE Transactions on Communications*, 37(8):834–842, 1989.

[SBDB01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, Washington, DC, USA, 2001. IEEE Computer Society.

[SBE+07a] Walter Stechele, Oliver Bringmann, Rolf Ernst, Andreas Herkersdorf, Katharina Hojenski, Peter Janacik, Franz Rammig, Jürgen Teich, Norbert Wehn, Johannes Zeppenfeld, and Daniel Ziener. Autonomic MPSoCs for Reliable Systems. In *Proceedings of Zuverlässigkeit und Entwurf (ZuD 2007)*, pages 137–138, Munich, Germany, March 2007.

[SBE+07b] Walter Stechele, Oliver Bringmann, Rolf Ernst, Andreas Herkersdorf, Katharina Hojenski, Peter Janacik, Franz Rammig, Jürgen Teich, Norbert Wehn, Johannes Zeppenfeld, and Daniel Ziener. Concepts for Autonomic Integrated Systems. In *Proceedings of edaWorkshop07*, Munich, Germany, June 2007.

[SBH+09] Volker Schöber, Oliver Bringmann, Andreas Herkersdorf, Walter Stechele, Norbert Wehn, Matthias May, Daniel Ziener, Abdelmajid Bouajila, Daniel Baldin, Johannes Zeppenfeld, Björn Sanders, Jürgen Teich, Maurice Sebastian, Rolf Ernst, and Dieter Treytnar. AIS – Autonomous Integrated Systems. *newsletter edacentrum*, (04):05–13, 2009.

[Sch08] Moritz G. Schmid. Proof of Concept Exploit for Leon3 Core. Technical report, Universität Erlangen-Nürnberg, 2008.

[SD07] G. Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 9–14, New York, NY, USA, 2007. ACM.

[SE09] Maurice Sebastian and Rolf Ernst. Reliability Analysis of Single Bus Communication with Real-Time Requirements. In *PRDC '09: Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 3–10, Washington, DC, USA, November 2009. IEEE Computer Society.

[See99] Ralf Seepold. Special Session—Virtual Socket Interface Alliance. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, pages 182–182, New York, NY, USA, 1999. ACM.

[SFF+02] Oliverio J. Santana, Ayose Falcón, Enrique Fernández, Pedro Medina, Alex Ramírez, and Mateo Valero. A Comprehensive Analysis of Indirect Branch Prediction. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, pages 133–145, London, UK, 2002. Springer-Verlag.

[Sha07] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, New York, NY, USA, 2007. ACM.

[SHB68] Frederick F. Sellers, Mu-Yue Hsiao, and Leroy W. Bearnson. *Error Detecting Logic for Digital Computers*. McGraw-Hill, 1968.

[SKB02] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic Power Consumption in Virtex-II FPGA Family. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 157–164, New York, NY, USA, 2002. ACM Press.

[SM90] Nirmal Raj Saxena and Edward J. McCluskey. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Transactions on Computers*, 39(4):554–559, 1990.

[SPA] SPARC International, Inc. The SPARC Architecture Manual V8. *URL:* `http://www.sparc.com/standards/V8.pdf`.

[SPE] SPEC (Standard Performance Evaluation Corporation). SPEC CPU2000 V1.3. *URL:* `http://www.spec.org`.

[SPR00] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream Processors: Improving Both Performance and Fault Tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

[SS87] Michael A. Schuette and John Paul Shen. Processor Control Flow Monitoring using Signatured Instruction Streams. *IEEE Transaction on Computers*, 36(3):264–277, 1987.

[SS98] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems (3rd ed.): Design and Evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[SS06] Eric Simpson and Patrick Schaumont. Offline Hardware/Software Authentication for Reconfigurable Platforms. *Cryptographic Hardware and Embedded Systems (CHES 2006)*, pages 311–323, 2006.

[SSB09] Björn Sander, Jürgen Schnerr, and Oliver Bringmann. ESL Power Analysis of Embedded Processors for Temperature and Reliability Estimations. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 239–248, New York, NY, USA, 2009. ACM.

