Imperial College London

Department of Computing

Reconfigurable Architectures for Cryptographic Systems

Adrien Le Masle

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

12th January, 2013

Declaration

I herewith certify that all material in this dissertation which is not my own work has been duly acknowledged.

Adrien Le Masle

Abstract

Field Programmable Gate Arrays (FPGAs) are suitable platforms for implementing cryptographic algorithms in hardware due to their flexibility, good performance and low power consumption. Computer security is becoming increasingly important and security requirements such as key sizes are quickly evolving. This creates the need for customisable hardware designs for cryptographic operations capable of covering a large design space. In this thesis we explore the four design dimensions relevant to cryptography – speed, area, power consumption and security of the crypto-system – by developing parametric designs for public-key generation and encryption as well as side-channel attack countermeasures. There are four contributions.

First, we present new architectures for Montgomery multiplication and exponentiation based on variable pipelining and variable serial replication. Our implementations of these architectures are compared to the best implementations in the literature and the design space is explored in terms of speed and area trade-offs.

Second, we generalise our Montgomery multiplier design ideas by developing a parametric model to allow rapid optimisation of a general class of algorithms containing loops with dependencies carried from one iteration to the next. By predicting the throughput and the area of the design, our model facilitates and speeds up design space exploration.

Third, we develop new architectures for primality testing including the first hardware architecture for the NIST approved Lucas primality test. We explore the area, speed and power consumption trade-offs by comparing our Lucas architectures on CPU, FPGA and ASIC.

Finally, we tackle the security issue by presenting two novel power attack countermeasures based on on-chip power monitoring. Our constant power framework uses a closed-loop control system to keep the power consumption of any FPGA implementation constant. Our attack detection framework uses a network of ring-oscillators to detect the insertion of a shunt resistor-based power measurement circuit on a device's power rail. This countermeasure is lightweight and has a relatively low power overhead compared to existing masking and hiding countermeasures.

Acknowledgements

I would like to express my greatest gratitude to my supervisor Prof. Wayne Luk for pointing at the right direction and giving me invaluable guidance throughout my research at Imperial College London. I am grateful to my second supervisor Dr. Cristian Cadar for his insightful comments on my work.

I would like to acknowledge Gary Chow for his help on designing the constant power framework presented in chapter 6, in particular its proposed architecture for the power consumer.

I am grateful to Professor Csaba Andras Moritz for giving me invaluable support and suggestions on my work, as well as all the people at BlueRISC: Sylvia Moritz, Kris Carver, Jeff Gummeson, Felipe Vilas-Boas and Alan Boulanger.

I express my gratitude to my fellow colleagues in the Custom Computing Group of Imperial College London: Dr. Tobias Becker, Brahim Betkaoui, Thomas Chau, Kit Cheung, Gary Chow, Dr. Gabriel De Figueiredo Coutinho, Stewart Denholm, Dr. Andreas Fidjeland, Gordon Inggs, Maciej Kurek, Qiwei Jin, Nicholas Ng, Xinyu Niu, Dr. Timothy Todman, Dr. Anson Tse, and Dr. Brittle Tsoi for their friendly support.

I would like to thank my family and my long-time friends Aurélie Villaume, Emilie Dupetit-magneux, Yoann Gasque, and Jérôme Raymond for their love and encouragement.

Special thanks to all my UK friends for providing happiness and unforgettable memories to my PhD life, in particular: Nuri Purswani, Felipe Nascimento, Sofia Galligani, Sabine Wilke, Laith Sami, Magdalena Wiktoria Sliwinska, Ian Gladman, Cindy Bhagwandin, Florence Picoli, Tiffany Boutwood, Katy Evans, Joseph K. Ansah, Edward L. Albino Torres, and Kathleen Diane Gosling.

The financial support of BlueRISC and UK EPSRC is gratefully acknowledged.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) are suitable platforms for implementing cryptographic algorithms in particular, and computationally demanding applications in general – this field is known as reconfigurable computing. First, the structure of FPGAs makes them particularly suitable for pipelined applications, which is the case for most of the basic cryptographic operations. Second, FPGAs can be used to embed security into low power environments keeping very good performance. Finally, a pure hardware implementation of a cryptographic algorithm is usually less vulnerable than its software counterparts which are usually run in a multi-tasking operating system.

FPGAs are quickly increasing in capability and in size, and it becomes a growing challenge to fully cover the design space available for a given budget and time to market. In parallel, as computer security is becoming increasingly important, protocols and security requirements are quickly evolving. For example an established algorithm may require increased key lengths to resist cryptanalytic attacks for a few more decades [NIS12]. How can we make use of the extra area provided by new devices or update a crypto-system with new security requirements without going through the whole design process? This question illustrates the need for parametric and customisable hardware designs for cryptographic operations capable of covering a large design space.

The three dimensions that a designer commonly has to deal with are speed, area and

power consumption. The main advantages of parametric designs are their scalability and their reusability. They can be reused as IP cores in many projects with different trade-offs. However, finding the optimal values for the parameters to meet a given design goal can be difficult and time-consuming. An exhaustive approach implementing designs for all the values of the parameters in a given search interval cannot be achieved in most projects where the time-to-market is an issue. Therefore we need models that allow rapid optimisation of design parameters and facilitates design space exploration.

A fourth dimension to consider when specifically dealing with cryptographic applications is the security of the crypto-system (or its resistance to attacks). Attacks can be divided into two categories: attacks on the encryption scheme and attacks on the implementation. Encryption algorithms are designed to resist brute-force attacks and cryptanalysis, provided that a proper key size is chosen. Hence the strength of the encryption scheme relies on choosing an approved algorithm [ITLN09,NIS01] together with a suitable key size. The key size is chosen according to the time the information needs to be kept secret [NIS12]. However, the physical implementation of an encryption algorithm can leak information and create security flaws. Attacks exploiting these physical flaws are called side-channel attacks.

Since their initial publication [KJJ99], a type of side-channel attack called power attacks have been extensively studied. Power attacks recover the key of the encryption algorithm by using one or multiple power traces, which are directly correlated to the switching activity of the transistors inside the device. They have been successfully demonstrated on many common encryption methods, including private key encryption such as DES [SOQP04] and AES [SMPQ06], finite field based public key encryption such as RSA [MDS99] and Diffie-Hellman, and elliptic curve based public key encryption [OOP03]. Theoretically, power attacks can be used to attack any crypto-system with key-dependent power consumption. Hence proper countermeasures against these attacks need to be incorporated and security of the implementation needs to be considered.

In this thesis we develop parametric designs for cryptographic applications. Our designs are especially useful for key generation and encryption in public-key cryptography. The design space is explored in terms of speed, area, power consumption and security. A technology-independent

parametric model is developed to allow rapid optimisation under speed and area constraints of a general class of algorithms, which include some of our cryptographic applications. Resistance to attacks is tackled by presenting two new power attack countermeasures based on on-chip power monitoring. The following section presents our contributions in more detail.

