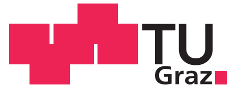

Manuel Jelinek

Protecting an Embedded Processor Design against Side-Channel Attacks by Domain-Oriented Masking

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Prof. Stefan Mangard

Institute for Applied Information
Processing and Communications (IAIK)

Co-Supervisor: Hannes Groß

Graz, September 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

Side-channel analysis (SCA) attacks pose a serious threat to embedded systems. The most common countermeasure against this attack is masking. So far, the research on masking, as a countermeasure against SCA, focuses merely on cryptographic algorithms for particular hardware or software implementations. However, the drawbacks of protecting specific implementations are the lack of flexibility in terms of used algorithms, the impossibility to update protected hardware implementations, and long development cycles for protecting new algorithms. Furthermore, cryptographic algorithms are usually just one part of an embedded system that operates on informational assets. Protecting only this part of a system is thus not sufficient for most security critical embedded applications.

This thesis introduces a flexible, SCA-protected processor design based on the open-source V-scale RISC-V processor. The introduced processor design can be synthesized to defeat SCA attacks of arbitrary attack order. Once synthesized, the processor protects the computation on security-sensitive data against side-channel leakage. The benefits of this approach are (1) flexibility and updatability, (2) faster development of SCA-protected systems, (3) transparency for software developers, (4) arbitrary SCA protection levels, (5) protection not only for cryptographic algorithms, but against leakage in general caused by processing sensitive data.

Keywords: protected CPU, domain-oriented masking, masking, side-channel protection, threshold implementations, RISC-V, V-scale.

Kurzfassung

Seitenkanalanalyse (SCA) Angriffe stellen eine ernsthafte Bedrohung für eingebettete Systeme dar. Die meistgenutzte Maßnahme gegen diese Art der Angriffe sind Maskierungsschemen. Bisher konzentriert sich die Forschung zum Schutz vor SCA mithilfe dieser Maskierungsschemen lediglich auf Implementierungen von kryptographische Algorithmen in Hardware oder Software. Allerdings besitzen diese Techniken den großen Nachteil dass diese speziell für Hardware- und Softwareimplementierungen entwickelt werden müssen und sich nur schwer auf neue Algorithmen anpassen lassen. Darüber hinaus sind kryptographische Algorithmen in der Regel nur ein Teil eines eingebetteten Systems zur Verarbeitung sensibler Information. Für sicherheitskritische eingebettete Anwendungen ist der ausschließliche Schutz dieser Teile eines Systems in den meisten Fällen nicht ausreichend.

Diese Arbeit stellt ein flexibles Prozessordesign vor, das auf den frei verfügbaren Quellen des V-scale RISC-V Prozessor basiert und gegen SCA Angriffe abgesichert wurde. Das eingeführte Prozessor-Design kann für beliebige Ordnungen von SCA Angriffen synthetisiert werden. Einmal synthetisiert schützt der Prozessor die Berechnungen der sensiblen Daten gegen das Durchsickern von Seitenkanalinformationen. Die Vorteile dieses Ansatzes sind (1) Flexibilität und Aktualisierbarkeit, (2) kürzere Entwicklungszeiten von SCA-geschützten Systemen, (3) Transparenz für Softwareentwickler, (4) beliebige SCA Schutzlevel, (5) Schutz nicht nur für kryptographische Algorithmen, sondern einen generellen Schutz vor dem Durchsickern von sensiblen Daten.

Stichwörter: geschützte CPU, domain-oriented masking, maskieren, Seitenkanalsicherheit, threshold implementations, RISC-V, V-scale.

Contents

Abstract iii

Kurzfassung iv

1. Introduction 1

2. Side-Channel Attacks and Countermeasures 4
2.1. Timing Attacks 5
2.2. Simple Power Analysis 7
2.3. Differential Power Analysis 7
2.4. Countermeasures to Power Analysis 8
2.4.1. Masking 8
2.4.2. Hiding 11

3. Hardware Masking Schemes 12
3.1. Classical Boolean Masking 13
3.2. Threshold Implementations 14
3.3. Domain-Oriented Masking 17

4. Target Processor Platform 22
4.1. RISC-V Instruction-Set Architecture 22
4.1.1. Instruction Length Encoding 23
4.1.2. Base RV32I Integer Instruction Set 24
4.1.3. Integer Multiplication Extension (RV32M) 29
4.2. V-scale Core 30
4.2.1. Pipeline Stages and Control Logic 31
4.2.2. Register-File 32
4.2.3. ALU 33
4.2.4. Multiplication and Division Unit 33

4.2.5. Data Memory 33

5. Protected Implementation of V-scale 35
5.1. Additional Pipeline Stage 37
5.2. Unprotected Operations 38
5.3. Protected ALU 39
5.3.1. DOM-AND 40
5.3.2. Protected Adder 41
5.3.3. Resharing of ALU Inputs and Outputs 46
5.3.4. Other ALU Operations 46

6. FPGA Design and Hardware Results 48
6.1. Device Under Attack – Kintex-7 50
6.1.1. Local Bus Interface 51
6.1.2. Memory Bridge 52
6.2. Control FPGA – Spartan-6 52
6.3. Hardware Results of V-scale 53

7. Evaluation 55
7.1. T-test Based Leakage Detection 55
7.2. Unprotected Device 56
7.3. First Order Protected Device 57

8. Conclusion 60

A. Abbreviations 62

Bibliography 64

List of Figures

2.1. Leakage observation without noise. 9
2.2. Effect of noise on the leakage observation. 10

3.1. Boolean masked AND ($GF(2)$ multiplier). 14
3.2. TI with component functions and three shares. 17
3.3. DOM-*indep* AND. 18
3.4. DOM-*dep* AND. 20

4.1. Instruction length encoding of RISC-V. 24
4.2. V-scale core overview. 30

5.1. Protected V-scale core overview. 36
5.2. Protected ALU. 39
5.3. Overview of the DOM-AND. 40
5.4. Kogge-Stone adder dependence tree. 42
5.5. Masked adder. 44

6.1. SAKURA-X block diagram. 48
6.2. SAKURA-X top view. 49
6.3. DUA block diagram. 50
6.4. Control FPGA block diagram. 53
6.5. Required LUT (left) and registers (right) on an FPGA. 54

7.1. T-test ($d = 1$) for unprotected device. 57
7.2. First order t-test value with inactive random generator. 58
7.3. First order t-test value with active random generator. 58
7.4. Second order t-test value with active random generator. . . . 59

1. Introduction

The resistance of security-critical systems against the broad field of passive physical attacks is a fundamental requirement of todays embedded devices and smart cards. If an attacker has direct or indirect physical access to an unprotected device, the observation of side-channel information (like power consumption [1] or electromagnetic emanation [2]) leaks information on the processed data. The security of such devices is then no longer guaranteed even if state-of-the-art cryptography is in place, because sensitive information like the used key material leaks through side-channel information.

The history of countermeasures against side-channel analysis (SCA) attacks is as old as the first paper targeting differential side-channel analysis by Kocher et al. [1]. Hereby, masking has become the first-choice measure to defeat SCA. The first masking approach was introduced by Goubin and Patarin [3], but many schemes followed like the Trichina gate [4] approach and the works of Ishai et al. [5], who introduced the concept of private circuits. However, many masking schemes have shown to be insecure in the presence of glitches that occur within the combinatorial logic of hardware implementations.

To overcome the inherent issue of glitches of these masking schemes, Nikova et al. [6] introduced the first-order secure threshold implementation (TI) masking scheme. The scheme was successively extended by Bilgin et al. [7–9] and Reparaz [10]. Moreover, there exist many scientific works that focus on the secure implementation of symmetric primitives following the TI scheme [11–13]. However, in comparison with software masking schemes, the original TI requires a higher number of random shares to handle glitches. A higher demand for fresh random shares goes hand in hand with increased hardware costs and higher randomness requirements, especially for implementations secure against higher-order attacks.

A primer for closing the gap between hardware and software masking schemes in terms of the required number of shares was the work of Reparaz et al. [14]. Their work demonstrates the feasibility of lowering the number of shares to $d+1$ (where d is the protection order) under certain circumstances. Most recently many works were published on the implementation of masked hardware implementations with reduced number of shares [15–18]. The work of Gross et al. [17] introduced the so-called domain-oriented masking scheme which follows on ideas of Ishai et al. [5] and Reparaz et al. [14] to build a masking scheme that requires only $d+1$ shares and allows for easy generalization to arbitrary protection orders.

Even though the trend to reduce the amount of shares to $d+1$ made protected hardware implementations more efficient and resulted in generic higher-order implementations, the efficient protection against SCA is still cumbersome, requires a lot of expertise for both implementation and evaluation, and is error-prone. Furthermore, the reduction of shares introduces additional register stages due to the decomposition of complex functions into a couple of algebraically simpler subfunctions [15]. This circumstance of additional delay cycles naturally brings implementations based on hardware masking schemes closer to software masking schemes in terms of throughput.

This thesis introduce a side-channel protected general-purpose CPU based on the RISC-V open instruction-set architecture [19] using the open-source V-scale [20] core. Therefore, the findings of domain-oriented masking [17] are used to modify the open-source V-scale CPU to be resistant against passive physical attacks.

The benefits of this approach compared to custom-made protected hardware implementations are, (1) more flexibility in terms of the selection of algorithms and updatability, (2) faster development of secure systems, (3) hardware-level protection that is transparent for both the running software and the designer, (4) the CPU can be synthesized for arbitrary protection orders by just changing one parameter, (5) a CPU is part of most security-critical systems and therefore requires SCA protection for security-sensitive data processed by the CPU anyway (which are not necessarily cryptographic operations).

The thesis starts with an overview of the most relevant side-channel attacks and their countermeasures in Chapter 2. The following Chapter 3 explains the concept of hardware masking schemes as a countermeasure to passive, non-invasive side-channel attacks. Chapter 4 introduces the structure of the open-source V-scale core and its provided operations. The changes to protect the core is described in Chapter 5 with detailed informations of the hardware implementations. The measurement setup with the used platform and the hardware results is shown in Chapter 6. The evaluation of the protected implementation is done in Chapter 7. Additionally, it introduces the used leakage detection methodology. The final Chapter 8 keeps the conclusion of this thesis.

2. Side-Channel Attacks and Countermeasures

Over the last years, the need for cryptographic devices has increased. Along with these devices, new kinds of attacks came up which increase the requirements for such security-critical devices. In a black-box model, the used cryptographic algorithms are secure because of their underlying mathematical structure. However, in many real world applications an attacker has full or partial physical access to these cryptographic device. These physical observability opens the gate to the broad class of so-called side-channel analysis attacks. Several kinds of side-channel analysis attacks have been developed over the last fifteen years. On the other hand, also new methods to protect and to evaluate the resistance to side-channel analysis attacks have been researched.

The attack scenarios considered for side-channel analysis attacks can be subdivided into to two major categories as shown by Mangard et al. [21]. The first category spits the attacks into active and passive:

Passive Attacks: The measurement setup of the attacked device is operated within its specification. Revealing secret information is based on observing physical properties like power consumption, electromagnetic emanation, execution time, ...

Active Attacks: The operation of the device is manipulated to cause an abnormal behavior to leak secure informations. This is done by manipulating the device inputs or environment.

The second category regards the exploited interface provided by the attacked device. Such interfaces can be directly accessible or could require special equipment. These attack types are one of the following and can be either active or passive:

Non-Invasive Attacks: The device is taken as it is and the attacks are done on the directly accessible interfaces, for example, by modifying the clock signal or supply voltage to force wrong operations of the logic.

Semi-Invasive Attacks: Similar to invasive attacks, it can be necessary to depackage the device. However, neither an electrical contact is made to the chip surface nor any irreversible device manipulations are done. Instead the device is manipulated, for example, by electromagnetic fields, X-rays or light to cause bit flips. This is a reversible effect and does not change the behavior of the device permanently.

Invasive Attacks: The device can be manipulated in any manner without any limits to reveal secret informations. Typically, it is necessary to depackage the device to obtain direct access of specific components. A probing station, for example, can be passively used to observe data signals like the data bus. Alternatively, the probing station can be used to alter the functionality of the device by actively changing signals. This requires special equipment like focused ion beams, probing stations or laser cutters. Such attacks can be irreversible and therefore manipulations are permanent.

The main goal of this thesis is to protect a micro controller design against SCA attacks. These attacks are defined as passive, non-invasive attacks and therefore, the device is taken as it is. This chapter gives an overview to the most commonly used passive side-channel attacks.

2.1. Timing Attacks

Timing attacks exploit runtime dependences of an underlying algorithm processing secure data. Timing differences may result from branching and conditional statements, or on hardware level by input dependent calculation times of an instruction (*e. g.*, multiplication). Therefore, the information of, for example, a single secret key bit can be exploited by using well taken inputs changing the runtime when the specific key bit is set. By doing this for all key bits, the secret key can be revealed. Timing attacks can be easily counteracted by adding dummy operations to get constant runtime.

Algorithm 1 gives an example of the square-and-multiply exponentiation algorithm which is used, for example, in asymmetric cryptography. The number of multiplications in Line 4 depends on the bits set in the exponent k and therefore the runtime of the algorithm changes significantly.

Algorithm 1: Square-and-multiply exponentiation.

```
  Input: Integers: x, k
  Result: y = x^k
1 y ← 1
2 foreach bit of k → k_i do
3   | y ← y^2
4   | if k_i = 1 then y ← y · x
5 end
6 return y
```

Adding a dummy operation as shown in Algorithm 2 at Line 7 always calculates the multiplication with storing the result on the dummy variable d. This simple countermeasure leads to a constant runtime independent from the exponent k. However, the necessary operations and therefore the runtime is maximized.

Algorithm 2: Secure square-and-multiply exponentiation.

```
   Input: Integers: x, k
   Result: y = x^k
1  y ← 1
2  foreach bit of k → k_i do
3    | y ← y^2
4    | if k_i = 1 then
5    |   | y ← y · x
6    | else
7    |   | d ← y · x
8    | end
9  end
10 return y
```

2.2. Simple Power Analysis

The goal of simple power analysis (SPA) is to derive a cryptographic key by direct interpretation of a small number of power traces or in an extreme case on a single trace. The number of power traces is therefore limited to the access of the device under attack (DUA) (*e. g.*, debit card). This restriction often requires detailed information of the attacked algorithm. With the possibility for a small number of runs, power traces with different inputs can be generated, or alternatively to reduce the noise an average of the traces with same input can be used. This method is quiet challenging due to the reduced number of power traces.

The examination of the power traces often requires visual analysis to find key-dependent patterns. This patterns in the power trace are based on runtime dependent algorithms using different operations of a system with significant power profiles. For example, different instructions of a micro controller accessing the external bus or peripherals requires more power or need more clock cycles whereas instructions using internal components have a lower power profile or operate in a single cycle. If such instruction sequences depends on the key an attacker can derive informations. Similar to timing attacks, a countermeasure against such attacks is to get a constant runtime which is independent from the input values.