[SSK+06] Norbert Seifert, Paul Gregory Slankard, M. Kirsch, Balaj Narasimham, Victor Zia, Chris Brookreson, A. Vo, Subhasish Mitra, Balkaran Gill, and Jose A. Maiz. Radiation-Induced Soft Error Rates of Advanced CMOS Bulk Devices. In *Proceedings of the 44th Annual IEEE International Reliability Physics Symposium*, pages 217–225, 2006.

[SSK07] Debasri Saha and Susmita Sur-Kolay. Fast Robust Intellectual Property Protection for VLSI Physical Design. In *ICIT '07: Proceedings of the 10th International Conference on Information Technology*, pages 1–6, Washington, DC, USA, 2007. IEEE Computer Society.

[ST82] Thirumalai Sridhar and Satish M. Thatte. Concurrent Checking of Program Flow in VLSI Processors. In *Proceedings International Test Conference (ITC 1982), Philadelphia, PA, USA*, pages 191–199, 1982.

[ST87] John Paul Shen and Stephen P. Tomas. A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems. *Microprocessing and Microprogramming*, 20(4-5):249–269, 1987.

[SXZ+04] Zili Shao, Chun Xue, Qingfeng Zhuge, Edwin Hsing Mean Sha, and Bin Xiao. Security Protection and Checking in Embedded System Integration Against Buffer Overflow Attacks. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, pages 409–412, Washington, DC, USA, 2004. IEEE Computer Society.

[Syna] Synopsys. Synplify Premier. *URL:* `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/CapsuleModule/syn_prem_ds.pdf`.

[Synb] Synopsys. Synplify Pro. *URL:* `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/CapsuleModule/syn_pro_ds.pdf`.

[SZHS03] Zili Shao, Qingfeng Zhuge, Yi He, and Edwin Hsing Mean Sha. Defending Embedded Systems Against Buffer Overflow via Hardware/-Software. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, pages 352–361, Washington, DC, USA, 2003. IEEE Computer Society.

[SZT08] Moritz Schmid, Daniel Ziener, and Jürgen Teich. Netlist-Level IP Protection by Watermarking for LUT-Based FPGAs. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2008)*, pages 209–216, Taipei, Taiwan, December 2008.

[TC00] Ilhami Torunoglu and Edoardo Charbon. Watermarking-based Copyright Protection of Sequential Functions. *IEEE Journal of Solid-State Circuits*, 35(3):434–440, 2000.

[TH07] Jürgen Teich and Christian Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer, 2007.

[TM97] Nur A. Touba and Edward J. McCluskey. Logic Synthesis of Multilevel Circuits with Concurrent Error Detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(7):783–789, 1997.

[TT90] Yuval Tamir and Marc R. Tremblay. High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback. *IEEE Transactions on Computers*, 39(4):548–554, 1990.

[UR94] Shambhu J. Upadhyaya and Bina Ramamurthy. Concurrent Process Monitoring with No Reference Signatures. *IEEE Transaction on Computers*, 43(4):475–480, 1994.

[US-08] US-CERT. Vulnerability Notes Database CERT Coordination Center. *URL:* `http://www.kb.cert.org/vuls/`, 2008.

[VBKM00] John Viega, J. T. Bloch, Yoshi Kohno, and Gary E. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257, Washington, DC, USA, 2000. IEEE Computer Society.

[vdV04] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. *Red Hat, August*, 2004.

[Ven00] Vendicator. Stack Shield: A Stack Smashing Technique Protection Tool for Linux. *URL:* `http://www.angelfire.com/sk/stackshield/info.html`, 2000.

[VIS] VISENGI. VHDL Obfuscator & Watermarker. *URL:* `http://www.visengi.com/en/products/software/vhdl_obfuscator`.