1.2 Contributions

The major contributions of this thesis are divided into the following parts. Our research contributions are also summarised in Table 1.1.

Parametric designs for public-key encryption

We present new hardware architectures for Montgomery modular multiplication and exponentiation. In order to explore a large design space in terms of speed-area trade-off, our Montgomery multiplier architecture uses variable pipeline stages and variable serial replications. Based on the multiplier architecture, we design a modular exponentiation circuit. Modular exponentiation is the main operation behind RSA encryption/decryption. We compare our designs to existing architectures and discuss their scalability. Our best multiplier is 13 times faster than the optimised software running on one core of a Core 2 Duo at 2.8 GHz, and is 4 times faster than the best hardware implementation in the literature. (Chapter 3)

Rapid optimisation of parametric designs

We generalise the idea used to scale our Montgomery multiplier. We present a coarse-grained method to automatically map a general class of algorithms consisting of a loop with loop dependencies carried from one iteration to the next to a parametric hardware design with pipelining and replication features. A technology-independent parametric model of the proposed design is developed to capture the variations of area and throughput with the number of pipeline stages and replications. Our model allows rapid optimisation of design parameters by a few pre-synthesis operations. We present an optimisation method based on the model. We show that our method facilitates design space exploration by quickly predicting the area taken by

the design as well as its maximum frequency and throughput. It is up to 96 times faster than a full search through the design space. (Chapter 4)

Novel architectures for primality testing

We present the first hardware architecture for the NIST approved Lucas primality test and an architecture for the Miller-Rabin primality test. Our architectures are based on our Montgomery multiplier and Montgomery exponentiator. We introduce our Miller-Rabin architecture briefly before focusing on the Lucas primality test. We present a hardware architecture to compute the Jacobi symbol based on the binary Jacobi algorithm. We analyse the dependencies in the Lucas sequences computation and propose a schedule for the different operations. Finally we compare the performance of our Miller-Rabin and Lucas architectures with an optimised software implementation in terms of speed, area and energy efficiency. Our Lucas architectures are implemented on a Xilinx Virtex-5 FPGA, and also synthesized for TSMC 65 nm and 45 nm ASICs. The FPGA implementation is 5 times more energy efficient than the optimised software running on an Intel Xeon at 2.53 GHz. The 65 nm ASIC implementation is up to 39 times more energy efficient than the FPGA implementation. The 45 nm ASIC implementation is 420 times more energy efficient than the optimised software implementation. (Chapter 5)

Power attack prevention and detection using on-chip monitoring

We present a novel approach involving on-chip power monitoring to prevent power attacks on reconfigurable hardware. Our power monitor circuit is based on a network of evenly distributed ring-oscillators. Two power attack countermeasures are derived from this circuit. They both work under the assumption that an attacker uses a shunt resistor-based power measurement circuit to perform the attack. The first countermeasure is a general framework based on a closed-loop control system that keeps the power consumption of any FPGA implementation constant. We analyse this new idea and show that using the framework as a hiding countermeasure is not directly feasible on current commercial FPGAs. However it could be implemented as a hard IP block on an FPGA dedicated to cryptography. The second countermeasure involves detecting the insertion of a shunt resistor-based power measurement circuit onto a device's

power rail. To our knowledge, this is the first attempt at detecting power attacks targeting reconfigurable hardware. This countermeasure is lightweight and has a relatively low power overhead. Shunt resistors as low as 2 Ω are detected with a maximum false-positive rate of 0.4% and no false-negatives. (Chapter 6)

Table 1.1: Summary of our research contributions

Research area	Contributions
Parametric designs for public-key encryption (Chapter 3)	• New parametric Montgomery multiplier architecture using variable pipeline stages and variable serial replications. • Montgomery exponentiation circuit based on our multiplier architecture with interleaved exponentiation support. • Speed/area comparisons with existing architectures and scalability results.
Rapid optimisation of parametric designs (Chapter 4)	• Coarse-grained method for mapping algorithms containing one loop with loop dependencies carried from one iteration to the next to a parametric design using pipelining and replication. • Model of the design that allows predicting the throughput and area variations with the number of pipeline stages and replications. • General optimisation process integrating the model. • Analysis of the accuracy and prediction capabilities of the optimisation method.
Primality testing architectures (Chapter 5)	• The first hardware architecture for the NIST approved Lucas probabilistic primality test. • Hardware architecture for computing the Jacobi symbol. • Dependence analysis and scheduling of the Lucas sequence computation. • Comparison of our architectures on FPGA and ASIC with an optimised software in terms of speed, area and energy efficiency.
Power attack prevention and detection (Chapter 6)	• On-chip power monitor circuit based on a network of ring oscillators. • General framework based on a closed-loop control system that keeps the power consumption of any FPGA implementation constant. • Attack detection framework to detect the insertion of a shunt resistor-based power measurement circuit onto a device's power rail. • Implementation and in-depth evaluation of our constant power and attack detection frameworks.

1.3 Thesis organisation

This thesis is organised as follows. Chapter 2 introduces the general background relevant to our work. Chapter 3 presents a parametric Montgomery multiplier based on variable pipelining and serial replication and its application to modular exponentiation. Chapter 4 generalises the idea of the previous chapter by presenting a coarse-grained method to map loops with loop dependencies carried from one iteration to the next to a parametric hardware design. Chapter 5 presents novel architectures for probabilistic primality testing based on our Montgomery multiplication and exponentiation designs. Chapter 6 describes new power attack countermeasures based on on-chip power monitoring using a network of ring oscillators. Finally chapter 7 summarises the contributions of this thesis and presents ideas for improvement and future work. The organisation of the thesis together with the research topics explored in each chapter are outlined in Fig. 1.1.