2.3. Differential Power Analysis

In contrast to SPA attacks, differential power analysis (DPA) attacks reveal the key by measuring a large number of power traces with different inputs. Furthermore, a big advantage is the fact that no detailed knowledge is required of the implementation attacked cryptographic algorithm. In contrast to SPA attack which search patterns along the time axis of a single trace, the DPA analyze the power consumption on fixed moments of time based on a large number of traces. In a first step, the attacker has to choose an intermediate result of the attacked algorithm. This intermediate result needs to process an input d which can be chosen by the attacker combined with the unknown secret key k. For the attack the hypothetical intermediate values

of every possible k has to be computed. To reduce the number of different possible intermediate values the key can be partitioned into small parts. This hypothetical intermediate values are now mapped to a power model. One of the simplest and most often used is the Hamming weight (HW) model which simply counts the number of bits that are high. In the next step, the attacker has to record the power traces by processing the algorithm for different inputs d. This power traces are now correlated with the values of the hypothetical power model for all possible keys. The key hypothesis with the highest correlation result reveals the used key.

2.4. Countermeasures to Power Analysis

Power analysis attacks exploit that the power consumption depends on the processed data. Consequently, countermeasure against SCA try to reduce, or better eliminate this dependency. The two major approaches to achieve this are called masking and hiding.

2.4.1. Masking

Masking is one of the most popular methods to protect sensible data against SCA. For first-order masking, for example, an intermediate value x is split into two uniformly random shares A_x and B_x. The second share can be generated by either an arithmetic or logic operation $B_x = x * A_x$ such that $x = A_x * B_x$ is always valid. Therefore, the $*$ operator denotes an arithmetic operation or a boolean function. Arithmetic masking uses, for example, the multiplication or addition to mask the intermediate value. In contrast, boolean masking uses an exclusive-or (XOR) operation for the shared representation such that $x = A_x + B_x$, which equals an addition over $GF(2)$. Furthermore, each non-linear operation requires a fresh random mask to keep the intermediate values pairwise independent. Because the shares A_x and B_x are independent from the sensitive data x, also the power consumption is independent.

Assuming a HW leakage model to observe the leakage of a single bit in a noise-free setting and without masking leads to two different HW's (see Table 2.1 (left)) with the same probability as shown in Figure 2.1 (left). Both values of x can be directly distinguished by their power consumption and hence the intermediate value can be recovered. Adding a mask to the intermediate value leads to three different HW's as shown in Table 2.1 (right) with the highest probability for HW $= 1$. For this three HW's different leakages can be observed as shown in Figure 2.1 (right). In average the HW is thus 1 for both shared representations of each unshared value.

x	HW(x)
0	0
1	1

A_x	B_x	$\rightarrow x$	HW
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	2

Table 2.1.: Hamming weight for the unshared (left) and shared (right) value.

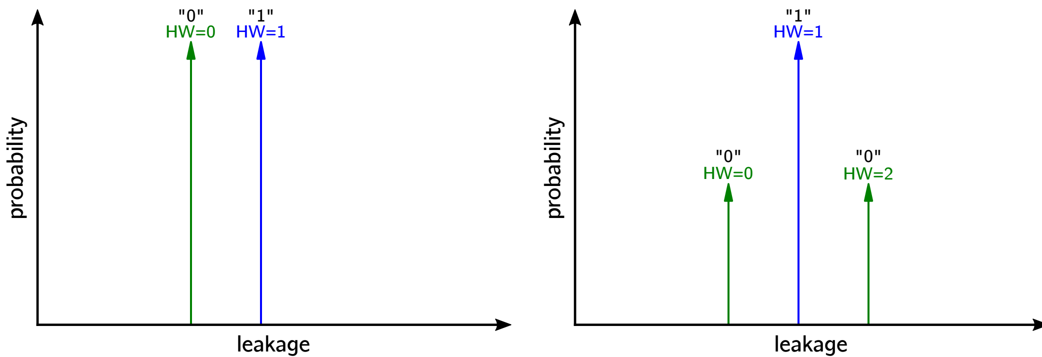

Figure 2.1.: Leakage observation without noise using the Hamming weight model for the unshared (left) and shared (right) value.

In a real world scenario the power consumption of a device consists not only of the single observed bit. Any device produces noise which complicates the differentiation of the exploitable power consumption. This noise is composed, for example, of static and dynamic power consumption of the

logic gates as well as for the measurement setup itself. Introducing this noise to the ideal leakage observation of the masked intermediate value in Figure 2.1 (right) leads to a normal distribution of the observed leakage for the different HW's as shown in Figure 2.2 (left).

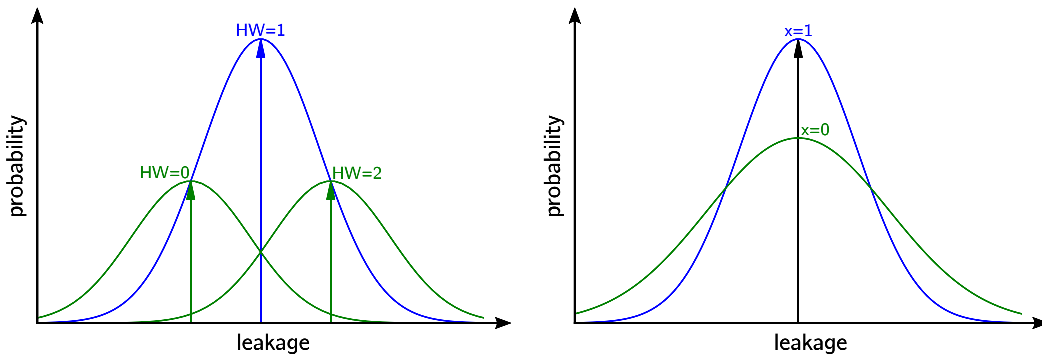

Figure 2.2.: Effect of noise on the observed leakage using a shared value.

The problem an attacker faces, lies in the interpretation of the measured power. An attacker which uses a first-order DPA attack with a HW model to exploit a single bit, searches in the power traces for differences in the mean of the power consumption as shown in Table 2.1 (left) leaking the information of the intermediate value. But the shared representation of the intermediate value consists of two independent bits which have in both cases the same probability to be "0" or "1". The power consumption is in average equal for $x = 0$ and $x = 1$. Only the variance differs as shown in Figure 2.2 (right), which is much harder to estimate and requires more traces.

This leads to the independence of the masked intermediate value from the unmasked one for a first-order DPA. But as shown in Table 2.1 (right), the power consumption is not independent if A_x and B_x are considered together.

The protection can also be increased by using multiple independent shares. This increases the number of HW's which represents the different power leakages an attacker has to consider. Therefore, the number of used shares for a single intermediate value is given by $d + 1$, where d is the so-called protection order.

2.4.2. Hiding

Hiding, on the other hand, follows a different approach to reduce the dependencies of the power consumption. This can be achieved either by building devices with equal or random power consumption on each clock cycle. A device with equal power consumption can execute the data almost the same way as an unprotected device. Therefore, the signal-to-noise ratio of an operation has to be lowered either by increasing the noise, using random switching activity in an device, or by reducing the signal, for example, by dual rail logic which additionally carries the complementary signal.

The second approach for hiding is by randomizing the power consumption. An algorithm can therefore execute random inserted dummy operations or randomly shuffle the execution order, for example, table look-ups. The hiding approach reduces the dependencies of the power consumption which increases the number of required traces to successfully achieve an attack. Therefore, the goal is to increase the required number of traces which is impracticable for a real device.

3. Hardware Masking Schemes

Side-channel attacks such as differential power analysis or chip probing attacks exploit data dependencies within the observed side-channel information. Therefore, the intuition behind masking is to make security-critical computations independent of the underlying data. Many masking schemes achieve this data independence by representing variables in a so-called shared representation which ensures independence up to a certain protection order d. One of the most popular formal models to investigate the security of masking schemes is the so-called d-probing model introduced by Ishai et al. [5]. In this probing model, the protection order d equals the number of needles an attacker can utilize in parallel. A circuit that resists probing attacks with up to d needles is said to be d-secure. It was shown by Faust et al. [22] and Rivain and Prouff [23] that the probing model is also applicable to side-channel attacks (like differential power analysis) to describe the resistance of an implementation in relation to the maximum statistical moment a certain attack is targeting. As it was shown by Chari et al. [24], there is an exponential relation between the protection order d and number of observations (leakage traces) an attacker has to perform for successfully exploiting the side-channel information.

The implementation costs for masking schemes, like chip area and randomness requirements, are strongly related to the number of used shares. Hereby, a variable x is represented as the sum of $d+1$ shares in $GF(2)$. Each of these shares are denoted by capital letters (see Equation 3.1) with the associated variable in the index. To denote the sharing itself, a bold capital letter is used as abbreviation for writing each share of x explicitly.

$$x = \underbrace{A_x + B_x + C_x + \dots}_{d+1} = \mathbf{X} \tag{3.1}$$

Furthermore, the protection also affects the applied functions. Therefore, any function that is intended to be performed on the unshared variable x is instead applied on the shares of x. As a result, any linear function $\mathrm{F}(x, y, \dots)$ is split up in $d+1$ independent component functions (see Equation 3.2). Each component function is applied on each share separately.

$$\mathrm{F}(x, y, \dots) = \underbrace{\mathrm{F}_A(A_x, A_y, \dots) + \mathrm{F}_B(B_x, B_y, \dots) + \mathrm{F}_C(C_x, C_y, \dots) + \dots}_{d+1} \quad (3.2)$$

Sharing any non-linear function on the other hand is quiet challenging. Computing non-linear functions requires to evaluate cross-domain terms in a secure manner. The best strategy for resolving these cross-domain dependencies is not trivial. Hence, every masking scheme applies its own strategy. An important requirement for all masking schemes is the statistical independence of all intermediate values compared to their unshared input and output values up to a certain degree. As it is not always possible to maintain this independence directly for every shared function, it is necessary to add fresh randomness to the intermediate results. This random values have to be picked uniformly random and are therefore statistically independent. In this work the random values are represented by the uppercase letter Z. The following sections explain different masking schemes in detail.

3.1. Classical Boolean Masking

The idea of classical boolean masking (CBM) is to hide the data dependency of any sensitive information by sharing them into multiple random values as shown in Equation 3.1. A first-order AND function with two inputs, for example, requires two shares per variable $x = A_x + B_x$ and $y = A_y + B_y$.

$$q = xy = (A_x + B_x)(A_y + B_y) = A_xA_y + A_xB_y + B_xA_y + B_xB_y \quad (3.3)$$

The partial products of the multiplication as shown in Equation 3.3 are independent of the inputs x and y, whereas the resulting sum represents

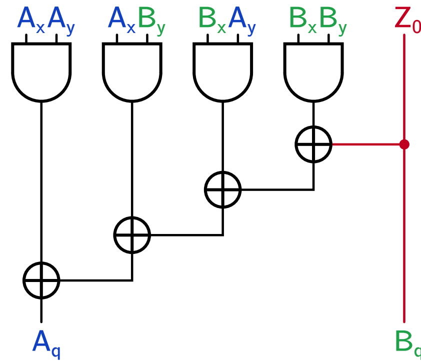

Figure 3.1.: Boolean masked AND ($GF(2)$ multiplier).

the unshared value. This dependence can be broken by adding a fresh random share Z_0 to the first multiplication result as shown in Figure 3.1. This guarantees the data security as long as the random share is the first signal propagated through the XOR gates. On hardware level it is hard to control the delay for signal transitions of the logic. This delays are influenced by transistor speed, wire lengths and other side effects. Therefore, the calculation of the first two multiplications $A_x A_y$ and $A_x B_y$ can reach the XOR gate before adding the random value Z_0 which causes data dependences to the value of y.

As shown by Mangard and Schramm [25] the leakages of such an boolean masked multiplier is not caused by the AND gates. Instead the leakage is caused by glitches, which toggle the output of the XOR gates. Removing this glitches requires reordering the logical blocks and signals on gate level. Because this is a tremendous effort, masking schemes avoid these glitches by design.

3.2. Threshold Implementations

The main goal of TIs is to get rid of the glitch vulnerability of the CBM by guaranteeing the data independence for all possible signal timings. Therefore, a TI has to fulfill three properties, correctness, non-completeness,

and uniformity. The properties are explained in the following for a first-order secure AND gate.

Correctness: A natural requirement for all masking schemes is correctness which requires that the resulting sum over all outputs of the component functions must be equivalent to the result of the unshared function:

$$q = A_q + B_q + C_q = \mathrm{F}_A + \mathrm{F}_B + \mathrm{F}_C = \mathrm{F}(x, y) \tag{3.4}$$

Non-completeness: The component functions of the TI have to be independent of at least one input share of each input variable. In order to achieve non-completeness, a first order secure TI requires at least three shares. Equation 3.5 gives an example of the component functions for a first-order AND gate. The first component function F_A is independent of all A shares and therefore fulfills the non-completeness property. This applies also to the other two functions.

$$\begin{aligned} A_q &= \mathrm{F}_A(B_x, C_x, B_y, C_y) = B_x B_y + B_x C_y + C_x B_y \\ B_q &= \mathrm{F}_B(A_x, C_x, A_y, C_y) = C_x C_y + A_x C_y + C_x A_y \\ C_q &= \mathrm{F}_C(A_x, B_x, A_y, B_y) = A_x A_y + A_x B_y + B_x A_y \end{aligned} \tag{3.5}$$

Uniformity: The uniformity requirement states that every input or output share of a component function has to be uniformly distributed and as a consequence cannot be distinguished from random. Therefore, it can be necessary to reshare the outputs by adding additional randomness. An example for a first-order AND gate is given in Equation 3.6 which is one possible solution as shown by Bilgin et al. [26]. This solution requires a single fresh random value (Z) to guarantee the uniformity of the outputs. It is also possible to use additional shares to avoid the usage of more fresh randomness. However, this naturally increases the required amount of logic gates and registers.

$$\begin{aligned} A_q &= \mathrm{F}_A(B_x, C_x, B_y, C_y, Z_0) = B_x B_y + B_x C_y + C_x B_y + Z_0 \\ B_q &= \mathrm{F}_B(A_x, C_x, A_y, C_y, Z_0) = C_x C_y + A_x C_y + C_x A_y + A_x Z_0 + A_y Z_0 \\ C_q &= \mathrm{F}_C(A_x, B_x, A_y, B_y, Z_0) = A_x A_y + A_x B_y + B_x A_y + A_x Z_0 + A_y Z_0 + Z_0 \end{aligned} \tag{3.6}$$

	(B_y, C_y)			
(B_x, C_x)	00	01	10	11
00	0	0	0	0
01	0	0	1	1
10	0	1	1	0
11	0	1	0	1

Table 3.1.: All input bit combinations for the first component function ($F_a = B_xB_y + B_xC_y + C_xB_y$) without fresh randomness.