[Vit90] Andrew J. Viterbi. Very Low Rate Convolutional Codes for Maximum Theoretical Performance of Spread-Spectrum Multiple-Access Channels. *IEEE Journal on Selected Areas in Communications*, 8(4):641–649, 1990.

[Vit95] Andrew J. Viterbi. *CDMA: Principles of Spread Spectrum Communication*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[VPC02] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-Fault Recovery using Simultaneous Multithreading. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 87–98, Washington, DC, USA, 2002. IEEE Computer Society.

[WA01] Chris Weaver and Todd M. Austin. A Fault Tolerant Approach to Microprocessor Design. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[WD01] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, Washington, DC, USA, 2001. IEEE Computer Society.

[WEMR04] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 264, Washington, DC, USA, 2004. IEEE Computer Society.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, 2000.

[Whe] David A. Wheeler. Flawfinder Home Page. *URL:* `http://www.dwheeler.com/flawfinder`.

[Wil07] Ron Wilson. Panel Unscrambles Intellectual Property Encryption Issues. *EDN Magazine URL:* `http://www.edn.com/article/CA6412249.html`, 2007.

[WLG+89] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton William Green, Karl N. Levitt, Peter Michael Milliar-Smith, Robert E. Shostak, and Charles Burr Weinstock. SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control. In *Tutorial: Hard Real-time Systems*, pages 560–575. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[Woj98] Rafal Wojtczuk. Defeating Solar Designer Nonexecutable Stack Patch. *Bugtraq mailinglist*, 1998.

[WP06] Nicholas J. Wang and Sanjay J. Patel. ReStore: Symptom-based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, 2006.

[Wri08] Craig Wright. Hacking Coffee Makers. *URL:* `http://www.securityfocus.com/archive/1/493387`, 2008.

[WS90] Kent Wilken and John Paul Shen. Continuous Signature Monitoring: Low-cost Concurrent Detection of Processor Control Errors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):629–641, 1990.

[Xila] Xilinx Inc. FPGA IFF Copy Protection Using Dallas Semiconductor/Maxim DS2432 Secure EEPROMs. *URL:* `http://www.xilinx.com/support/documentation/application_notes/xapp780.pdf`.

[Xilb] Xilinx Inc. ISE Design Suite Software Manuals and Help - PDF Collection These. *URL:* `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/manuals.pdf`.

[Xilc] Xilinx Inc. JBits 3.0 SDK for Virtex-II. *URL:* `www.xilinx.com/labs/projects/jbits/`.

[Xild] Xilinx Inc. MicroBlaze Processor Reference Guide. *URL:* `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`.

[Xile] Xilinx Inc. Protect Your Brand with Extended Spartan-3A FPGAs. *URL:* `http://www.xilinx.com/products/design_resources/security/devicedna.htm`.

[Xilf] Xilinx Inc. Virtex-II Platform FPGAs: Complete Data Sheet. *URL:* `http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf`.

[Xilg] Xilinx Inc. Xilinx Core Generator. *URL:* `http://www.xilinx.com/ise/products/coregen_overview.pdf`.

[Xilh] Xilinx Inc. Xilinx Virtex-II Pro Libraries Guide for HDL Designs. *URL:* `www.xilinx.com/itp/xilinx10/books/docs/virtex2p_hdl/virtex2p_hdl.pdf`.

[Xili] Xilinx Inc. XST User Guide. *URL:* `http://toolbox.xilinx.com/docsan/xilinx5/pdf/docs/xst/xst.pdf`.

[Xil03] Xilinx Inc. Next-Generation Virtex Family From Xilinx to top one Billion Transistor Mark. *URL:* `http://www.xilinx.com/prs_rls/silicon_vir/03131_nextgen.htm`, 2003.

[Xil05] Xilinx Inc. Virtex-II Platform FPGA User Guide (UG002). 2.0, pages 269–358. March 2005.

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 260–272, 2003.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture Support for Defending against Buffer Overflow Attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[YC80] Stephen S. Yau and Fu-Chung Chen. An Approach to Concurrent Control Flow Checking. *IEEE Transactions on Software Engineering*, 6(2):126–137, 1980.