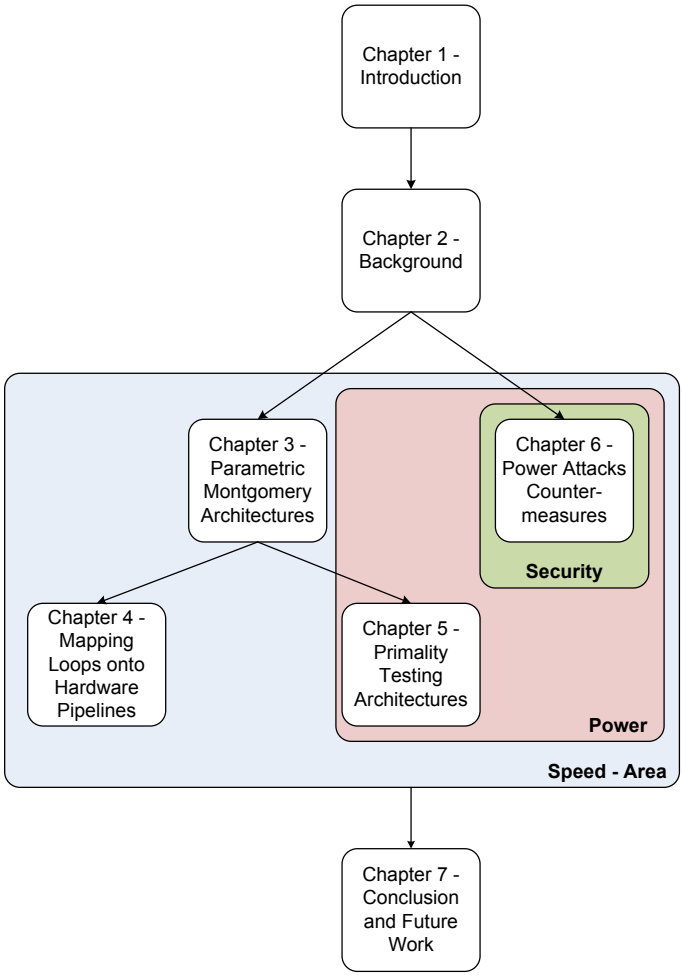


Figure 1.1: Organisation of this thesis

1.4 Publications

Journal Papers

1. Adrien Le Masle and Wayne Luk. Mapping Loop Structures onto Parametrised Hardware Pipelines. Submitted to *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (under review).

2. Adrien Le Masle and Wayne Luk. Power Attack Countermeasures Involving On-Chip Power Monitoring. Submitted to *IEEE Transactions on Computers* (under review).

Conference Papers

1. Adrien Le Masle, Wayne Luk, Jared Eldredge, and Kris Carver. Parametric Encryption Hardware Design. In *ARC '10: Proceedings of the 6th International Symposium on Applied Reconfigurable Computing*, 2010.

2. Adrien Le Masle and Wayne Luk. Design Space Exploration of Parametric Pipelined Designs. In *ASAP '10: Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2010.

3. Adrien Le Masle, Wayne Luk, and Csaba Andras Moritz. Parametrized Hardware Architectures for the Lucas Primality Test. In *SAMOS XI: Proceedings of the 11th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2011.

4. Adrien Le Masle, Gary C. T. Chow, and Wayne Luk. Constant Power Reconfigurable Computing. In *FPT 2011: 2011 International Conference on Field-Programmable Technology*.

5. Adrien Le Masle and Wayne Luk. Detecting Power Attacks on Reconfigurable Hardware. In *FPL 2012: 2012 International Conference on Field Programmable Logic and Applications*.

Chapter 2

Background

This chapter presents the general background relevant to our work. We first introduce Field Programmable Gate Arrays (FPGAs). Then we discuss the two main types of encryption schemes, namely symmetric-key and public-key cryptography. Finally, we present side-channel attacks and power attacks. Additional background material is given in each main chapter in order to facilitate their understanding.

2.1 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Arrays (FPGA) is a reconfigurable hardware device that can be programmed with any digital design after manufacturing. In this section we present the FPGA structure, the typical FPGA design flow and describe some FPGA applications. Extensive surveys of reconfigurable computing architectures and design methods are available in [Cha08, TCW+05].

2.1.1 Device structure

The three principal components of an FPGA are logic, interconnects and I/O blocks. In island-style FPGAs, logic, interconnects and I/O blocks are organised in a two-dimensional array as shown in Fig. 2.1. The logic blocks connect to the general programmable interconnects via SRAM-based programmable switch boxes (not shown in Fig. 2.1).

Each logic block contains a cluster of N logic elements. A simplified logic element contains

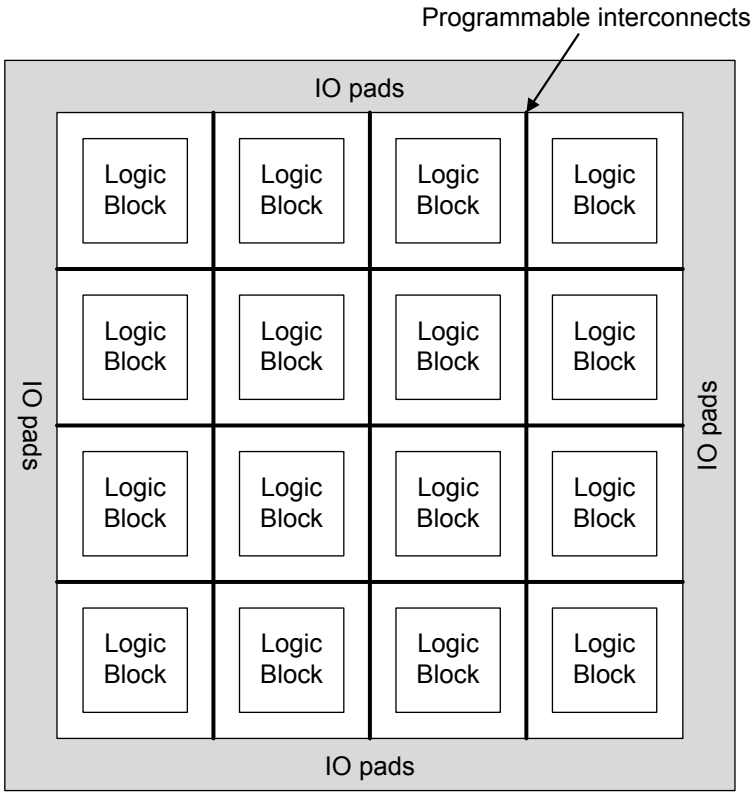


Figure 2.1: Island-style architecture

a lookup table (LUT), a Flip-Flop (FF) and a multiplexer (MUX) as depicted in Fig. 2.2. The output of the LUT can be registered into the FF. The MUX selects either the output of the LUT or the output of the flip-flop.

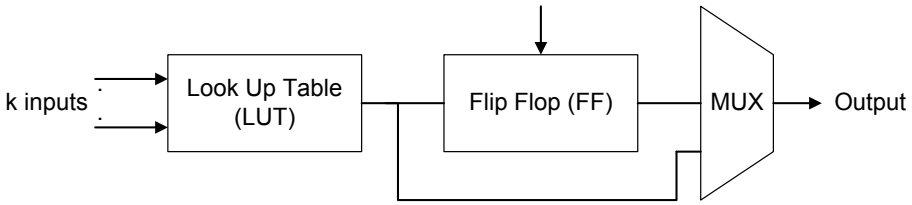


Figure 2.2: Simple logic element

A k-input LUT (k-LUT) can implement any logical function of k inputs by configuring 2^k SRAM cells with the truth table of the function. An example of a 3-LUT performing a 3-input logical XOR is shown in Fig. 2.3. Complex functions can be implemented by combining LUTs together. Most modern FPGAs are based on 4-input or 6-input LUTs.