Table 3.1 shows the resulting bits for the first component function F_A without the additional randomness. Therefore, all possible input combinations are evaluated to illustrate the non-uniformity. Another possibility to prevent resharing, is to increase the number of used shares to find component functions generating uniform outputs.

Before applying the outputs of the component functions to the next one a decoupling register is required to guarantee the *non-completeness* property. Directly connecting TI functions can leak again data through glitches.

One drawback at TI are the additional shares compared to CBM which leads to more hardware overhead required for the shared function, using the same protection order d. Moreover, the minimal required input shares s_{in} depends on the product of the protection order and the degree of the function (t) given by $s_{in} \geq d \times t + 1$. The fact that the degree of a non-linear function is always larger than one ($t > 1$) increases the required input shares compared to CBM. Furthermore, non-linear functions additionally increase the required output shares s_{out} depending on the function degree given by $s_{out} \geq \binom{s_{in}}{t}$ and hence the number of shares increases drastically.

The following example shows a first-order secure ($d = 1$) AND with the minimum amount of shares—three input and output shares. Figure 3.2 shows the three component functions with the corresponding input shares from Equation 3.6 fulfilling the *non-completeness* property. Compared to CBM, the first-order TI requires an additional share per variable.

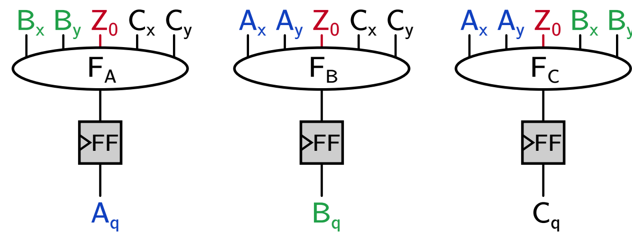

Figure 3.2.: TI with component functions and three shares.

A big advantage of the TI compared to the CBM scheme is the resistance against glitches which is guaranteed by the *non-completeness* property. On the other hand, this glitch resistance comes with an enormous logic overhead compared to the CBM. The first-order boolean masked AND for example requires four XOR as well as four AND gates, whereas the TI requires 12 XOR and 13 AND gates which is about three times as much. Furthermore, the TI requires an additional register for the third share.

3.3. Domain-Oriented Masking

While the focus of the TI is the functional level, the domain-oriented masking (DOM) scheme targets share domains. The number of domains as well as the number of shares is given by $d + 1$. Furthermore, each share is associated with one specific domain. For example, the first shares of the variables x (A_x) and y (A_y) are associated with domain A, the second shares B_x and B_y with domain B, and so on.

The idea of the DOM scheme is a strict separation of shares from other domains. Linear functions use only inner-domain terms which are always independent from their unshared input values and hence they are unproblematic like it is shown by the first-order protected AND in Figure 3.3. Therefore, all terms of a shared function can be separated into inner-domain terms using only shares of the same domain (*e.g.*, $A_xA_y, B_xB_y, \ldots$) and cross-domain terms mixing shares of different domains (*e.g.*, $A_xB_y, B_xA_y, \ldots$) which requires a special treatment.

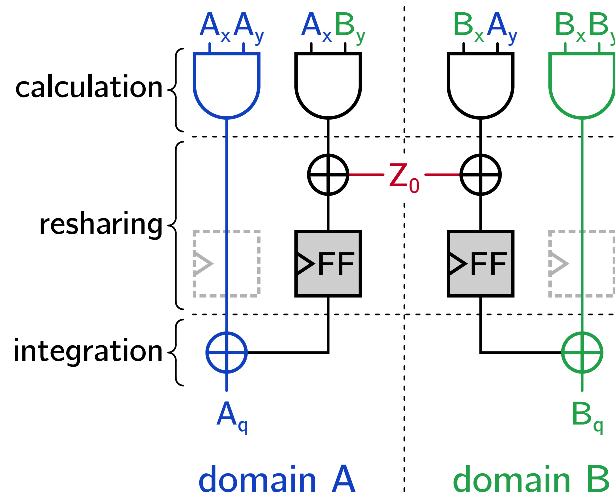

Figure 3.3.: DOM-*indep* AND.

The first step is the *calculation* of all multiplication terms. Adding the cross-domain terms to the inner-domain terms in the *integration* step violates the independence by using shared inputs from different domains. To restore the independence of this cross-domain terms a fresh random value (*e. g.*, $Z_0, Z_1, \dots$) is added in the *resharing* step. A further requirement is an additional register after the resharing to prevent glitches propagating data from one domain to the other. Considering the independence of the inner-domain terms, this register optional and only used for pipelining reasons. In the last *integration* step the cross-domain terms can safely be added to their inner-domain terms.

An additional requirement for a non-linear DOM-*indep* implementation is the independence of the input shares. Consider, for example, a protected AND as shown in Figure 3.3 uses value x as the first and second operand and identical sharing ($A_y = A_x$ and $B_y = B_x$). The shares of the inner-domain terms $A_x A_x$ and $B_x B_x$ are independent and therefore uncritical, whereas the cross-domain terms $A_x B_x$ and $B_x A_x$ brings shares of different domains from the same variable together. This violation does not occur by using the same input variable x twice. The problem refers to the identical input shares and hence can easily be solved by sharing both inputs independently,

for example, by adding fresh randomness before applying it as one of the inputs. To suppress the propagation of identical shares to the input caused by glitches requires a decoupling register after adding the fresh randomness which increase the delay.

DOM-dep variant: In order to prevent additional delays, in case of dependent input sharings of x and y, Gross et al. [17] shows the DOM-*dep* variant, which replaces the direct calculation of xy using a blinding value z as shown in Equation 3.7.

$$xy = x\underbrace{(y+z)}_{blinding} + \underbrace{(xz)}_{correction} \tag{3.7}$$

The shares of the blinding value z are therefore pairwise added to the shares of the second input y (see Equation 3.8).

$$y + z = (A_y + A_z) + (B_y + B_z) + (C_y + C_z) + \ldots \tag{3.8}$$

This blinding guarantees that every share is independent from the unshared value of y which is required for the next calculation step which demasks the blinded value. Before applying the demasking, an additional register stage is required to prevent glitches before the blinding of y is finished. The single demasked value of $(y+z)$, represented by value b, is then multiplied to every share of x which does not compromise the shares as they remain to their specific domains (see Equation 3.9).

$$x(y+z) = xb = bA_x + bB_x + bC_x + \ldots \tag{3.9}$$

Furthermore, their are no additional DOM multipliers required for this operation. For pipelining reasons, an optional register stage can be applied to the shares of x before performing the multiplication.

In a last step, the multiplication result must be corrected by the second term of Equation 3.7 to get the correct result. Therefore, the input x and the blinding value z are applied to a DOM-*indep* multiplier to ensure that

any two shares of the same input come together. This result represents the correction term which is then added to the multiplication result of Equation 3.9 and represents the result of the DOM-*dep* variant.

Figure 3.4 shows an example of a DOM-*dep* AND with two shares. Additionally to the required random values of the internal used DOM-*indep* multiplier, further random values are required for each share of the blinding value z.

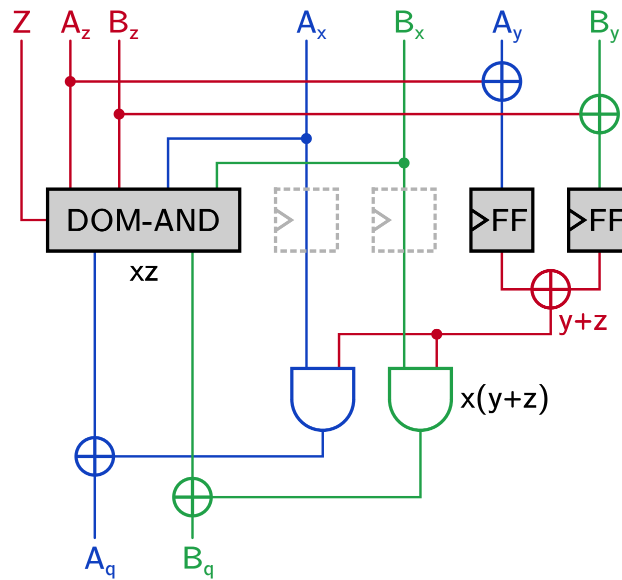

Figure 3.4.: DOM-*dep* AND.

Summary: The first-order protected AND (see Figure 3.3) requires four AND as well as four XOR gates for the calculation which is equal compared to the CBM and less than the TI (see Table 3.2). On the other hand, the two registers are mandatory compared to the CBM whereas the TI requires one additional register. Another major problem with the CBM are the glitches which is not practical for a simple protected design. The reduced hardware cost compared to TI and the unproblematic behavior of DOM makes this to the more suitable masking scheme for protecting the targeted micro controller.

	AND	XOR	Reg.	Randomness
CBM	4	4	–	1
TI	13	12	3	1
DOM-*indep*	4	4	2	1
DOM-*dep*	6	9	4	3

Table 3.2.: Required logic cells and fresh random values for first-order masking schemes.

Another big advantage is the genericity of DOM, allowing to design protected hardware with arbitrary protection order. This simplifies the usage for designers as they can simply change the protection order by changing a single value.

4. Target Processor Platform

This work builds upon the open-source V-scale processor that implements the RISC-V instruction-set architecture (ISA). The RISC-V ISA was originally developed at the University of California, Berkely. RISC-V is a customizable, modular, free and open RISC ISA which perfectly suits academic purposes. The architecture is highly flexible, meaning that the register size (32, 64, or 128 bit), their number (16 or 32), the number of privilege levels (1 to 4), and the supported instructions can be chosen according to the desired use case.

Like the ISA, also the V-scale processor core has been developed in Berkely. V-scale is a Verilog implementation of the RV32IM instruction set, *i. e.*, it is a RISC-V processor with 32 registers with 32 bit width featuring the mandatory base integer instruction set and the optional extension for integer multiplication and division. This chapter outlines the V-scale core and the related RISC-V ISA features.

4.1. RISC-V Instruction-Set Architecture

The ISA defines the mandatory base integer instruction set (I or E) which contains the most basic memory, arithmetic, logic, and control-flow instructions. The architecture is highly flexible, meaning that the register size, their number, the number of privilege levels and the supported instructions can be chosen according to the desired use case. The base integer instruction set RV32I and RV64I uses 32 or 64 bit width for address as well as for the 32 integer registers respectively. Alternatively, the RV32E instruction set variant can be chosen which was designed for embedded system to reduce the size and energy effort of the core. Therefore, the integer registers are

reduced to 16 and the mandatory counters of RV32I are removed. Moreover, a RV128I variant will be provided in the future.

Optionally, more complex instructions can be implemented and are defined via various extensions of the base integer instruction sets. These extensions include, for example, instructions for integer multiplication/division (M), atomic (A) operations, as well as single- (F) and double-precision (D) floating-point computations. The instructions in the base instruction set and the mentioned extensions are all encoded in 32 bits. However, both shorter and longer instructions are supported too. The extension for compressed instructions (C), for example, defines 16-bit instructions which map to the base instruction set to increase code density. Furthermore, RISC-V also supports the addition of fully-custom instructions as so called non-standard extensions (X).

The RISC-V, unlike the AVR, x86, and the ARM ISA has no status flags (carry, overflow, zero, ...). Carry propagation as well as comparisons are instead performed with dedicated instructions. The lack of status flags makes the RISC-V ISA a more suitable basement for side-channel protected processors than other ISA's because it reduces the complexity of the masked functionality.

4.1.1. Instruction Length Encoding

The RISC-V ISA uses a fixed 32-bit instruction length which is mandatory for all implementations independent from the used register size (32, 64 or 128 bit). Therefore, all instructions are naturally aligned on 32-bit boundaries. However, when the compressed (C) instruction set extension with 16-bit instruction length is used, the data alignment changes to 16-bit boundaries for both 16-bit and 32-bit instructions. Furthermore, the RISC-V ISA uses an elaborated instruction length encoding to be prepared for the future. The instruction length encoding is selected by the opcode part of the instruction (`instr[6:0]`) by increasing the numbers of successive one bits (see Figure 4.1). For instructions larger than 16 bits, the two least significant bits of the instruction are always set to one (`instr[1:0]=11`). Any other con-

figuration of the two least significant bits are reserved for the compressed 16-bit instructions.

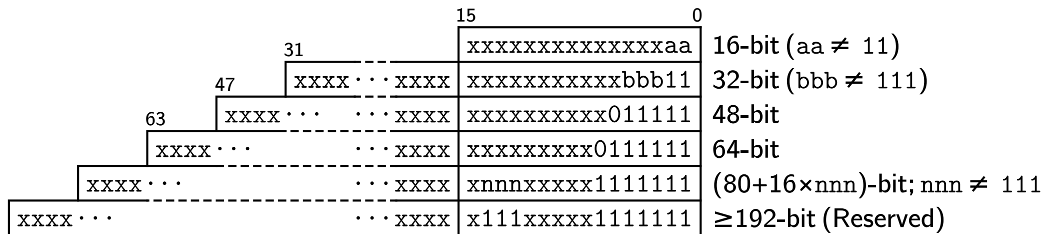

Figure 4.1.: Instruction length encoding of RISC-V.

4.1.2. Base RV32I Integer Instruction Set

The base integer instruction set operates on the 32 user-visible general-purpose registers `x0–x31` with a data width of 32 bits for RV32 and 64 bits for RV64. These registers are accessible by all integer instructions and none of these instructions are bounded to a special instruction. However, some standardized special purpose registers are required (*e. g.*, stack pointer, return address, function arguments, . . .) which are defined in the application binary interface (ABI). Register `x0` is a special case where written data is discarded and a reading from this register always returns zero.

The last user-visible register is the program counter (**PC**) which holds the address of the executed instruction. This register is not directly accessible like the general-purpose registers, but it can be modified by special instructions like jump and branch operations.

Table 4.1 gives an overview of the opcode groupings for the RISC-V standard extensions (IMAFD). The two least significant bits are not shown as they are always one for 32-bit instructions (`inst[1:0]=11`). Furthermore, if the three following bits in the opcode are set (`inst[4:2]=111`) indicates an instruction greater than 32-bit. The mandatory base integer instruction types are highlighted in blue and hence they are present in all RISC-V implementations.

		inst[4:2]							
		000	001	010	011	100	101	110	111 (>32b)
inst[6:5]	00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
	01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
	10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
	11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥80*b*

Table 4.1.: RISC-V opcode map of 32-bit instruction. Two least significant bits of instruction always one (`inst[1:0]=11`). Base integer instructions are highlighted in blue.