[YP97] Lynn Youngs and Siva Paramanandam. Mapping and Repairing Embedded-Memory Defects. *IEEE Design and Test of Computers*, 14(1):18–24, 1997.

[ZAT06] Daniel Ziener, Stefan Aßmus, and Jürgen Teich. Identifying FPGA IP-Cores based on Lookup Table Content Analysis. In *Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 481–486, Madrid, Spain, August 2006.

[ZBT10a] Daniel Ziener, Florian Baueregger, and Jürgen Teich. Multiplexing Methods for Power Watermarking. In *Proceedings of the IEEE Int. Symposium on Hardware-Oriented Security and Trust (HOST 2010), Anaheim, USA*, June 2010.

[ZBT10b] Daniel Ziener, Florian Baueregger, and Jürgen Teich. Using the Power Side Channel of FPGAs for Communication. In *Proceedings of the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010)*, pages 237–244, May 2010.

[Zim95] Philip R. Zimmermann. *The official PGP user's guide*. MIT Press, Cambridge, MA, USA, 1995.

[ZT05] Daniel Ziener and Jürgen Teich. Evaluation of Watermarking Methods for FPGA-Based IP-cores. Technical Report 01-2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, D-91058 Erlangen, Germany, March 2005.

[ZT06] Daniel Ziener and Jürgen Teich. FPGA Core Watermarking Based on Power Signature Analysis. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2006)*, pages 205–212, Bangkok, Thailand, December 2006.

[ZT07a] Daniel Ziener and Jürgen Teich. Watermarking Apparatus, Software Enabling an Implementation of an Electronic Circuit Comprising a Watermark, Method for Detecting a Watermark and Apparatus for Detecting a Watermark Europäisches Patent EP1835425, Anmeldetag 17.03.2006, veröffentlicht 19.09.2007, Patentklassen (IPC) G06F 17/50; G06F 21/00, September 2007.

[ZT07b] Daniel Ziener and Jürgen Teich. Watermarking Apparatus, Software Enabling an Implementation of an Electronic Circuit Comprising a Watermark, Method for Detecting a Watermark and Apparatus for Detecting a Watermark. US-Patent US2007/0220263, Anmeldetag 19.10.2006 aus EP 1835425, veröffentlicht 20.09.2007, Patentklassen (IPC) H04L 9/00, September 2007.

[ZT08a] Daniel Ziener and Jürgen Teich. Concepts for Autonomous Control Flow Checking for Embedded CPUs. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*, pages 234–248, Oslo, Norway, June 2008.

[ZT08b] Daniel Ziener and Jürgen Teich. Power Signature Watermarking of IP Cores for FPGAs. *Journal of Signal Processing Systems*, 51(1):123–136, April 2008.

[ZT09] Daniel Ziener and Jürgen Teich. Concepts for Run-time and Error-resilient Control Flow Checking of Embedded RISC CPUs. *Int. Journal of Autonomous and Adaptive Communications Systems*, 2(3):256–275, July 2009.

[ZV96] Zeljko Zilic and Zvonko G. Vranesic. Using BDDs to design ULMs for FPGAs. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 24–30, New York, NY, USA, 1996. ACM Press.

[ZZPL04] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Hardware Supported Anomaly Detection: Down to the Control Flow Level. Technical report, Georgia Institute of Technology, 2004.

[ZZPL05] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous Path Detection with Hardware Support. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 43–54, New York, NY, USA, 2005. ACM.