A logic block and a logic element are respectively called Configurable Logic Block (CLB) and Slice in Xilinx FPGA families [Xil11b] and Logic Array Block (LAB) and Adaptative Logic Module (ALM) in Altera FPGA families [Alt12]. In modern FPGAs logic elements contain

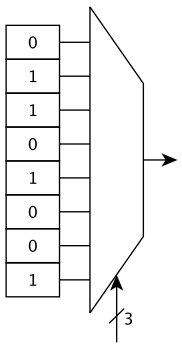


Figure 2.3: 3-LUT configured as a 3-input XOR

several LUTs, FFs and MUXes together with extra logic. The extra logic is often used for fast carry-chain addition or shift registers.

The logic element consisting of LUTs, FFs and extra logic is a fine-grained computation block. Most FPGAs contain a mix of such fine-grained blocks together with coarse-grained blocks such as Digital Signal Processors (DSPs) and memory elements. DSPs speed up operations such as additions and multiplications whereas embedded memory elements offer fast and low area-consuming storage.

As an example a Xilinx Spartan-6 contains two slices per CLB. Each slice contains four 6-LUTs, eight FFs and eight MUXes. In addition, one of the CLB slices contains extra logic for fast carry generation. In about half of the device's CLBs, this slice also contains distributed RAM and shift registers [Xil10a]. The Spartan-6 also contains several DSP blocks. The XC6SLX150T, which is the largest FPGA in the Spartan-6 family has 23 038 slices and 180 DSP blocks. Most recent FPGAs such as the Virtex-7 can reach more than 300 000 slices and 3000 DSP blocks [Xil12d].

2.1.2 Tools

The standard FPGA design flow is shown in Fig. 2.4. After specification, the design is programmed in a Hardware Description Language (HDL). The most popular HDLs are Verilog and VHDL, which work at the Register-Transfer Level (RTL). Some approaches involving high level programming languages targeting hardware also exist. We can cite for example Handel-C [Gra10], Vivado High-Level Synthesis [Xil12a] and MaxCompiler [Tec11]. These high level

languages often generate RTL code. They usually speed up design time but are less flexible than standard HDLs. One way of verifying HDL designs is by behavioural testing. Testbenches are written in order to verify each component of the design. Then the design is synthesized. This step takes the RTL design and turns it into a logic gate netlist. After synthesis we obtain useful estimates of the area consumption and the speed of the design. The logic gate netlist is tested again usually using the same behavioural testbenches. The mapping process maps the logic gate netlist to FPGA primitives such as LUTs, FFs, RAMs and DSPs. If not enough resources are available on the device, the mapping fails and the user has to modify the RTL design. The place-and-route process takes user constraints, places the primitives and routes the design. User constraints are typically timing constraints and optional placement and routing constraints. The placer and router try to match these constraints and output a timing report. If the timing is not met, the user can choose to relax the constraints or to modify the HDL design. Post-place and route tests can be performed at that stage using accurate timing information generated by the place-and-route process. Then the bitstream for the configuration SRAMs of the FPGA is generated. Finally the FPGA is programmed with the bitstream of the design which can be tested on chip. A common FPGA design cycle iterates through all these steps until a functional implementation meeting timing and area requirements is found.

2.1.3 Applications

The structure of an FPGA makes it naturally suited to parallel and pipelined applications. Many FPGA applications exist in a wide variety of areas such as networking [SP03], video processing [LAD+98], high-performance computing [HVG+07], software-defined radio [BLC09], industrial control systems [Liu11] and cryptography [BP99]. FPGAs can also be used as rapid-prototyping platforms for application-specific integrated circuits (ASICs) and microprocessors. ASICs are faster and consume less power than FPGAs. However, FPGAs have many advantages that can influence a designer to choose an FPGA over an ASIC implementation. In particular the time-to-market of a project using an FPGA is much shorter than the same project using an ASIC. As a matter of fact, unlike ASICs FPGAs are standard devices which have already been fabricated, packaged and tested by the vendor. FPGAs can also be less expensive than

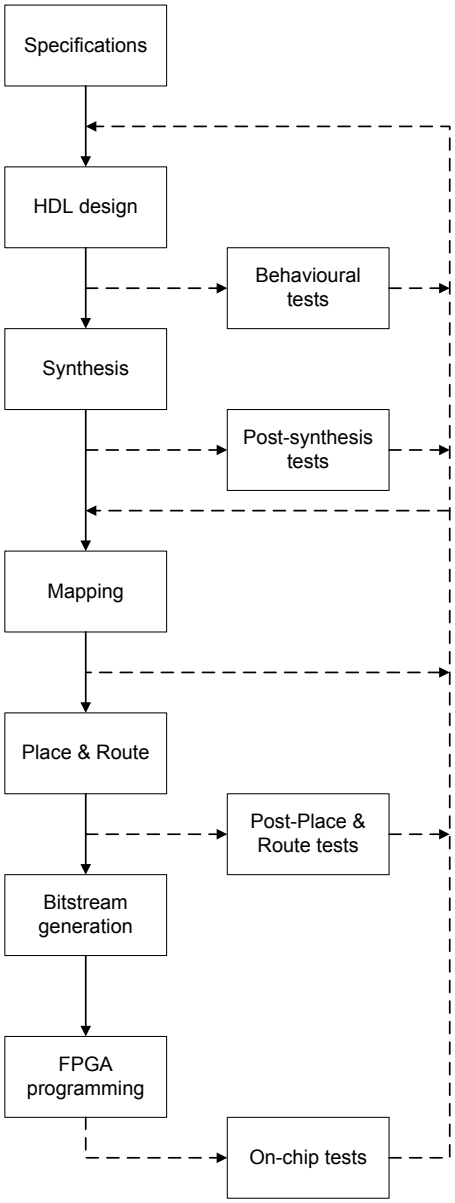


Figure 2.4: FPGA design flow

ASICs, especially in lower volumes since no user-specific fabrication is involved.

FPGAs are suitable platforms for implementing cryptographic algorithms. First, the structure of FPGAs makes them particularly suitable for pipelined applications, which is the case for most of the basic cryptographic operations. Second, FPGAs can be used to embed security into low power environments keeping very good performance. Finally, a pure hardware implementation of a cryptographic algorithm is usually less vulnerable than its software counterparts which are usually run in a multi-tasking operating system.

2.2 Symmetric-key cryptography

In this section we give an overview of symmetric-key cryptography and we present the two most common symmetric-key encryption algorithms.

2.2.1 Principles

Generalities

Symmetric-key cryptography was the only encryption scheme until the invention of public-key cryptography in the 70s. It usually has five main components [Sta11, p. 57]:

- The plaintext P which is the non-encrypted message.
- The encryption algorithm E used to encrypt the plaintext by applying a set of substitutions and transformations.
- The secret key K used both for encryption and decryption. The operations performed by the algorithm depend on the key.
- The ciphertext C which is the encrypted message produced by the encryption algorithm.
- The decryption algorithm D used to decrypt the ciphertext using the secret key.