A detailed overview of all base integer instructions is shown in Table 4.3. These instructions uses only the four different decoding formats, namely R, I, S and U (see Table 4.2) to reduce the decoding complexity. The S and U instruction formats have an additional subtype SB and UJ which differs only in the decoding of the immediate values. To further decrease the decoding complexity, the source registers (*rs1* and *rs2*) as well as the destination register (*rd*) stays on a fixed position in the instruction. Therefore, no additional logic is required to get the register addresses which saves area and increases speed. A further simplification is the fact that all immediate values are sign extended and the sign bit of all immediate value types use the most significant bit of the instruction (see blue highlighted position in Table 4.2). This reduces the hardware complexity for the sign extension.

31	30 ... 25	24 ... 21	20	19 ... 15	14 ... 12	11 ... 8	7	6 ... 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	SB-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	UJ-type

Table 4.2.: Instruction formats of RV32I. Most significant bit of IMM (highlighted in blue) always on the same position in the instruction.

31–25	24–20	19–15	14–12	11–7	6–0	Instruction
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred | succ		00000	000	00000	0001111	FENCE
0000 | 0000 | 0000		00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
001100000010		00000	000	00000	1110011	MRET
000100000101		00000	000	00000	1110011	WFI
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Table 4.3.: RV32I instructions.

In the following a short overview of the instructions of RV32I given since these are the instructions that are targeted in this work. More detailed information can be found in the RISC-V instruction set manual [27].

LUI - Load Upper Immediate (U-type): Zero append the lower 12 bits of the U-type immediate value and write it to the destination register. An **LUI** followed by an add immediate (**ADDI**) operation can generate every possible 32-bit value.

AUIPC - Add Upper Immediate to PC (U-type): Same immediate value type as **LUI** with zero appending and adding it to the current **PC** to build **PC**-relative addresses.

JAL - Jump and Link (UJ-type): The unconditional jump instruction stores the incremented **PC** into the destination register and adds the sign extended UJ-type immediate value to the current **PC** which gives a range of ± 1 MiB.

JALR - Jump and Link Register (I-type): Stores the incremented **PC** in the destination register. Afterwards the **PC** is set to the sum of the source register value with the sign extended I-type immediate value.

Branch Instructions (SB-type): The RISC-V ISA has no status flags to perform the conditional jumps. Instead, dedicated instructions are used which compare the two source registers *rs1* and *rs2*. If the branching condition is fulfilled, a relative jump is performed by adding the sign extended SB-type immediate value to the current **PC**. This leads to a maximum jump range of ± 4 KiB. The possible compare operations for branching are **BEQ** and **BNE** to branch if the source registers are equal or not as well as **BLT[U]** and **BGE[U]** to branch if the signed or unsigned values in source register *rs1* is less (**LT**) or greater equal (**GE**) than *rs2* respectively.

Load and Store Instructions (I/S-type): The load (**Lx[U]**) and store (**Sx**) instructions use the first source register (*rs1*) as base address with the sign extended immediate value as offset. Therefore, the lower 5-bit of the 12-bit immediate value is taken by the unused register part of the instruction. This is the *rs2*-part for load instructions (I-type IMM) and the *rd*-part for the store instructions (S-type IMM). This 12-bit offset allows to address memory inside a range of ± 2 KiB. Furthermore, the load and store instructions support 8-bit (**B**), 16-bit (**H**) and 32-bit (**W**)

values which are selected by the *funct3* part of the instruction. For the load instructions the shorter values (**B**/**H**) are sign extended before writing them into the destination register otherwise, the corresponding unsigned load instruction (**LxU**) should be used.

Register-Immediate Instructions (I-type): This instructions uses the first source register *rs1* as first input operand and the sign extended 12-bit I-type immediate value as second input operand. An exception are the shift operations which uses only the lower 5-bit of the immediate value. The result is stored in the destination register *rd*. Supported operations are AND, OR, XOR, ADD, a signed and unsigned comparison, and logical and arithmetic shifts which are selected by the `funct3` part of the instruction. In contrast to the branch instructions, only a signed and unsigned compare operation for "less than" (**SLTI[U]**) is supported.

Register-Register Instructions (R-type): In contrast to the register-immediate operations the immediate input is replaced by a second register input using the R-type format. Moreover, the same instructions as for the register-immediate instructions are supported with an additional subtraction operation. As the immediate values are always sign extended the subtraction is performed implicitly when an addition is performed. Using an register as second input operand, however, requires an explicit subtraction instruction.

Memory Fence Instructions: The fence instructions ensure that previously initiated memory accesses are completed before continuing. Therefore, the most fence instructions in a single-issue in-order pipeline architecture like the V-scale processor are no operations which flush the execution pipeline. This can be used, for example, in self modifying code to guarantee that modified instructions are correctly loaded.

System Instructions (I-type): The system instructions uses an I-type format and supports operations for system calls, which are named environmental calls (**ECALL**, **EBREAK**) in the RISC-V ISA, to enter a higher privilege level. Therefore, the current (**PC**) is stored in a control register and can be restored by an return instruction (**MRET**) to continue the execution.

The RISC-V ISA owns control and status registers (CSRs) to handle informations of the processing unit (*e. g.*, used RISC-V extensions, counter, ...) as well as configuration registers (*e. g.*, interrupt enable,

interrupt-handler base address, . . .). The atomic **CSRRx[I]** instructions are used to read the old value of a CSR, store it in the destination register *rd* and modify or replace it with a new value. Therefore, a 5-bit immediate value (*zimm*) which is zero extended or the source register *rs1* can be used as input to write (**CSRRW[I]**) into the control register or use it as a bit mask to set (**CSRRS[I]**) or clear (**CSRRC[I]**) the corresponding bit positions. The wait for interrupt instruction (**WFI**) is used to stall the environment until an interrupt appears.

4.1.3. Integer Multiplication Extension (RV32M)

The integer multiplication extension provides additional instructions to multiply or divide integer values of two source registers (see Table 4.4). For this extensions the same opcode is used as for the other register-register instructions (*e. g.*, ADD, OR, AND, . . .). Again the R-type instruction format is used (see Table 4.2). The basic operations for multiplication is **MUL** using the two 32-bit register inputs and generate the lower 32-bit result. There exist dedicated operations to handle the upper 32 bits of the result. These are separated in variants for *signed* $\times$ *signed* (**MULH**), *unsigned* $\times$ *unsigned* (**MULHU**) and *signed* $\times$ *unsigned* (**MULHSU**) input operands.

The division operations support signed (**DIV**) and unsigned (**DIVU**) inputs and the (**REM**) and (**REMU**) instructions provide the appropriate remainder.

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	MUL
			001			MULH
			010			MULHSU
			011			MULHU
			100			DIV
			101			DIVU
			110			REM
			111			REMU

Table 4.4.: RV32M instructions.

4.2. V-scale Core

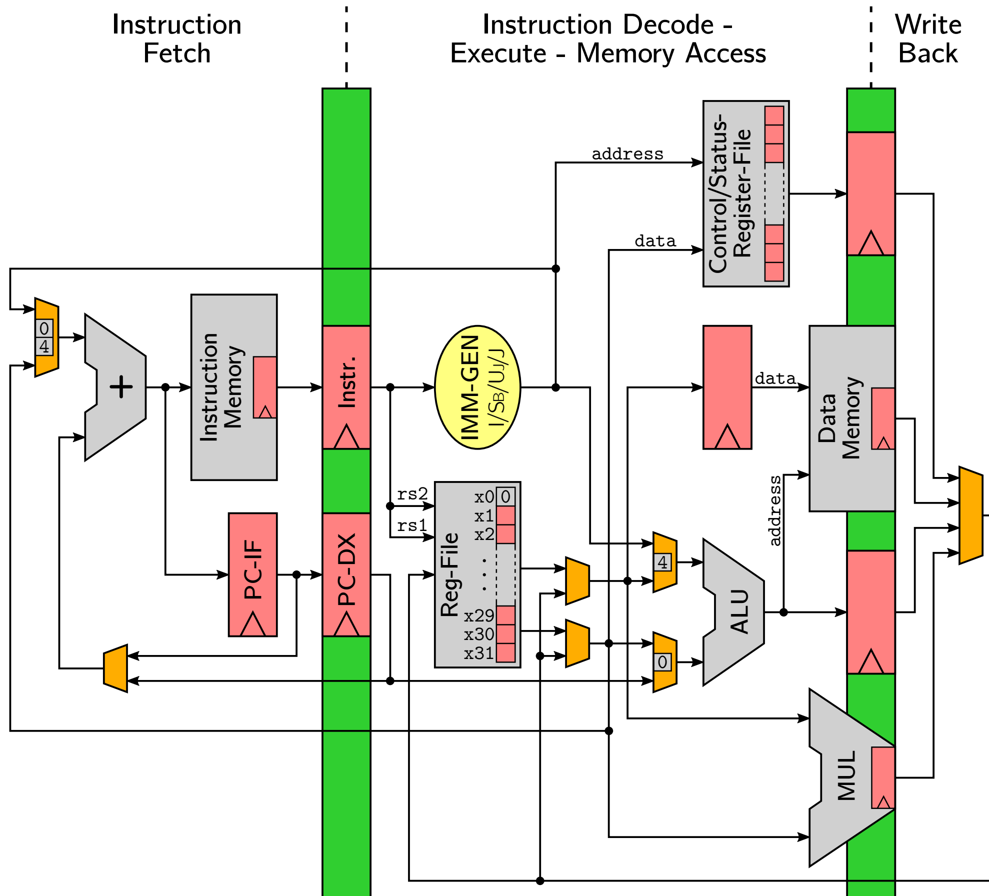

Figure 4.2.: Overview of the V-scale core pipeline.

The V-scale processor core is described in the hardware description language Verilog and implements a subset of the RISC-V ISA. The core consists of 32 general-purpose registers and implements the 32-bit base integer instruction set and the additional hardware multiplication and division extension (RV32IM). Figure 4.2 shows the basic processor design which is based on a single-issue in-order 3-stage pipeline architecture comprising a fetch stage, a combined decode and execute stage, and a write back stage. Additionally, data dependencies between consecutive instructions are

handled more efficiently by using a dedicated bypass multiplexer in the write back stage which allows to increase the utilization of the arithmetic logic unit. Communication with the external memory relies on separated AHB-Lite memory interfaces for instruction and data handling. This separation allows the flexibility to build both Harvard as well as von Neumann architectures.

4.2.1. Pipeline Stages and Control Logic

The most sophisticated part of the core is formed by the control logic. It has to apply the correct control signals to the data path logic for the different pipeline stages and has to deal with hazards in the instruction pipeline. The pipeline uses the following three stages:

FETCH: The first task of the pipeline is to fetch an instruction from the memory by incrementing the **PC** or alternatively alter the pipeline in the case unconditional jump and branch instructions are executed or hardware exceptions occur (*e. g.*, interrupts, bad memory access, . . .). This stage can cause an exception when the **PC** uses an unaligned or invalid memory address. This happens when the two lowest bits of the destination address are not zero. For the invalid memory access the processor owns the bad memory access input. Both exceptions set the **PC** to the trap-vector base-address placed in the CSR which holds the code to handle such exceptions.

DECODE+EXECUTE: The decoding of the instruction is mainly performed by the control unit. Depending on the opcode, the source register or immediate value is chosen as well as the decoding of the immediate value is performed and the control signals for the multiplexers in the data path are generated. The execution of the operation is performed in the same clock cycle as the decoding. Therefore, the corresponding inputs are propagated through the data path and processed in the arithmetic logic unit (ALU). As a result, it is possible to execute each base integer instruction within one cycle. However, this does not apply for the multiplication and division instructions which stalls the pipeline execution until a signal indicates that

the result is valid.

Algorithm 3: Example for data hazard: read after write (RAW)

```
i1 x3 ← x1 + x2
i2 x1 ← x3 + x2
```

In this stage the control logic must deal with some hazards caused by data dependencies of consecutive instructions or code that should not be executed because a branch occurs. Algorithm 3 shows an example of the read after write (RAW) data hazard where the destination register of the first instruction (**i1**) is immediately used as source register for the following instruction (**i2**) which still contains the old register value due to the write back delay. To prevent a stalling of the pipeline, a bypass multiplexer is added to use the result directly as an input for the next execution stage.

Another control hazard appears on conditional jumps when the branch is taken. In this case the control logic needs to flush the instruction pipeline and insert no operations until the next instruction is correctly fetched from the memory.

WRITE BACK: In the final write back stage the source is chosen at first which is then stored in the destination register. Possible sources are the data memory, the control and status register, the ALU, and the hardware multiplication and division unit.

4.2.2. Register-File

The register file contains the 32 general-purpose registers which are accessible over two independent ports for reading and an additional port for writing. Switching between the registers works immediately whereas the write operation is always performed on a positive clock edge. The first register `x0`, also called the zero register, is hardwired to the constant zero and the data written to the `x0` register is always discarded.

4.2.3. ALU

The ALU has two data input ports and provides logic operations for AND, OR, XOR, comparison, addition, subtraction, arithmetic and logic shifts. It is formed by purely combinatorial logic and therefore no internal register is used. This is necessary to prevent the address calculation of the load and store instructions from an additional delay. As shown in Figure 4.2 a register is added at the output of the ALU as write back source.

4.2.4. Multiplication and Division Unit

The hardware multiplier and divider is a separate module which is directly connected to the register sources, since the appropriate instructions use no immediate values. In difference to the ALU, the hardware multiplier and divider module takes several cycles to generate a valid output. Hence, the instruction pipeline needs to be stalled until the response valid signal indicates that the operation is finished. An internal result register is used as source for the write back stage.

4.2.5. Data Memory

The data memory supports read and write operations with 8, 16 and 32-bit data width. The address generation is done by adding a source register and the immediate value using the ALU and applying it to the memory. Since the memory uses a register at the output to minimize its critical path, the reading from a memory address is delayed by one clock cycle. The output of the register is directly used as a source for the write back. The input of the data memory is directly taken from the second source register whereas the memory address is taken from the ALU output. The value from the second source register is stored in an additional register before it is applied to the memory in order to shorten the signal propagation path. Unfortunately, the write operation is therefore always delayed by one clock cycle. This leads to no additional delays for sequential write operations, however, using a read

operation after writing to the memory leads to a pipeline stall since reading is done immediately and writing is delayed.

5. Protected Implementation of V-scale

The protection of the implemented V-scale core addresses the problem that processed data is subject to side-channel attacks. This works focus affects solely the protection of the base RV32I instruction set as it is the most versatile. Nevertheless, the multiplication/division (M) extension of the original V-scale processor has been kept to maintain compatibility but is still unprotected.

Therefore, the register file, the majority of the ALU and the data memory interface of the V-scale processor have been protected using the DOM scheme. Other parts, like the instruction memory interface and the decoder have been left unprotected. The reason for this split is that in any case the implemented code must be written such that it does not leak information about the processed data over the instruction sequence because different instructions show different power signatures in leakage traces as also mentioned in [28]. Otherwise, even on a fully shared processor, timing attacks would for example be possible.