Symbols

Adr_{LUTF} ..... address of LUTF
Adr_{LUTG} ..... address of LUTG
$\mathcal{A}$ ..... attacker
BB ..... set of basic blocks
BB_i ..... basic block
c ..... scaling factor
C ..... criterion value for HDL core identification
C_D ..... cost to develop a core
C_O ..... cost to obtain a core
C_P ..... cost to purchase a core
CF_{lut0} ..... first LUT frame in a CLB
$CFG(BB,T)$ ..... control flow graph
CFI ..... set of control flow instructions
CFI_i ..... control flow instruction
$CFIG(CFI,T)$ ..... control flow instruction graph
CY ..... number of CLB rows in an FPGA
d ..... LUT distance
D_{max} ..... maximal number of verified indirect jump destination in one clock cycle
d_{stack} ..... return stack depth
$\mathcal{D}$ ..... watermark or core detector
e_i ..... power event
$\mathcal{E}$ ..... watermark embedder
f ..... function
f_{clk} ..... clock frequency
f_{BPSK} ..... frequency of S_{BPSK}
F_{CLB} ..... number of CLB frames
F_{cy0} ..... frame number of the first CLB frame
F_{GCLK} ..... number of GCLK frames

F_{IOB} . . . . . number of IOB frames
F_{IOI} . . . . . number of IOI frames
$f_{lf}(x_S)$ . . . . . frame storing LUTs of x_S
FL . . . . . frame length
F_p . . . . . fitting value of p
f_u . . . . . number of different unique function
f_{wm} . . . . . signature bit rate
$\widehat{f}$ . . . . . predicted function
$\mathcal{G}$ . . . . . watermark generator
$h(t)$ . . . . . impulse response
i . . . . . index variable
I . . . . . work
I_A . . . . . author's work
I_B . . . . . bitfile core or design
I_L . . . . . core at logic level
I_P . . . . . fake original work
$\widetilde{I}$ . . . . . watermarked work
$\widetilde{I_A}$ . . . . . author's watermarked work
$\widetilde{I_B}$ . . . . . watermarked bitfile core
$\widetilde{I_L}$ . . . . . watermarked core at logic level
$\widetilde{I_P}$ . . . . . pirate's watermarked work
$\widehat{I}$ . . . . . published work
$\widehat{I}_{A_x}$ . . . . . a variant of author's watermarked work
$\mathcal{I}$ . . . . . universe of work elements
$\mathcal{I}_B$ . . . . . universe Bit
j . . . . . index variable
k . . . . . discrete point in time
K . . . . . key
K_A . . . . . author's key
K_c . . . . . number of applied additional constraints
K_{max} . . . . . signal length
K_x . . . . . extracted key
l . . . . . number of different frequency components of approx. $h(t)$
L_{eff} . . . . . physical gate length
lut_{L_i} . . . . . a LUT primitive cell in a netlist

$\mathcal{L}$ ..... lookup table extractor
n ..... word width
n_{crc} ..... bit width of the CRC
n_{cupc} ..... bit width of the CUPC
n_E ..... number of lookup table entries
n_i, α_i, f_i, ϕ_i ..... parameters for $h(t)$ approximation
n_{iJ} ..... number of all indirect jumps
n_l ..... number of LUT inputs
n_{max} ..... number of power watermarked cores
n_o ..... number of non-watermarked solutions
n_{pnrg} ..... bit width of the PNRG
n_s ..... number of signature repetitions in a signal
n_t ..... number of targets of an indirect jump
n_T ..... number of all indirect jump targets
n_w ..... number of watermarked solutions
$n(t)$ ..... noise signal
m ..... length of something
p ..... signature start position
P ..... equivalence class with input permutation
$P_{a\ in\ b}$ ..... percent of the unique functions of the core I_{LA} in I_{LB}
$P_{b\ in\ a}$ ..... percent of the unique functions of the core I_{LB} in I_{LA}
P_E ..... probability of successful decoding of a signal
P_f ..... probability that a fault is injected
P_c ..... probability of carry the watermark by chance
$\mathbf{p}_q$ ..... probability distribution of the unique functions of $\mathbf{q}$
$P(n,t)$ ..... density function of the χ^2_n-distribution
PC_n ..... current program counter
PC_{n+1} ..... next program counter
q ..... number of lookup tables in a netlist core
$\mathbf{q}$ ..... set of unique functions in a netlist core
q_i ..... number of appearance of unique function i in a core
q_M ..... current state of an FSM
r ..... number of lookup tables in a bitfile design
$\mathbf{r}$ ..... set of unique functions in a bitfile design
r_i ..... number of appearance of unique function i in a design