The main feature of symmetric-key cryptography is the use of the same key K for encryption and decryption. The key has to be shared between the sender and the receiver in a secure way and must be kept safe. The encryption algorithm is often required to be designed so that an attacker having access to a number of ciphertexts together with the corresponding plaintexts cannot decrypt any other ciphertext and/or recover the key. Note that the security of a well-designed cryptographic system should not depend on the secrecy of the algorithm, but only on the secrecy of the key. In other words a cryptosystem should be secure even if everything about the system, except the key, is publicly available. This principle, known as Kerckhoffs's principle, was stated by Auguste Kerckhoffs in 1883 [Ker83].

The main steps of public-key encryption are the following:

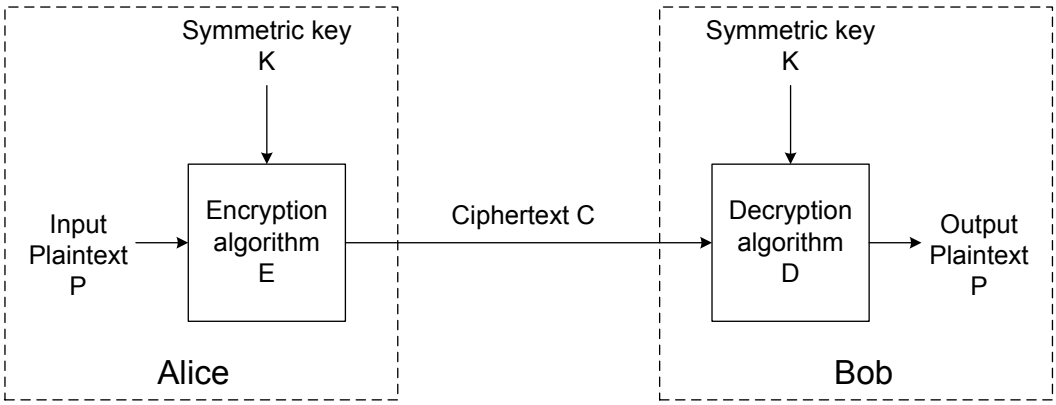


Figure 2.5: Symmetric key encryption

1. A secret key K is generated and shared between Alice and Bob over a secure channel. This can be done using public-key encryption, which is presented in section 2.3.

2. If Alice wants to send a confidential message P to Bob, Alice encrypts the message using the secret key K. The ciphertext is therefore:

$$C = E(K, P) \tag{2.1}$$

3. After receiving the message, Bob can decrypt it using the same key K as follows:

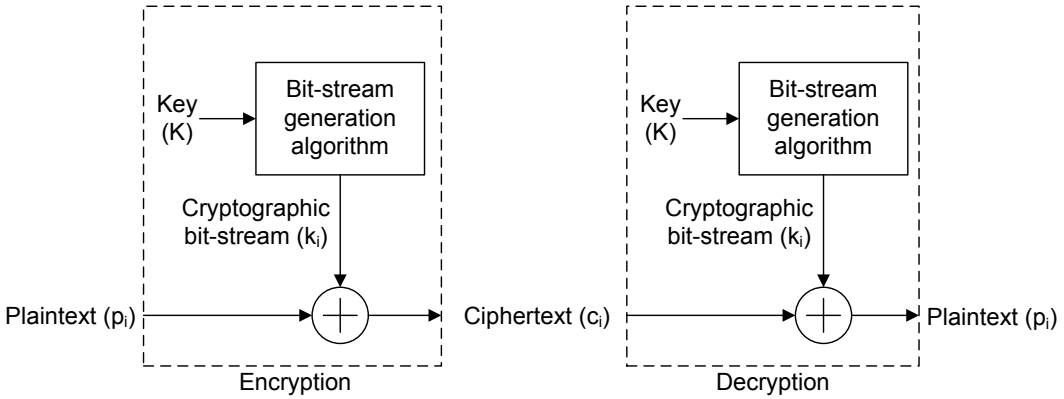
$$P = D(K, C) \tag{2.2}$$

The encryption process is summarised in Fig. 2.5.

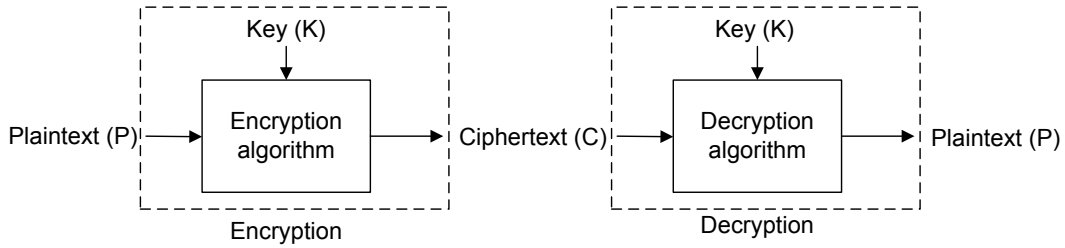
Block Cipher and Stream Cipher

A block cipher is an encryption/decryption algorithm which processes a fixed-size plaintext block and produces a ciphertext block of the same size. Longer plaintexts/ciphertexts can be processed using several modes of operation (electronic codebook, cipher block chaining, cipher feedback mode, output feedback mode or counter mode, etc) approved by the National Institute of Standards and Technology (NIST) [NIS].

A stream cipher encrypts a data stream one bit or one byte at a time, usually combining the data stream with a cryptographic bit-stream. The cryptographic bit-stream is generated



(a) Stream cipher



(b) Block cipher

Figure 2.6: Stream and block ciphers (adapted from [Sta11, p. 93])

by a bit-stream generation algorithm using the secret key.

The differences between a block and a stream cipher are shown in Fig. 2.6. Widely used symmetric encryption algorithms such as DES and AES are all block ciphers.

2.2.2 Cryptanalysis and brute-force attacks

Cryptanalytic attacks exploit knowledge about the cryptographic algorithm and optionally some plaintexts or plaintext-ciphertext pairs, in order to recover a given plaintext or the secret key. Several types of cryptanalytic attacks exist according to how much information the attacker can obtain from the cryptographic system. The five main types of attacks are shown in Table 2.1.

In every case, we suppose that the attacker knows the encryption algorithm and has access to the ciphertext, which constitutes the ciphertext-only attack type. In a known-plaintext scenario, the attacker is able to capture some plaintext messages and their corresponding ciphertexts. A chosen-plaintext attack is possible if the attacker can encrypt any plaintext using the crypto-system. The chosen-ciphertext attack is the symmetric of the chosen-plaintext attack.

Table 2.1: Types of attacks (adapted from [Sta11, p. 60])

Attack type	Additional information available to attacker (encryption algorithm and ciphertext known by default)
Ciphertext-only	• None
Known-plaintext	• Some plaintext-ciphertext pairs
Chosen-plaintext	• Plaintext chosen by attacker and access to corresponding ciphertext encrypted with secret key
Chosen-ciphertext	• Ciphertext chosen by attacker and access to corresponding plaintext decrypted with secret key
Chosen-text	• Plaintext chosen by attacker and access to corresponding ciphertext encrypted with secret key • Ciphertext chosen by attacker and access to corresponding plaintext decrypted with secret key

The chosen-text attack assumes that the attacker can encrypt/decrypt any plaintext/ciphertext with the crypto-system and obtain the corresponding ciphertext/plaintext. Encryption algorithms are usually designed to withstand at least known-plaintext attacks.