The resulting processor's architecture is depicted in Figure 5.1. One major difference to the original V-scale processor is that the protected core now has four pipeline stages. The additional pipeline stage (see (1) in Figure 5.1) splits the previously combined decode+execute stage and is necessary to prevent leakage due to glitches when data shares are merged. This aspect is described in more detail in Section 5.1.

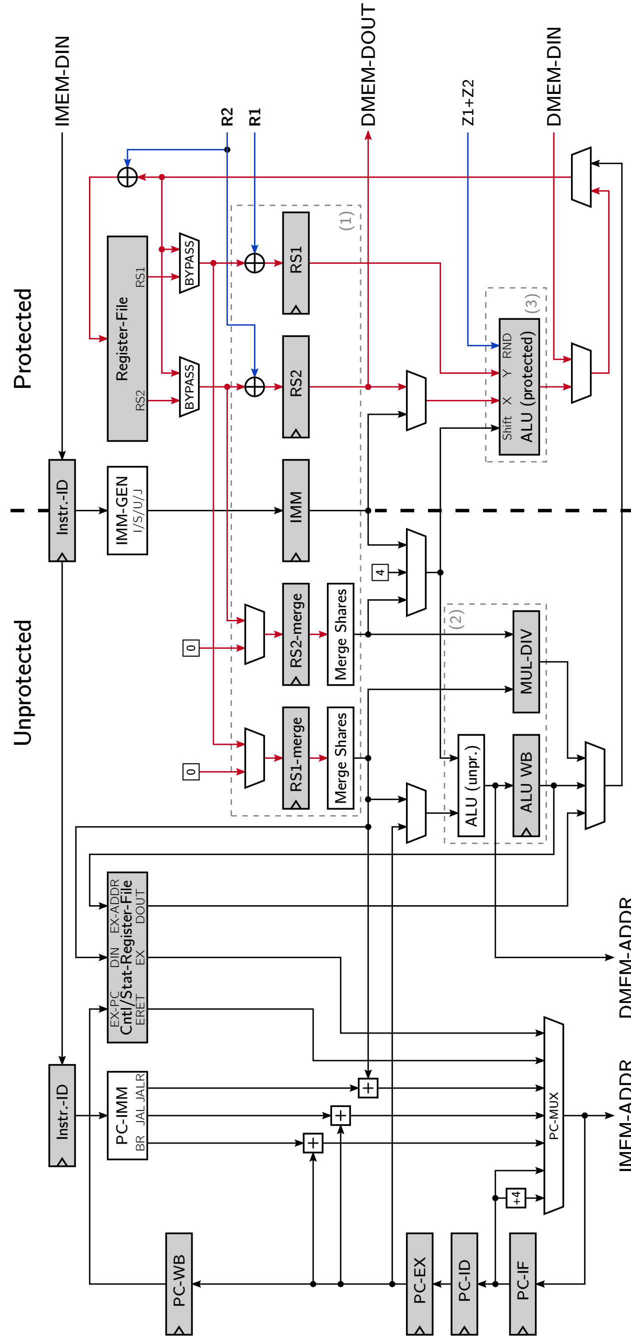

Figure 5.1.: Overview of the protected V-scale core. Grey blocks are registers or use a register stage internal. Shared data connections are illustrated in red, unshared in black and the randomness in blue.

From another perspective, the processor is split into a part that operates on DOM-shared data and a part operating with merged data shares. Accordingly, the ALU itself has been split into a protected and an unprotected part. The unprotected ALU (see (2) in Figure 5.1) implements multiplication/division, address calculation, and data comparison for conditional jumps. Performing comparisons for conditional jumps in an unprotected way is legitimate as code is not allowed to branch on secure data anyway to avoid timing attacks. More details on the logic to securely merge the different DOM shares and on the unprotected ALU itself can be found in Section 5.2. All the remaining functionality being part of the base instruction set (*e. g.* AND, OR, XOR, ADD, . . .) is implemented in the protected ALU in a DOM-protected way. The protected ALU is visualized in Figure 5.1 at (3) and is thoroughly described in Section 5.3.

5.1. Additional Pipeline Stage

The major change to the unprotected processor are the additional source registers shown in Figure 5.1 at (1). The main purpose of these buffer registers is to prevent glitches in the merging units connected to *RS1-merge* and *RS2-merge*. These merging units recombine the shares to the original value as shown in Equation 3.1.

Without the registers *RS1-merge* and *RS2-merge*, (de-)activation of the merging units can result in data dependent glitches. This is illustrated using two basic scenarios. First, the output of the register file switches to sensitive data. This requires the merging units to be disabled by detaching their inputs from the source register. However, if the sensitive data is selected faster than the merging unit is disabled, sensitive data propagates into the merging unit and results in the leakage of sensitive data. Second, the output of the register file switches from sensitive data to data to be merged. This enables the merging unit by switching the multiplexer to the output of the register file. Here, if the multiplexer switches faster than the register file output is selected, the sensitive data from before glitches into the merging units which leaks information. Both scenarios are prevented by the additional buffer registers *RS1-merge* and *RS2-merge*. These effectively decouple the merging

units from the register file selector by setting the input to the merging units to zero if not required. To adapt the delay of the protected to the unprotected data path, further buffer registers *RS1* and *RS2* are needed.

Another change to the processor design is the addition of fresh randomness to the processed values before the ALU result is written back to the register file and before the registers *RS1* and *RS2* are used as the operands for the protected ALU. This allows to restore the independence of the sharings after unprotected operations and shifts operations which generate zeros or duplicate the most significant bit, respectively. Furthermore, the addition of fresh randomness is required right before operating on identical operand registers for protected ALU operations.

5.2. Unprotected Operations

Figure 5.1 shows at (2) the modules *MUL-DIV* and *ALU (unpr.)* providing the unprotected operations of the core. These modules operate natively with 32-bit word size and use the merged data as described in Section 5.1. The *MUL-DIV*-module is the unprotected hardware multiplication and division unit from the original V-scale processor design and kept to maintain compatibility.

The unprotected ALU implements different compare operations, *i. a.*, for branch instructions. However, the comparison results can also be written back to a register. While all branch instructions use two source register inputs, instructions storing the comparison result allow to alternatively use an immediate value as the second source. Note that the compare functionality could have been implemented without merging the data, but branching on protected data must anyway be avoided due to possible timing attacks [29]. This design decision should be kept in mind as it makes it necessary to avoid compare operations on protected data.

Furthermore, the unprotected ALU provides an adder to perform address calculations within load and store operations. Note however that the required merging of source register before the actual address computation does not reduce security. As the second operand is constant and determined

by a known software implementation, the value of the source register can always be reconstructed, also if a masked adder was used and the shares of the memory address were merged afterwards. Besides, the unprotected adder is also used within two further instructions. First, the adder is used in the jump and link instruction to increment the program counter in the computation of the address of the following instruction. Second, in the add upper immediate to program counter instruction both the program counter and the immediate input are publicly known making a masked adder obsolete.

5.3. Protected ALU

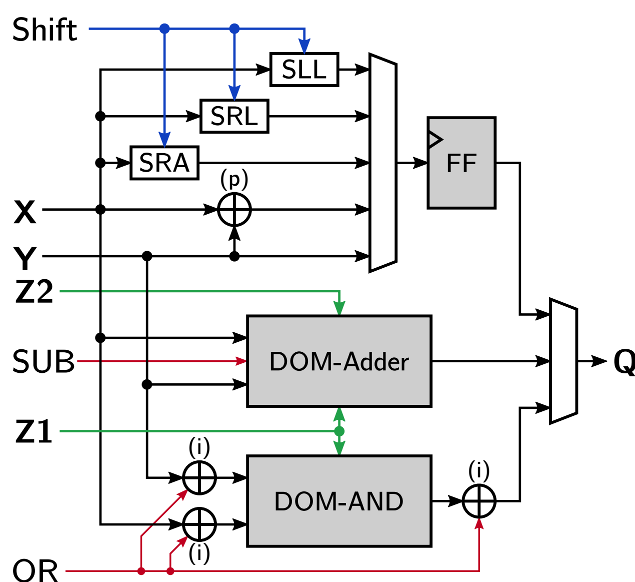

Figure 5.2.: Protected ALU using a single *DOM-AND* for AND and OR operation. Shown XOR operations used in different manner: (p)airwise XOR operation of inputs shares (*e. g.* $A_x + A_y; B_x + B_y; \ldots$); (i)nverting the operand XORing the signal *OR* with every element of the corresponding first share;

The protected ALU is shown in Figure 5.1 at (3) which provides the masked functionality for bit-wise logic operations and arithmetic operations. Both

input sharings **X** and **Y** are composed of $d+1$ independent shares (see Equation 5.2), where d is the protection order of the DOM implementation. For resharing purposes, the protected ALU has two additional inputs **Z1** and **Z2** holding the required fresh random shares. The data width of the input shares and the fresh random Z shares is 32 bits each.

$$\mathbf{X} = \underbrace{(A_x, B_x, C_x, \dots)}_{d+1} \qquad \mathbf{Z1} = \underbrace{(Z1_0, Z1_1, Z1_2, \dots)}_{d(d+1)/2} \tag{5.1}$$

$$\mathbf{Y} = \underbrace{(A_y, B_y, C_y, \dots)}_{d+1} \qquad \mathbf{Z2} = \underbrace{(Z2_0, Z2_1, Z2_2, \dots)}_{d(d+1)/2} \tag{5.2}$$

5.3.1. DOM-AND

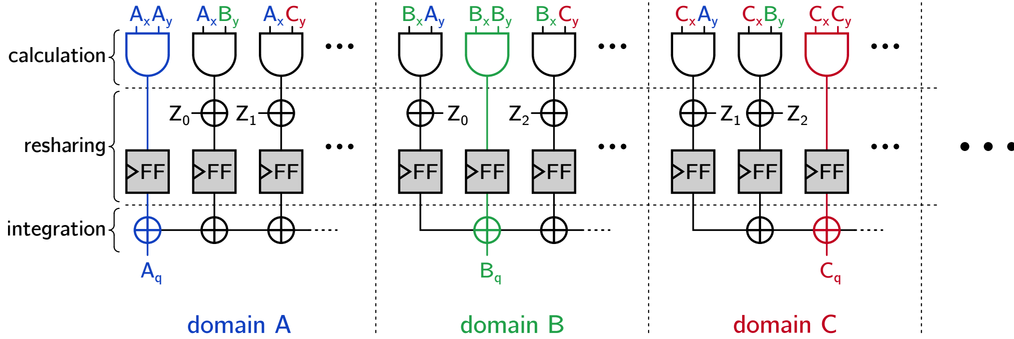

Figure 5.3.: Overview of the DOM-AND.

The basis for all implemented non-linear operations is the so-called DOM-*indep* $GF(2)$ multiplier variant (see [17]) which corresponds to a logic *AND* gate with two one-bit inputs. The DOM-*indep AND* gate is illustrated in Figure 5.3. A basic requirement of the DOM-*indep* multipliers is that the two inputs **X** $(A_x, B_x, C_x, \dots)$ and **Y** $(A_y, B_y, C_y, \dots)$ are independently shared which is ensured by design of the protected core.

The construction of the *DOM-AND* is generic and can thus be extended to arbitrary protection orders by adding additional shares. For the protection order d, $d+1$ shares per variable are required giving $d+1$ independent share domains. Every domain consists of $d+1$ AND gates and flip-flops which results in a quadratical growth of the chip area accordign to the protection order. The three steps (calculation, resharing, and integration) of the DOM implementation are applied independently for every bit position of the 32-bit shares. Therefore, a 32-bit AND gate consists of 32 *DOM-AND* gates.

In the *calculation* step the terms resulting from the calculation of $\mathbf{X} \times \mathbf{Y}$ ($A_x A_y$, $A_x B_y$, $A_x C_y$, $B_x A_y \ldots$) are calculated separately. In the next step (*resharing*) all terms that contain shares which are not associated with the respective domain are reshared by using a fresh random Z share. The subsequent register ensure that no early propagation effects occur which could result in glitches that would effect the SCA resistants of the gate. To keep the timing of the masked *AND* synchronous the register is also inserted in inner-domain paths of the domains (*e.g.*, $A_x A_y$ or $B_x B_y$). The last step reduces the number of terms again to $d+1$ by integrating the freshly masked cross-domain terms into the inner-domain terms and hence generates the output $\mathbf{Q}$ ($A_q, B_q, C_q \ldots$).

5.3.2. Protected Adder

Another important operation of the ALU is the addition. A main requirement for this hardware adder is the possibility to protect it with the DOM scheme and keep the delay as small as possible. Using a design with iterative adder blocks as shown by Gross [28] for example, increases the required clock cycles to the operands width. This iterations are necessary to guarantee the independence requirement of the DOM. Such an implementation is only practicable for small bit widths. To increase the performance using wider input operands, a Kogge-Stone similar construction as shown by Schneider et al. [30] is used.

Kogge-Stone Adder

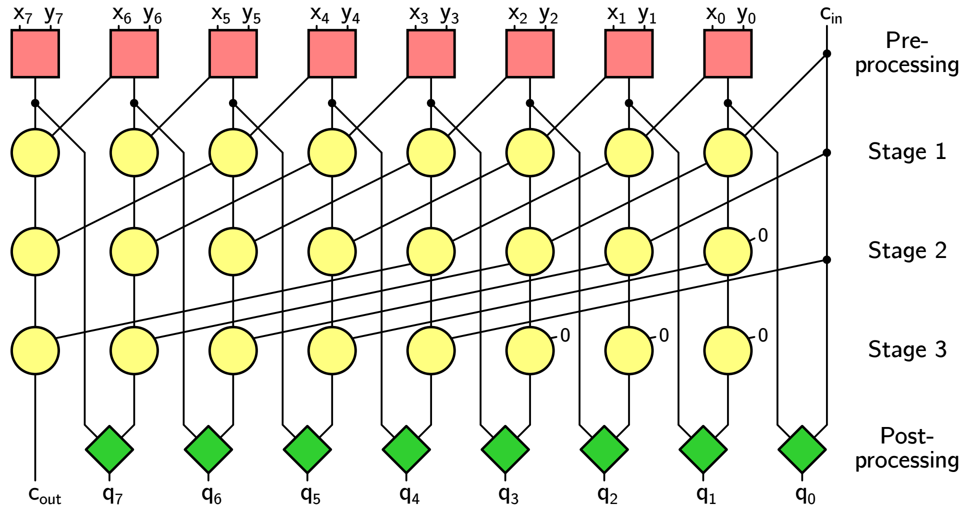

Figure 5.4.: Kogge-Stone adder dependence tree for 8-bit inputs with carry in/out.

This is a parallel prefix adder using a carry lookahead with a tree like structure. An example of an 8-bit adder is shown in Figure 5.4. Therefore, the addition is separated into propagation and carry generation which is illustrated by the yellow nodes in the tree. The main advantage of this structure is the logarithmic logical depth which increases only by a single additional stage when the input operands width is doubled. Furthermore, this adder type provides a good basic structure for a generic implementation, with regard to the bit width of the input operands as well as the protection order of the DOM implementation.