s ..... number of combinations of cores
S ..... signal
$S_0, S_{90}, S_{180}, S_{270}$ ..... phase signals
s_1, s_2, s_3 ..... different signatures
S_{AS} ..... accumulated signal
S_{BPSK} ..... BPSK modulated signal
S_{cc} ..... combined base band signal
S_D ..... signal after the differential step
S_{DS} ..... down sampled signal
S_{pr} ..... sampled probed voltage signal
S_M ..... finite set of states of an FSM
s_s ..... signature sequence
S_{sb1}, S_{sb2} ..... side bands of the carrier signal
S_ϕ ..... signal after the phase detection step
$s0_M$ ..... initial state of an FSM
SNR ..... signal to noise ratio
t ..... time
T ..... set of transitions
t_{dec} ..... time span for the first collision-free decoding
t_j ..... transition
t_I ..... threshold value
t_{ox} ..... oxide thickness
t_{rst} ..... reset time
$t_{sig,i}$ ..... time for sending the signature
$t_{wait,i}$ ..... waiting time
$\mathcal{T}$ ..... abstraction level transformation
V_{dd} ..... power supply voltage
V_t ..... threshold voltage
w_i ..... watermark element
W ..... watermark
W_A ..... author's watermark
W'_A ..... fake watermark
W_B ..... watermark at device level
W_L ..... watermark at logic level
W_{eff} ..... physical gate width

W_P ..... pirate's watermark
$\mathcal{W}$ ..... universe of watermark elements
X, Y, Z ..... example abstraction levels
x_{B_i} ..... bitfile element
x_i ..... work element
x_M ..... current input of an FSM
x_S ..... slice x coordinate
$x(t)$ ..... input signal
$x^*(t)$ ..... complex conjugate of $x(t)$
$\mathcal{X}$ ..... watermark extractor
y_S ..... slice y coordinate
Y_S ..... number of slice rows in an FPGA
$y(t)$ ..... output signal
$Z_{x,y}(t)$ ..... cross-correlation
$\mathcal{Z}$ ..... encoding alphabet
δ_i ..... period length
δ_M ..... state-transition function of an FSM
$\delta(t)$ ..... Dirac-pulse
γ ..... power consumption event through a shift operation
$\bar{\gamma}$ ..... no power consumption event
Γ_M ..... output alphabet of an FSM
$\Gamma(x)$ ..... gamma function
ω_i ..... number of repetitions
ω_M ..... output function of an FSM
ϕ_i ..... period length
σ_i ..... symbol which carries modulated data
Σ_M ..... input alphabet of an FSM
τ_0 ..... symbol length
τ' ..... transmitting slot length

Curriculum Vitae

Daniel Ziener took his university entrance qualification in 1998. He received his diploma degree (Dipl.-Ing. (FH)) in Electrical Engineering from University of Applied Science Aschaffenburg, Germany, in August 2002. Beside his studies, he gained industrial research experience during an internship at the IBM Germany Development Labs in Böblingen. From 2003 to 2009 he worked for the Fraunhofer Institute of Integrated Circuits (IIS) in Erlangen, Germany as a research staff in the electronic imaging department. Furthermore, in 2003 he joined the Chair of Hardware-Software-Co-Design at the University of Erlangen-Nuremberg, Germany, headed by Prof. Jürgen Teich as Ph.D. student. His main research interests are IP core watermarking, the efficient usage of the FPGA structures, design of signal processing FPGA cores, and reliable and fault tolerant embedded systems. Daniel Ziener has been a reviewer for several international conferences, for the IET Journal on Computers & Digital Techniques, the IEEE Transactions on Very Large Scale Integration Systems, and the Elsevier Journal for Microprocessors and Microsystems.