Brute-force attacks enumerate every possible key until an intelligible plaintext is recovered from the ciphertext. For a key of size n, there are 2^n possible keys. If n is large enough, such an attack is not computationally feasible in practice.

Two common definitions try to quantify the security of an encryption scheme [Sta11, p. 61]. We say that an encryption scheme is unconditionally secure if not enough information is present in the ciphertext to determine the plaintext, no matter how much ciphertext can be obtained. An unconditionally secure encryption algorithm must use keys with the same security requirements as the one-time pad. For each message, the one-time pad uses a new random key of the same length as the message for encryption and decryption. This scheme produces random ciphertexts with no statistical relationship to the corresponding plaintexts and is therefore unbreakable. However it is limited by the fact that creating a large quantity of truly random keys as well as distributing and protecting all these keys is impractical. Hence

none of the widely used encryption algorithms is unconditionally secure. An encryption scheme is computationally secure if the cost of breaking the cipher is higher than the value of the encrypted information, or if the time required to break the cipher is higher than the time the information needs to be kept secret. It can be hard to estimate whether a crypto-system is computationally secure. NIST gives recommended key sizes and lifetimes for their approved encryption ciphers [NIS12] in order to guide the design of a crypto-system.

2.2.3 Data Encryption Standard (DES)

Algorithm

The Data Encryption Standard (DES) is one of the best known symmetric-key encryption schemes. It was adopted in 1977 by the NIST [NIS99]. DES is a block cipher encrypting 64-bit blocks using a 56-bit key[1]. DES is based on a Feistel cipher structure that alternates between substitutions and permutations. The Feistel cipher structure is shown in Fig. 2.7. The data are divided into a right and a left half. Round i takes the left part L_{i-1}, the right part of the data R_{i-1} and K_i as inputs. K_i is derived from the key K through a subkey generation algorithm. The output data are computed as:

$$L_i = R_{i-1} \tag{2.3}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \tag{2.4}$$

A substitution is performed on the left half L_{i-1} of the data by taking the exclusive or of L_{i-1} with the round function f applied to the right half R_{i-1} and the round key K_i. Then the two halves of the data are permuted. As shown in Fig. 2.7, the encryption and decryption processes are very similar. The ciphertext is decrypted by swapping its two halves and performing the same round operations using the round keys in reverse order. The output of these operations give the plaintext (with its left and right halves swapped).

The DES encryption algorithm is shown in Fig. 2.8. DES has 16 rounds following the Feistel structure with an extra permutation at the beginning and at the end. In the key schedule, the

[1]The key is actually 64-bit long but only 56 bits are used.