The input operands x and y as well as the output operand q are n-bit wide vectors as shown in Equation 5.3.

$$x = (x_{n-1}, \dots, x_1, x_0) \qquad y = (y_{n-1}, \dots, y_1, y_0) \qquad q = (q_{n-1}, \dots, q_1, q_0) \tag{5.3}$$

The calculation of the sum is performed in three different processing steps. Each generating n-bit width intermediate values for propagation $p^{(i)}$ and the carry generation $g^{(i)}$.

Preprocessing: As shown in Figure 5.4 the addition starts with the preprocessing step to generate the initial values of carry generation $g^{(0)}$ and propagation $p^{(0)}$ (see Equation 5.4).

$$g^{(0)} = xy \qquad\qquad p^{(0)} = x + y \tag{5.4}$$

Processing: The processing step is performed several times depending on the operands data width. Therefore, the number of stages increases logarithmically and is given by $N = \lceil log_2(n) \rceil$. In each stage the new carry generation and propagation values are generated from old values as shown in Equation 5.5. The equations use the $\ll$ operand which indicates a left shift. This shifts are increased with each stage to 2^{i-1} where $i \in \{1 \ldots N\}$ means the current stage.

$$\begin{aligned} g^{(i)} &= p^{(i-1)}(g^{(i-1)} \ll 2^{i-1}) + g^{(i-1)} \\ p^{(i)} &= p^{(i-1)}(p^{(i-1)} \ll 2^{i-1}) \end{aligned} \qquad i = 1 \ldots N \tag{5.5}$$

Postprocessing: The final postprocessing step generates the sum of the addition by shifting the last carry generation value one to the left ($g_N \ll 1$) and add it to the initial propagation sum (p_0) as shown in Equation 5.6.

$$q = p^{(0)} + (g^{(N)} \ll 1) \tag{5.6}$$

The adder design also provides carry input and output as shown in Figure 5.4. To get the carry output of the adder the most significant bit of the last carry propagation value $c_{out} = g_{n-1}^{(N)}$ is used. The carry input is used in each processing state as carry propagation bit $g_{-1}^{(i)} = c_{in}$ and is therefore shifted into the carry propagation path. Additionally to the processing stages, the carry input must be used in the postprocessing step as least significant bit of the shifted carry generation value.

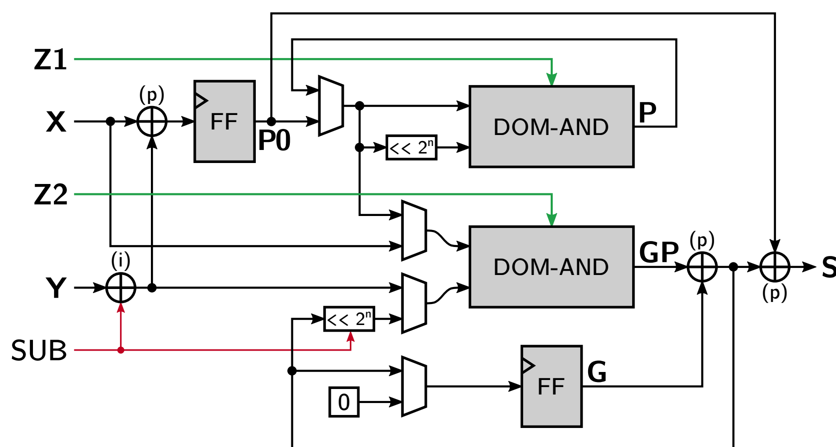

Figure 5.5.: Masked adder using two *DOM-ANDs*. Shown XOR operations used in different manner: (p)airwise XOR operation of inputs shares (*e. g.* $A_x + A_y; B_x + B_y; \ldots$); (i)nverting the operand by adding the signal *SUB* with every element of the first share of **Y** (A_y).

DOM Implementation

Figure 5.5 shows the secure DOM adder. It is composed of two *DOM-ANDs*, two bit shifts, and multiple XORs. The XOR as well as the shift operations can be performed independently for each share domain and each input. The nonlinear parts of the adder are formed by two *DOM-AND* gates. To make the illustration of the adder in Figure 5.5 more concise, the three steps for calculating the DOM-AND are only indicated by the respective function (see Figure 5.3 for more details). The *DOM-AND*'s internal registers together with the **G** are used as the working registers for the iterative calculation of the sum. The *DOM-AND*'s internal registers are indicated by **GP** which belongs to the carry generation path and **P** which belongs to the propagation path.

For the carry generation path the register **G** is used to store the previous value of the generation step as it is required in the next iteration. An important requirement of the used *DOM-AND* gate is an independent sharing the both inputs. This independence is ensured for both AND gates because the bit positions of one operand is always shifted by at least one position. With

the same argument the random Z shares in each cycle are applied for both AND gates without violating the independence requirement.

The subtraction operation can easily performed by calculating the twos-complement of the subtrahend. The subtraction is controlled by the *SUB* input. Therefore, the input signal *SUB* is XORed with every bit of the first share of $\mathbf{Y}$ (A_y). Incrementing the result by one is done by connecting the carry in of the adder with *SUB* which is active on a subtraction. This is done in the shifter of the generation path by appending the carry bit below the least significant bit of the first share and shifting it into the carry generation path. The following equations uses the $\ll$ operation to indicate a left shift performed independently on every input share supporting only shifts with 2^n where $n \geq 0$. The calculation of the sum is performed in three steps called preprocessing, processing, and postprocessing.

An addition is started with the initial preprocessing step initializing the registers $\mathbf{G}$, $\mathbf{P0}$ and $\mathbf{GP}$ according to Equation 5.7.

$$\mathbf{G}_0 = 0 \qquad \mathbf{P0} = \mathbf{X} + \mathbf{Y} \qquad \mathbf{GP}_0 = \mathbf{XY} \tag{5.7}$$

The processing step is performed five times in a row ($n = 1 \ldots 5$).The first and last steps are diverging form the normal processing operation. In the first step the input register $\mathbf{P}$ is replaced by $\mathbf{P0}$. In the last processing step the register update of $\mathbf{P}$ is omitted (see Equations 5.8 to 5.11).

$$\mathbf{G}_n = \mathbf{G}_{n-1} + \mathbf{GP}_{n-1} \qquad n = 1 \ldots N \tag{5.8}$$

$$\mathbf{P}_1 = (\mathbf{P0} \ll 1)\mathbf{P0} \qquad \mathbf{GP}_1 = \mathbf{P0}(\mathbf{G}_1 \ll 1) \tag{5.9}$$

$$\mathbf{P}_n = \mathbf{P}_{n-1}\,(\mathbf{P}_{n-1} \ll 2^{n-1}) \qquad n = 2 \ldots N-1 \tag{5.10}$$

$$\mathbf{GP}_n = \mathbf{P}_{n-1}(\mathbf{G}_n \ll 2^{n-1}) \qquad n = 2 \ldots N \tag{5.11}$$

In the final postprocessing step the resulting sum is simply computed by a single XOR operation as shown in Equation 5.12.

$$\mathbf{S} = \mathbf{P0} + (\mathbf{G}_N \ll 1) \tag{5.12}$$

5.3.3. Resharing of ALU Inputs and Outputs

To reduce the required fresh randomness the two resharing values **R1** and **R2** in Figure 5.1 are generated from the random Z shares. Furthermore, the merged value of both **R** shares is always zero so that an addition of the shares with a sharing of the register file input or output always result in a resharing without changing the underlying value. For first-order protection the resharing value is generated by duplicating a single random share as shown in Equation 5.13.

$$\mathbf{R1} = (Z1_0, Z1_0) \qquad \mathbf{R2} = (Z2_0, Z2_0) \tag{5.13}$$

For other protection orders the randomness is composed as shown in Equations 5.14 and 5.15.

$$\mathbf{R1} = (Z1_0, Z1_0 + Z2_1, Z1_2 + Z2_1, Z1_2 + Z2_3, Z1_4 + Z2_3, \ldots) \tag{5.14}$$
$$\mathbf{R2} = (Z2_0, Z1_1 + Z2_0, Z1_1 + Z2_2, Z1_3 + Z2_2, Z1_3 + Z2_4, \ldots) \tag{5.15}$$

To guarantee the independence of both resharing values the first sharing **R1** uses the shares of **Z1** with even and shares of **Z2** with odd indexes, whereas the second sharing **R2** uses the remaining shares of **Z1** and **Z2**. This combination of both Z shares is necessary to prevent adding of two shares which are also used in the *DOM-AND* for the integration step. For example, if the second term of **R1** uses the same random Z share $(Z1_0 + Z1_1)$ it can be used to eliminate two random values in domain A as shown in Figure 5.3. This reduces the number of signals an attacker has to know to eliminate the randomness.

5.3.4. Other ALU Operations

The remaining operations of the protected ALU (see Figure 5.2) are the shift operations, the logic operations XOR and OR, and the pass-through path. The shift operations are represented by the blocks SLL, SRL and SRA, which

perform logical left or right shift or an arithmetic right shift. The $Shift$ operand uses a separate unshared input for selecting the shift width which is generated outside the module as shown in Figure 5.1. This is necessary to prevent an unwanted merging of the default used shift operand **Y**. The shifts are performed independently on every share of **X**. For the arithmetic right shift the most significant bit of every share is duplicated. The logical shift operations add zeros to the shares. Therefore, the shares must be refreshed which is done before writing back the result into the register file or the buffer registers adding fresh randomness (see Figure 5.1).

The XOR operation is done in a straight-forward way by adding the input shares of **X** and **Y** share wise. This leads to a zero result using the same input values. Again the results are reshared using fresh randomness before storing them in the buffer registers *RS1* and *RS2* to guarantee independence of the shares.

The pass-through applies the second input **Y** unmodified to the output. To prevent a duplication of the sharing of **Y** in different registers, the sharing is again refreshed before writing it to a register.

The OR operation is combined with the AND operation formed by the *DOM-AND* to reduce the logic overhead. This is done by transforming the logical OR into an AND by inverting both inputs and the output. If the OR operation is used, the input *OR* is set which inverts the first share of both input operands as well as the resulting output of the *DOM-AND* by adding to all bits the *OR* signal.

6. FPGA Design and Hardware Results

The evaluation of the protected V-scale implementation is done on a standardized measurement board. This board is one out of a group of so-called side-channel attack standard evaluation boards (SASEBOs) which are especially designed for SCA evaluation processes. The used board in this work is a commercial version of the SASEBO-GIII [31], the SAKURA-X board. Figure 6.1 shows the block diagram with its main components and the picture in Figure 6.2 illustrates the top view of the board with the highlighted components from the block diagram.

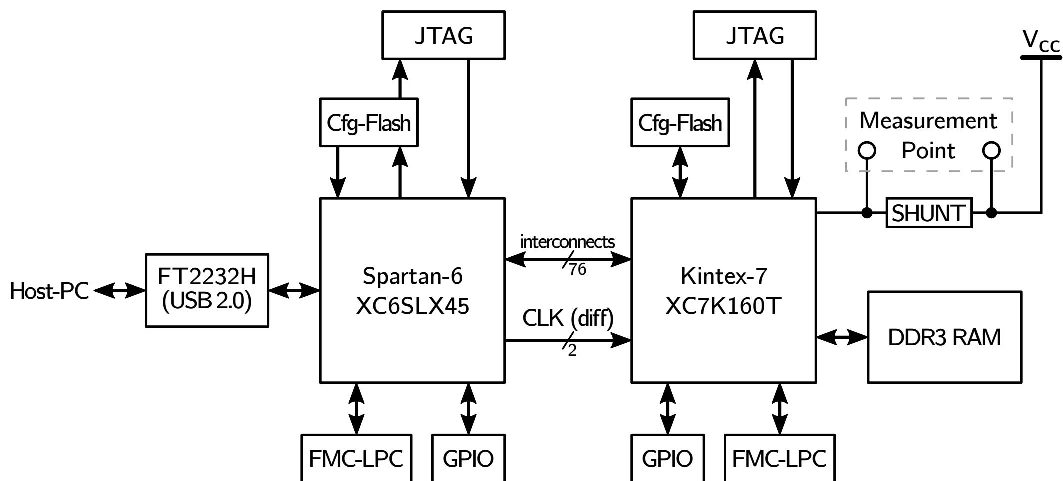

Figure 6.1.: Block diagram of the SAKURA-X board.

This board is equipped with two Xilinx field-programmable gate arrays (FPGAs). One Xilinx Spartan-6 FPGA device working as controller connected

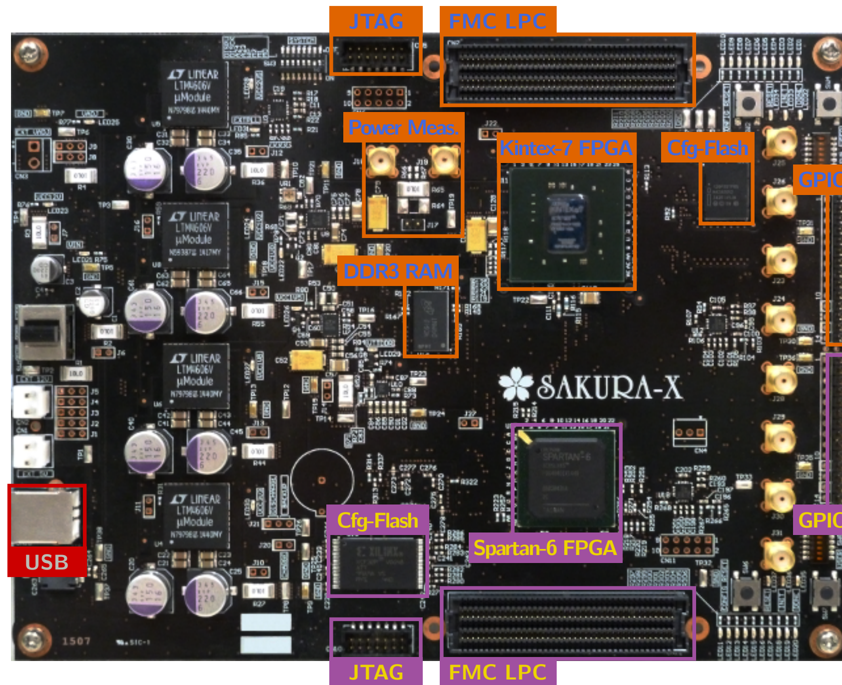

Figure 6.2.: Top view with main components of the SAKURA-X board.

to the measurement PC, and the second Xilinx Kintex-7 FPGA implements the DUA. For this purpose the FPGAs are connected over a local bus interface with a maximum of 78 interconnects, where two connections are especially declared as differential clock signal. The data connection of the control FPGA with the host PC uses a configurable protocol chip providing different industrial serial and parallel interfaces (*e. g.*, UART, SPI, JTAG, . . .). For expandability reasons an FPGA mezzanine card (FMC) low pin count (LPC) connector is available for each device. The configuration of both FPGAs can be done over separate joint test action group (JTAG) interfaces or by the provided flash memories, each connected via byte peripherial interface (BPI).