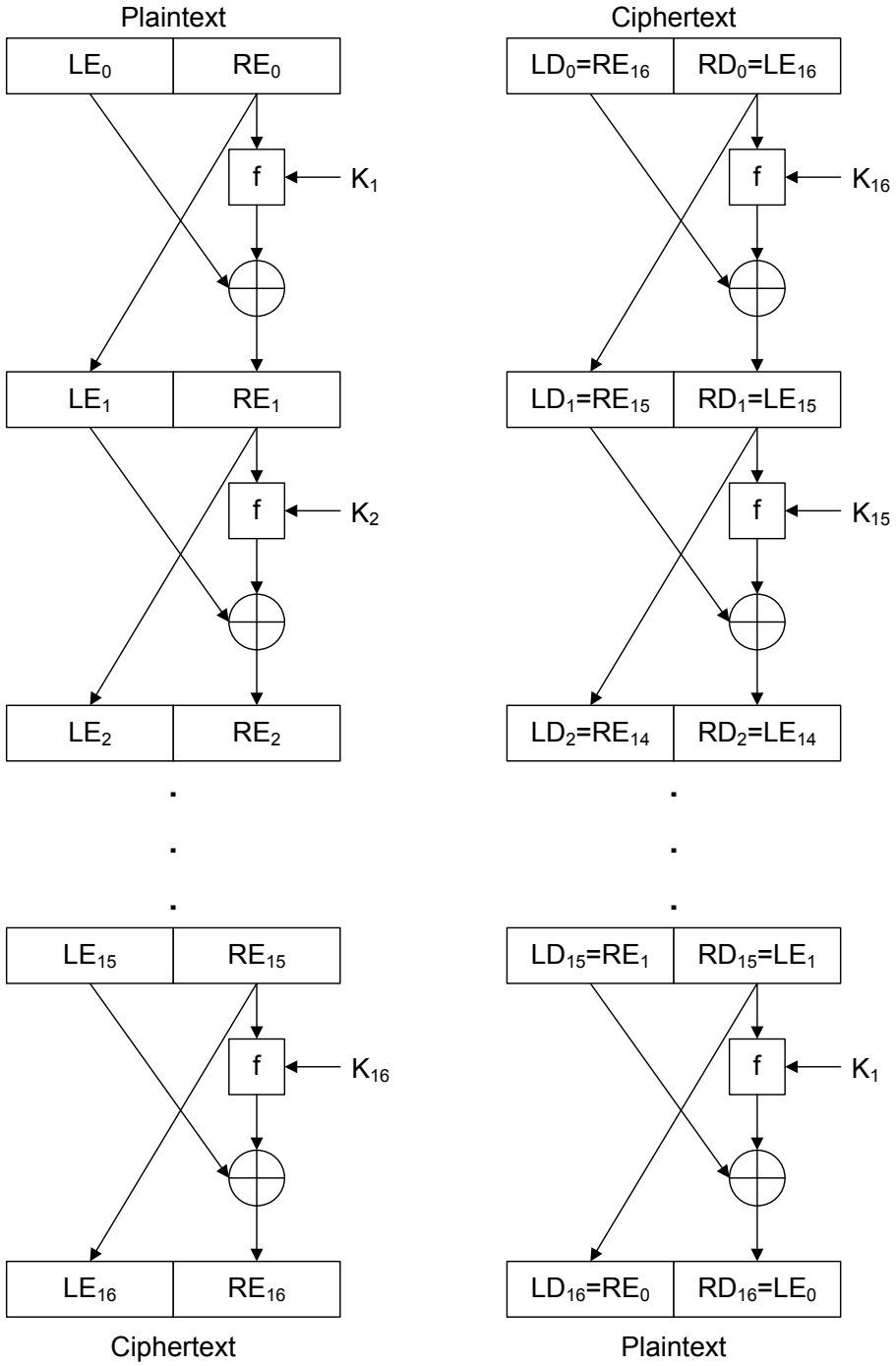


Figure 2.7: 16-round Feistel encryption and decryption

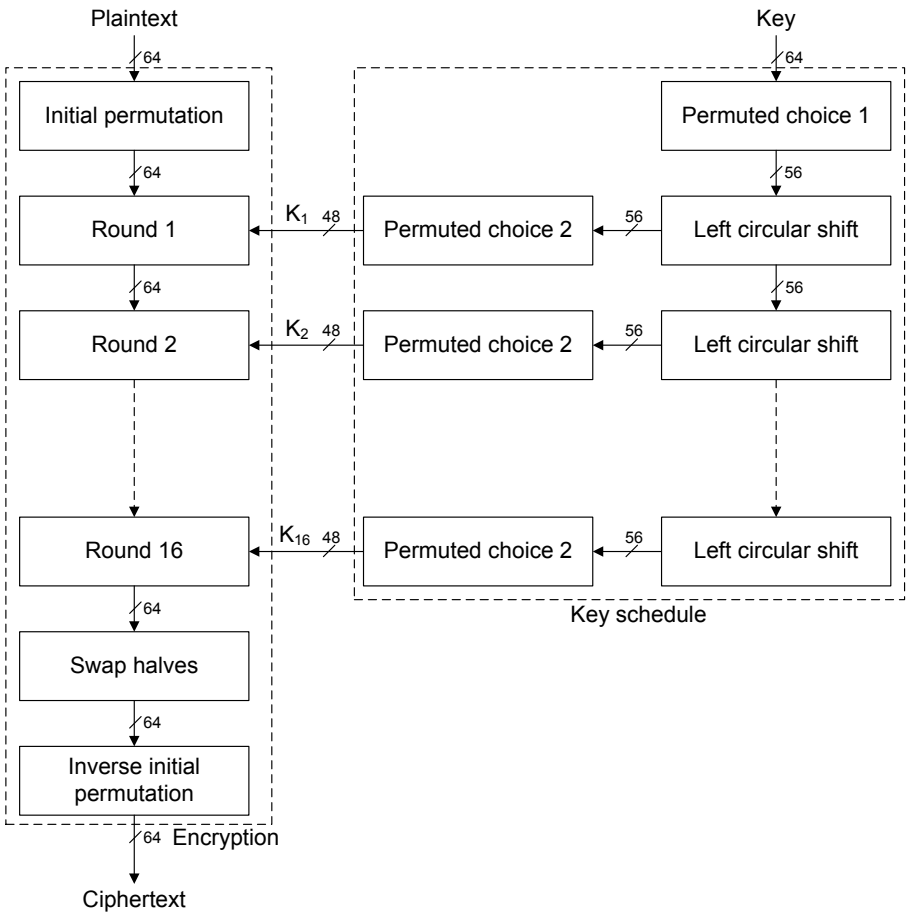


Figure 2.8: DES encryption algorithm

64-bit key is initially truncated and permuted (Permuted choice 1) to obtain a 56-bit key. In each round the key is rotated using a predefined shift schedule. The shifted key is truncated to 48-bit and permuted again (Permuted choice 2) to form the round key K_i.

The Feistel function used for DES encryption/decryption is shown in Fig. 2.9. The 32-bit right half R is expanded to 48 bits using an expansion permutation E and combined with the 48-bit round key K using exclusive or. The 48-bit result is dispatched to a series of S-boxes $S1$ to $S8$. Each S-box performs a substitution on a 6-bit input and returns a 4-bit output. Finally the 32-bit output of the set of S-box is permuted using a permutation function P. The decryption algorithm is similar to the encryption algorithm with the key schedule performed in reverse order.

The operations of each permutations box, expansion box and S-box are defined precisely by the standard in [NIS99].

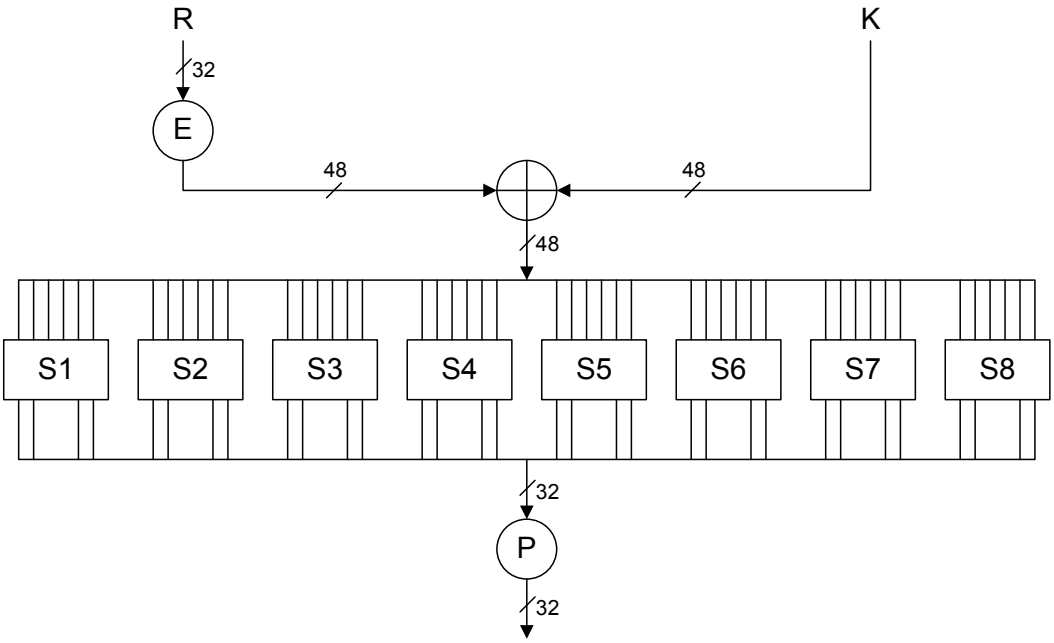


Figure 2.9: Calculation of $f(R, K)$

Security

The relatively short key size of DES makes it vulnerable to brute-force attacks. In 1998, a custom machine that is able to crack DES in a few days for less than $250,000 was built by the Electronic Frontier Foundation [Fou98]. Since then much cheaper machines have been built such as the Copacobana machine made of 120 low-cost FPGAs that can break DES in less than nine days on average for $10,000 [KPP+06]. DES is still approved by NIST in the triple-DES mode, which uses a larger 112-bit key [NIS12]. However, in new applications DES is superseded by the Advanced Encryption Standard (AES).

2.2.4 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) was published by NIST in 2001 as a symmetric block cipher to replace DES [NIS01]. AES is not a Feistel cipher. In AES, all operations are performed on byte-wide data represented as polynomials over $GF(2^8)$ with the irreducible polynomial $(x^8 + x^4 + x^3 + x + 1)$. The plaintext block size is 128 bits and the key can be 128, 192 or 256 bits. The cipher consists of N rounds, depending on the key size: 10 rounds for a 128-bit key, 12 rounds for a 196-bit key and 14 rounds for a 256-bit key. A key expansion mechanism creates a round key for each of the N rounds. The round keys are always 128-bit

long for any of the three secret key sizes. In the first (N-1) rounds, four transformations are performed in the following order: SubBytes, ShiftRows, MixColumns, AddRoundKey.