The connection for the power measurement is realized over two SMC connectors. Alternatively, the measurement points can also be accessed over a pin header. Both are directly connected to the shunt resistor allowing voltage measurements with a differential amplifier. This makes a better use

of the maximum measurement range of an oscilloscope.

6.1. Device Under Attack – Kintex-7

The implementation of the attacked V-scale core is simulated and synthesized with the Xilinx Vivado Design Suite 2014.3. Figure 6.3 illustrates the block diagram of the main modules running on the Kintex-7 FPGA.

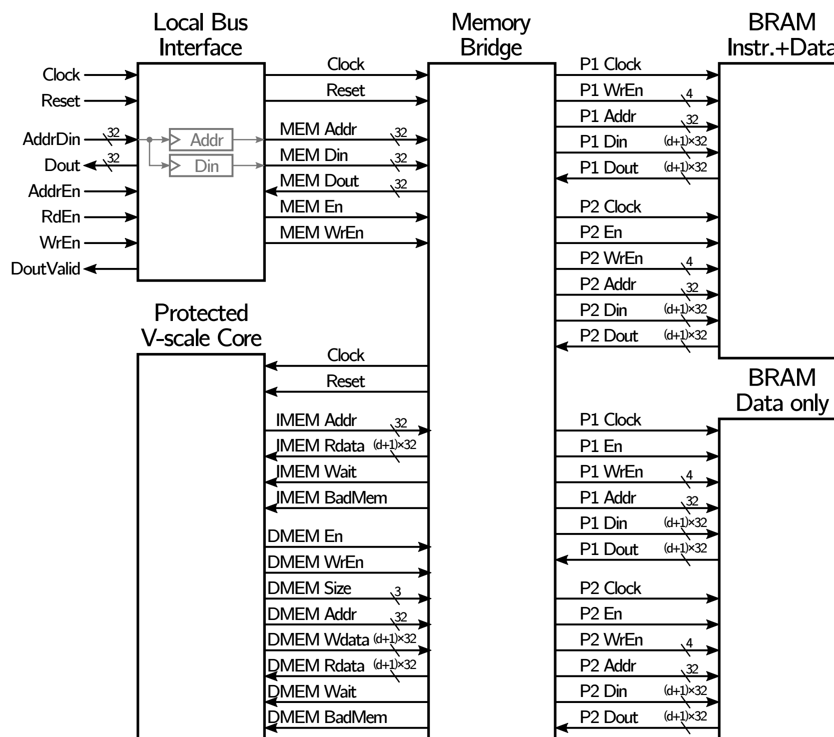

Figure 6.3.: Block diagram of the device under attack (Kintex-7).

Details on the implementation of the protected V-scale core design can be found in Chapter 5. Communication of the V-scale core provides two separated memory interfaces for instructions and data. For this purpose the internal block RAMs (BRAMs) provided by the Kintex-7 FPGA are utilized. This BRAM support native sizes of 18Kbits or 36Kbits and can be

direct cascaded for larger memory requirements. Furthermore, different bit widths and interface types can be configured. To reduce the delay when data is loaded while fetching an instruction, a true dual-port interface is used for the BRAM. To access the data memory over the local bus interface without stopping the V-scale core requires two separated memory blocks with different accessibility:

Instruction+Data: The first BRAM memory block is used to hold the program as well as data, and is therefore accessible over both interfaces of the V-scale core. This memory always starts at address 0. To load a new program the memory bridge can link the local bus interface to the first port **P1** which is normally used by the instruction interface of the V-scale core.

Data only: The second BRAM memory block uses the first port **P1** to connect the data interface of the V-scale core. To load any data via the local bus interface without interrupting the execution of the V-scale core the second port **P2** is used which restricts this memory to be used only for data. The start address of this memory block depends on the size of the instruction memory and can be changed in the memory bridge.

6.1.1. Local Bus Interface

The data requests from the local bus are handle over the interface module. All data exchanges are synchronous and uses a separate input and output port with 32 bit width. To perform a read or write operation, the two independent signals **RdEn** and **WrEn** are used, respectively it is possible to perform reading and writing simultaneously. With the **AddrEn** signal the destination can be changed from the data to the address register. Furthermore, the signal can be used to read the current address instead of the data coming from the memory bridge. Valid data applied to the data output **Dout** is signaled by **DoutValid**.

Most read or write operations are performed on consecutive data blocks. Therefore, it is necessary to increment the address. To reduce the data overhead by manually writing the new address it can be incremented

automatically after read of write operations. Since all addresses are 4 byte aligned the two least significant bits of the address register are used for this purpose. Setting bit 0 of the address increments on read operations whereas setting bit 1 does the same on write operations.

6.1.2. Memory Bridge

The memory bridge has to perform several tasks. It connects the V-scale core and the local bus interface with the two memory blocks and provides special access to the instruction memory for program loading. Therefore, a control register is used in the bridge which is mapped into the address space of the local bus interface. It provides three bits, where bit 0 is used to reset the V-scale core. Bit 1 change the access from the first port **P1** of the instruction memory block to the local bus interface and additionally sets the memory wait signal to prevent the V-scale core from fetching data. An additional bit at position 2 in the control register is applied to the data memory wait signal and can be used to prevent the V-scale core loading any data from the memory.

The V-scale core uses a direct address mapping to the memory blocks. This is not possible for the local bus interface as it works with 32 bits data width whereas the protected implementation requires $32 \times (d+1)$ bits. Therefore, the upper 8-bit of the address coming from the local bus interface are used to select the desired share and to switch between the instruction and data memory blocks.

6.2. Control FPGA – Spartan-6

For the simulation and synthesis of the control logic running on the Spartan-6 FPGA, the Xilinx ISE Design Suite 14.7 is used. This controller supports read and write operations to exchange data with the Kintex-7 FPGA. Therefore, the host PC is connected over USB to the SAKURA-X board using the onboard converter chip (FTDI FT2232H) providing an asynchronous parallel transfer interface with a bidirectional 8-bit data bus and a simple

4 wire handshake interface. The block diagram in Figure 6.4 illustrates the modules running on the Spartan-6 FPGA handling the data transfers from the converter chip to the local bus interface. All data transfers are buffered in the two "first in, first out" (FIFO) memories with a data width of 8 bits. The upper FIFO in Figure 6.4 is used to buffer data coming from the host PC whereas the lower FIFO buffers data coming from the local bus.

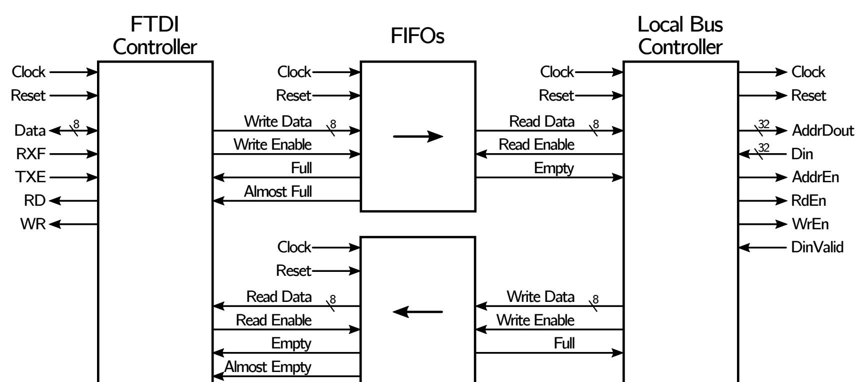

Figure 6.4.: Block diagram of the control FPGA (Spartan-6).

6.3. Hardware Results of V-scale

The hardware results are gathered for a Xilinx Kintex-7 FPGA with the Xilinx Vivado Design Suite 2014.3. Therefore, the synthesis was done for the unprotected core as well as for the protected V-scale core with protection orders from 1 up to 4. Figure 6.5 shows the evolution of required look up tables (LUTs) (left) as well as the required registers (right) for increasing protection order. The overall area seems to grow only linearly with the protection order. The design of the *DOM-AND* gates which are part of the nonlinear modules of the protected ALU increase quadratically which, however, contribute only marginally to the overall size for lower protection orders.

Table 6.1 shows the area result in numbers. Additional the required randomness is shown which increases quadratically with the protection order. In

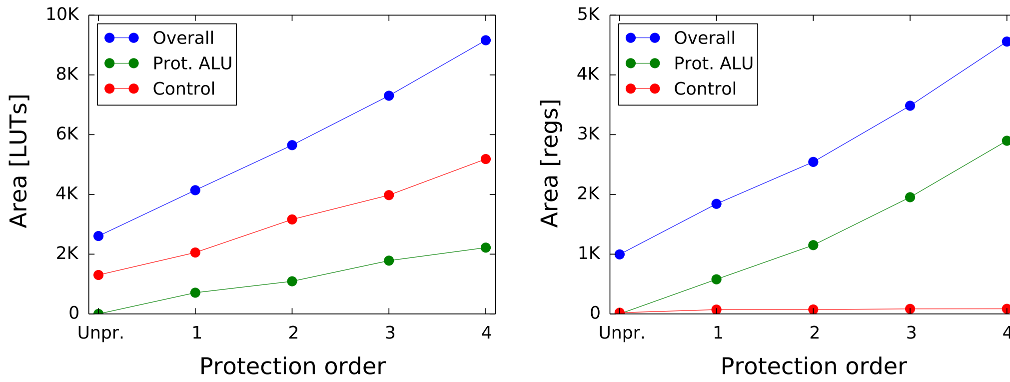

Figure 6.5.: Required LUT (left) and registers (right) on an FPGA.

Prot. Order	FPGA Logic		Randomness	Max. Clock
d	*[LUTs]*	*[regs]*	*[Bits]*	*[MHz]*
Unpr.	2,607	996	0	45.6
1	4,143	1,842	64	61.0
2	5,626	2,551	192	59.5
3	7,259	3,484	384	58.3
4	9,244	4,561	640	41.0

Table 6.1.: V-scale core implementation results.

particular the randomness required for the protected ALU is $32 \times d(d+1)$ bits in each cycle. The last column shows the maximum clock frequency which is higher for the protection orders 1 up to 3 as for the unprotected implementation. This results from the additional pipeline stage of the protected implementation which reduces the critical path but increases the delay on the other hand.

7. Evaluation

The security of the DOM implemented V-scale core in the d-probing model is discussed in Chapter 5. To show practical evidence for the first-order resistance of the protected V-scale design, the Welch's t-test is used according to the recommendations of Ding et al. [32].

To make the leakage assessment as reproducible as possible, the SASEBO-GIII [31] based SAKURA-X FPGA board is used. The board is especially designed for side-channel evaluation and provides special measurement connectors for measuring the power consumption. The leakage traces are collected by a Picoscope 6404C oscilloscope at 312.5 Ms sampling rate for an 8 MHz DUA clock. An implementation of the round transformations of an authenticate encryption scheme (Ascon) together with additional code that triggers particular instructions and instruction sequences that are considered critical is used as the targeted software implementation.

7.1. T-test Based Leakage Detection

For this work it is more interesting to detect if a DUA leaks information which can be used for a practical attack. The attacks based on the analysis of the measured power traces use statistical dependencies of the processed data with the consumed power. With the t-test based leakage detection, it can be shown whether or not an arbitrary design leaks security critical information. The t-test requires two different power measurement sets for the investigated cryptographic algorithm. For the first set L_A, a fixed input is used whereas the second test set L_B takes randomly chosen inputs. This two sets of measured power traces are compared together using the Welch's t-test as shown by Ding et al. [32]. Therefore, the test methodology uses

the difference of the sample means $\overline{L}_A - \overline{L}_B$ which is then scaled to the estimated standard deviations s_A and s_B with respect to the size of the trace sets denoted by n_A and n_B, respectively. With this values the t-test statistic is given by:

$$t = \frac{\overline{L}_A - \overline{L}_B}{\sqrt{\frac{s_A^2}{n_A} + \frac{s_B^2}{n_B}}} \tag{7.1}$$

If the t-value exceeds the confidence interval of ± 4.5 the null-hypothesis is rejected with confidence greater than 99.99% for large size of traces n_A and n_B [32].

The t-test methodology can also detect leakage in higher-order protected designs. Therefore, the size of both trace sets must be the same ($n = n_A = n_B$). For a d-th order leakage detection at a single point of time, the sample traces of both sets are compared by freeing their samples from the corresponding sample mean and exponentiated by the order of leakage detection:

$$D = (L_A - \overline{L}_A)^d - (L_B - \overline{L}_B)^d \tag{7.2}$$

To get the t-test statistic for higher-order leakage detection with Equation 7.3, the standard deviation s_D and the mean $\overline{D}$ is calculated from the paired differences D.

$$t = \frac{\overline{D}}{\sqrt{\frac{s_D^2}{n}}} \tag{7.3}$$

7.2. Unprotected Device

The first-order t-test is performed according to Equation 7.1 for the two trace sets A and B with random and constant inputs. A t-test is performed on the unprotected device which serves as reference for further evaluations.

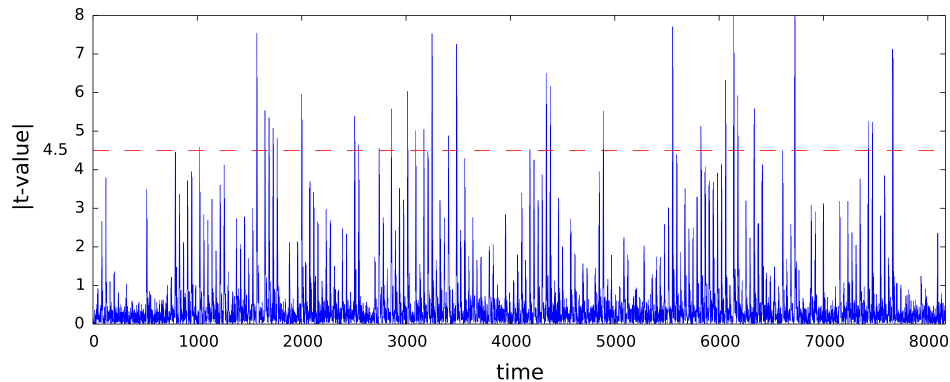

Figure 7.1.: First order t-test values of unprotected device with 20K traces.

Figure 7.2 shows the absolute t-values where multiple peaks exceed the 4.5 border with only 20,000 measurement traces for both sets. The small peaks indicate positive clock edges changing the values stored in the registers and the subsequent switching activity of the logic applying the new values. One difference to the protected device is the shorter length of the traces for the same executed operations. This comes from the shorter delay when a value in the write-back stage of the processor is used in the next operation. The protected implementation has to wait one cycle until the value is available.

7.3. First Order Protected Device

The t-test on the first order protected device is performed under different conditions.

Random Number Generator Turned Off: To show the functionality of the measurement setup, the internal random number generator is turned off for the first measured trace sets. Figure 7.2 shows the t-values with the expected significant peaks over the 4.5 border which indicates first-order side-channel leakage for 2 million traces per trace set. Compared to the unprotected device the number of required traces increases drastically to get roughly the same maximum t-value. Additionally, the number of peaks exceeding

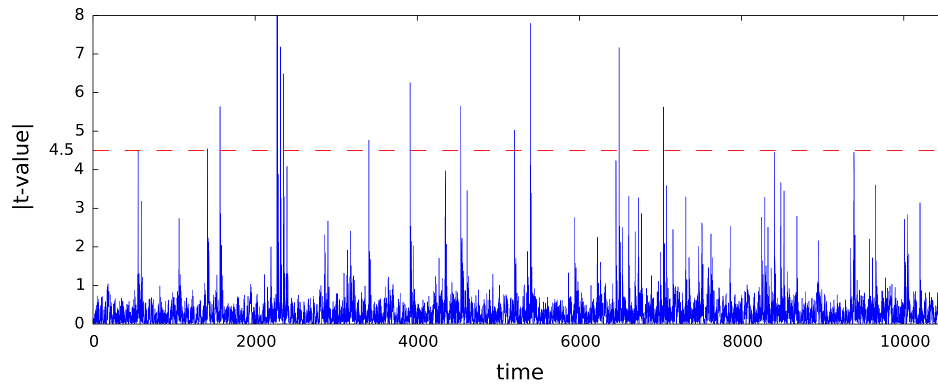

Figure 7.2.: First order t-test value of protected ALU operations with inactive random generator with $2M$ traces.

the 4.5 border is reduced. This is due to the uniformly distributed input data which does not leak any informations until performing an operation requiring fresh randomness.

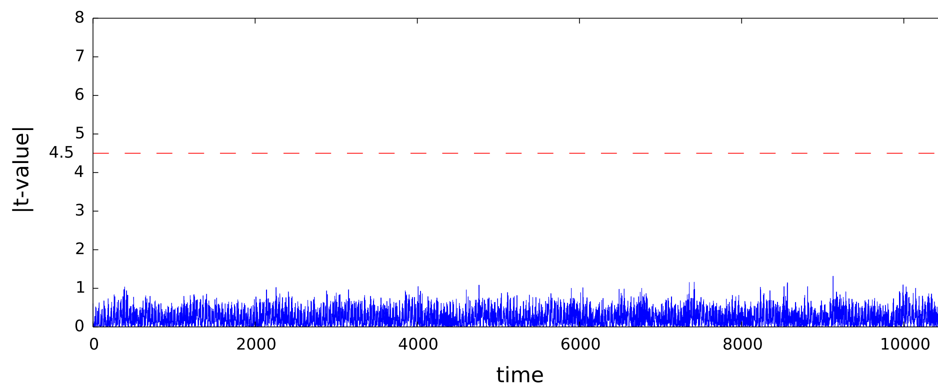

Figure 7.3.: First order t-test value of protected ALU operations with active random generator with $100M$ traces.

Random Number Generator Turned On: The t-test is repeated with the random number generators turned on and for 100 million traces. Even with 50 times more traces compared to the first t-test the leakage evaluation does not show any significant peaks any more (see Figure 7.3). Therefore, the

side-channel countermeasures are considered to work as expected.

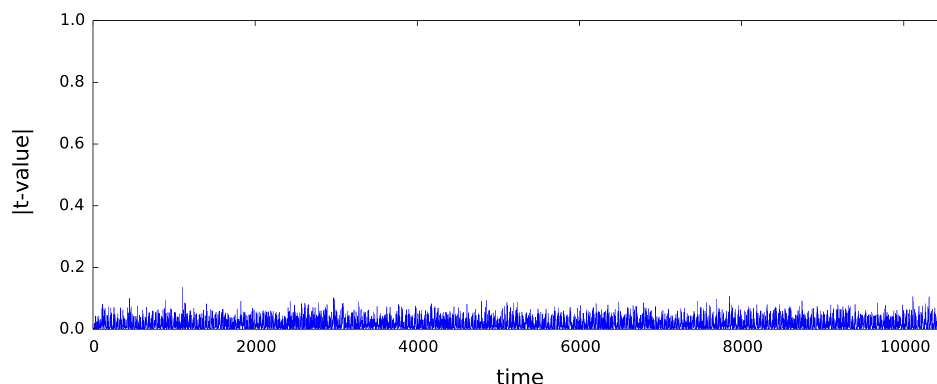

Figure 7.4.: Second order t-test value of protected ALU operations with active random generator with $11M$ traces.

Second-Order Leakage Detection: Additionally to the first-order leakage detection a second-order t-test is performed according to Equation 7.2 with $d = 2$. Compared to the first-order t-test with activated random number generator, only 11 million traces are generated. This is due to the enormous amount of required memory to store the recorded traces. For the first-order t-test the data can be reduced by calculating intermediate values without storing every trace. The second-order t-test requires the centering terms, represented by the subtraction of the mean $\overline{L}$ (see Equation 7.2), which is only possible after finishing the measurement. Therefore, all traces have to be stored which reduces the number of possible traces for the t-test. If enough traces are processed there should emerge some peaks in the t-values indicating the second-order leakage for the first-order protected device. As shown in Figure 7.4 the values are extremely small which indicates that the number of traces are too small.

8. Conclusion

In this thesis, a side-channel protected V-scale core following the DOM scheme was implemented. The implemented core is fully scalable in terms of protection order and allows to protect informational assets that are processed by the protected V-scale core. The overhead for the side-channel protected implementation of the core increases only to a factor of roughly 1.5 for the first-order implementation. Synthesis of the processor with different protection orders up to four increase the size of the core linearly. This results from the fact that the protected ALU, containing the non-linear modules which grow quadratically, is relatively small compared to other parts of the processor like the register file which grows linearly with the protection order.

To show the resistance of the protected implementation against side-channel analysis attacks, a first-order t-test is performed. Therefore, a SAKURA-X FPGA evaluation board is used to run the core and measure the power consumption. The practical evaluation even with 100 million leakage traces does not show any statistical significance. However, a practical evaluation is of course never complete nor a complete argument for the security of an implementation. The expected leakage from the second-order t-test can not be shown due to the huge amount of required power traces. The formal analysis of the implementation is thus considered as part of future work. Furthermore, the security of the design is in general only given for software that does not introduce any control flow changes based on the asset one tries to protect (timing attacks). However, we do not consider this much of a drawback since constant runtime implementations are a basic requirement of protected software and hardware.

Appendix

Appendix A.

Abbreviations

ABI	application binary interface	24
ALU	arithmetic logic unit	31–33, 35, 37–41, 46
BPI	byte peripherial interface	49
CBM	classical boolean masking	13, 14, 16, 17, 20
CSR	control and status register	28, 29, 31
DOM	domain-oriented masking	17–21, 35, 37, 40–42, 44, 46, 47, 53, 55, 60

DPA	differential power analysis	7, 10
DUA	device under attack	7, 49, 55
FMC	FPGA mezzanine card	49
FPGA	field-programmable gate array	48–50, 52, 53
HW	Hamming weight	8–10
ISA	instruction-set architecture	22, 23, 27, 28, 30
JTAG	joint test action group	49
LPC	low pin count	49
SASEBO	side-channel attack standard evaluation board	48, 55
SCA	side-channel analysis	iii, 1, 2, 5, 8, 48
SPA	simple power analysis	7
TI	threshold implementation	1, 14–17, 20

Bibliography

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '99 (1999), pp. 388–397 (cit. on p. 1).

[2] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards," English, in *Smart card programming and security*, Vol. 2140, edited by I. Attali and T. Jensen, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2001), pp. 200–210 (cit. on p. 1).

[3] L. Goubin and J. Patarin, "DES and Differential Power Analysis The Duplication Method," English, in *Cryptographic hardware and embedded systems*, Vol. 1717, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 1999), pp. 158–172 (cit. on p. 1).

[4] E. Trichina, "Combinational logic design for AES subbyte transformation on masked data," IACR Cryptology ePrint Archive **2003**, 236 (2003) (cit. on p. 1).

[5] Y. Ishai, A. Sahai, and D. Wagner, "Private Circuits: Securing Hardware against Probing Attacks," English, in *Advances in cryptology - crypto 2003*, Vol. 2729, edited by D. Boneh, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2003), pp. 463–481 (cit. on pp. 1, 2, 12).

[6] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold Implementations Against Side-Channel Attacks and Glitches," English, in *Information and communications security*, Vol. 4307, edited by P. Ning, S. Qing, and N. Li, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2006), pp. 529–545 (cit. on p. 1).

[7] B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. Van Assche, "Efficient and First-Order DPA Resistant Implementations of Keccak," English, in *Smart card research and advanced applications*, edited by A. Francillon and P. Rohatgi, Lecture Notes in Computer Science (Springer International Publishing, 2014), pp. 187–199 (cit. on p. 1).

[8] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Higher-Order Threshold Implementations," English, in *Advances in cryptology – asiacrypt 2014*, Vol. 8874, edited by P. Sarkar and T. Iwata, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2014), pp. 326–343 (cit. on p. 1).

[9] B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz, "Threshold Implementations of All 3x3 and 4x4 S-Boxes," English, in *Cryptographic hardware and embedded systems – ches 2012*, Vol. 7428, edited by E. Prouff and P. Schaumont, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2012), pp. 76–91 (cit. on p. 1).

[10] O. Reparaz, "A note on the security of Higher-Order Threshold Implementations," IACR Cryptology ePrint Archive **2015**, 1 (2015) (cit. on p. 1).

[11] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "A More Efficient AES Threshold Implementation," English, in *Progress in cryptology – africacrypt 2014*, Vol. 8469, edited by D. Pointcheval and D. Vergnaud, Lecture Notes in Computer Science (Springer International Publishing, 2014), pp. 267–284 (cit. on p. 1).

[12] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Trade-Offs for Threshold Implementations Illustrated on AES," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **34**, 1188–1200 (2015) (cit. on p. 1).

[13] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, "Pushing the Limits: A Very Compact and a Threshold Implementation of AES," in Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology, EUROCRYPT'11 (2011), pp. 69–88 (cit. on p. 1).

[14] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, "Consolidating Masking Schemes," in Advances in cryptology - CRYPTO 2015 - 35th annual cryptology conference, santa barbara, ca, usa, august 16-20, 2015, proceedings, part I, Vol. 9215, edited by R. Gennaro and M. Robshaw, Lecture Notes in Computer Science (2015), pp. 764–783 (cit. on p. 2).

[15] C. Chen, M. Farmani, and T. Eisenbarth, *A Tale of Two Shares: Why Two-Share Threshold Implementation Seems Worthwhile-and Why it is Not*, Cryptology ePrint Archive, Report 2016/434, `http://eprint.iacr.org/2016/434`, 2016 (cit. on p. 2).

[16] T. D. Cnudde, O. Reparaz, B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen, *Masking AES with d+1 Shares in Hardware*, Cryptology ePrint Archive, Report 2016/631, `http://eprint.iacr.org/2016/631`, 2016 (cit. on p. 2).

[17] H. Gross, S. Mangard, and T. Korak, *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*, Cryptology ePrint Archive, Report 2016/486, `http://eprint.iacr.org/2016/486`, 2016 (cit. on pp. 2, 19, 40).

[18] S. M. D. Pozo and F.-X. Standaert, *A note on the security of threshold implementations with $d+1$ input shares*, Cryptology ePrint Archive, Report 2016/420, `http://eprint.iacr.org/2016/420`, 2016 (cit. on p. 2).

[19] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*, tech. rep. UCB/EECS-2016-118 (EECS Department, University of California, Berkeley, May 2016) (cit. on p. 2).

[20] Y. Lee, A. Ou, and A. Magyar, *V-scale*, `https://github.com/ucb-bar/vscale`, 2013 (cit. on p. 2).

[21] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks - revealing the secrets of smart cards* (Springer, 2007) (cit. on p. 4).

[22] S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan, "Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases," English, in *Advances in Cryptology – EUROCRYPT 2010*,

Vol. 6110, edited by H. Gilbert, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2010), pp. 135–156 (cit. on p. 12).

[23] M. Rivain and E. Prouff, "Provably Secure Higher-Order Masking of AES," English, in *Cryptographic Hardware and Embedded Systems, CHES 2010*, Vol. 6225, edited by S. Mangard and F.-X. Standaert, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2010), pp. 413–427 (cit. on p. 12).

[24] S. Chari, C. Jutla, J. Rao, and P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," English, in *Advances in cryptology crypto 99*, Vol. 1666, edited by M. Wiener, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 1999), pp. 398–412 (cit. on p. 12).

[25] S. Mangard and K. Schramm, "Pinpointing the side-channel leakage of masked AES hardware implementations," in Cryptographic hardware and embedded systems - CHES 2006, 8th international workshop, yokohama, japan, october 10-13, 2006, proceedings, Vol. 4249, edited by L. Goubin and M. Matsui, Lecture Notes in Computer Science (2006), pp. 76–90 (cit. on p. 14).

[26] B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, N. Tokareva, and V. Vitkup, "Threshold Implementations of Small S-boxes," Cryptography and Communications **7**, 3–33 (2015) (cit. on p. 15).

[27] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, *The risc-v instruction set manual, volume i: user-level isa, version 2.1*, tech. rep. UCB/EECS-2016-118 (EECS Department, University of California, Berkeley, May 2016) (cit. on p. 27).

[28] H. Gross, "Sharing is caring—on the protection of arithmetic logic units against passive physical attacks," English, in *Radio frequency identification*, Vol. 9440, edited by S. Mangard and P. Schaumont, Lecture Notes in Computer Science (Springer International Publishing, 2015), pp. 68–84 (cit. on pp. 35, 41).

[29] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in cryptology — crypto '96: 16th annual international cryptology conference santa barbara, california, usa august 18–22, 1996 proceedings*, edited by N. Koblitz (Springer Berlin Heidelberg, Berlin, Heidelberg, 1996), pp. 104–113 (cit. on p. 38).

[30] T. Schneider, A. Moradi, and Güneysu, *Arithmetic Addition over Boolean Masking - Towards First- and Second-Order Resistance in Hardware*, Cryptology ePrint Archive, Report 2015/066, 2015 (cit. on p. 41).

[31] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh, "SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA," in The 1st IEEE Global Conference on Consumer Electronics 2012 (Oct. 2012), pp. 657–660 (cit. on pp. 48, 55).

[32] A. A. Ding, C. Chen, and T. Eisenbarth, "Simpler, faster, and more robust t-test based leakage detection," IACR Cryptology ePrint Archive **2015**, 1215 (2015) (cit. on pp. 55, 56).