An initial AddRoundKey transformation is performed before the first round. The last round consists only of three operations (SubBytes, ShiftRows, AddRoundKey). During the encryption/decryption process, the intermediate data (state) and the round keys are represented as 4x4 matrices of bytes. The state is initialised column by column with the 16-byte plaintext. The SubBytes, ShiftRows and MixColumns operations only take the state as inputs, whereas the AddRoundKey operation uses both the state and the round key. The final state is returned as the ciphertext. Each column of the state matrix is considered as a polynomial of degree 4 whose coefficients are finite field elements in $GF(2^8)$. The AES encryption process is shown in Fig. 2.10 and the encryption algorithm is shown in Algorithm 2.1.

In the following, we give an overview of each of the four AES transformations. These transformations as well as the key expansion mechanism are explained in more detail in [NIS01, Sta11].

Algorithm 2.1: AES encryption algorithm

Input: *plaintext*, *roundkey*, number of rounds $ROUNDS$
Output: *ciphertext*

```
1 state = plaintext
2 state = AddRoundKey(state, roundkey[0])
3 for round = 1 to ROUNDS do
4     state = SubBytes(state)
5     state = ShiftRows(state)
6     if round < ROUNDS then state = MixColumns(state)
      state = AddRoundKey(state, roundkey[round])
7 end
8 ciphertext = state
```

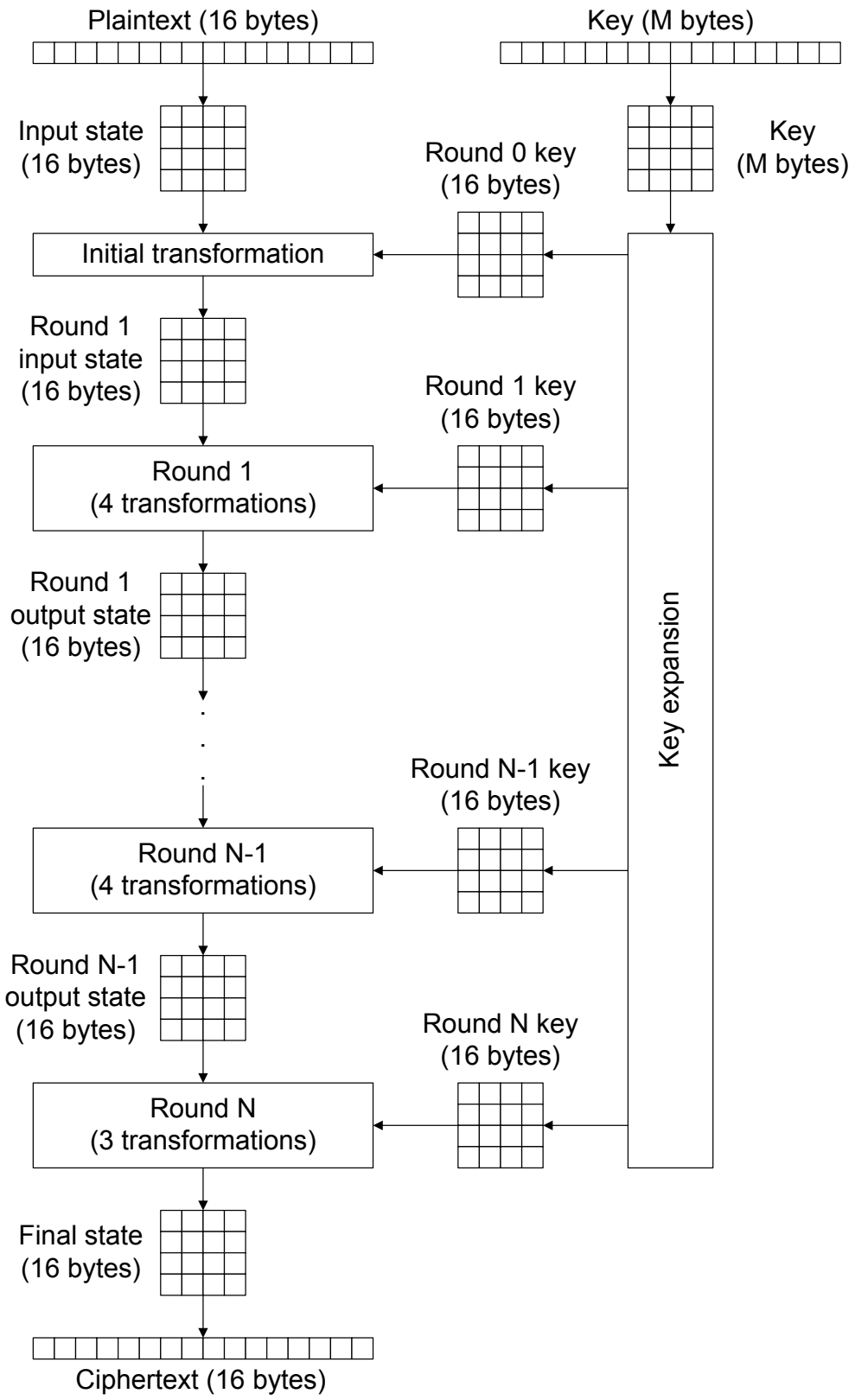


Figure 2.10: AES encryption algorithm (adapted from [Sta11, p. 175])

SubBytes

The SubBytes transformation changes each byte of the state with a corresponding byte from an SBOX as follows [DH10]:

$$New\_State[Row, Col] = SBOX[X, Y] \tag{2.5}$$

$$X = Old\_State[Row, Col] \ div \ 16 \tag{2.6}$$

$$Y = Old\_State[Row, Col] \ mod \ 16 \tag{2.7}$$

$SBOX$ is a 16x16 byte array which is indexed by the hexadecimal digits $(0..F, 0..F)$ as:

$$SBOX[X, Y] = Affine Transformation(\{XY\}^{-1}) \tag{2.8}$$

$\{XY\}^{-1}$ is the multiplicative inverse of byte $\{XY\}$ in $GF(2^8)$. *Affine Transformation* performs a matrix multiplication followed by a vector addition and is described in detail in [Sta11, pp. 180–185].

ShiftRows

ShiftRows cyclically rotates left the last 3 rows of the state matrix by 1, 2 and 3 bytes respectively as follows:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \xrightarrow{ShiftRows} \begin{pmatrix} a & b & c & d \\ f & g & h & e \\ k & l & i & j \\ p & m & n & o \end{pmatrix} \tag{2.9}$$

MixColums

The MixColums transformation multiplies each column, considered as a word-polynomial, by $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \ mod \ (x^4 + 1)$. This is equivalent to performing the

following transformation on the 4 bytes of each column:

$$\begin{pmatrix} m \\ n \\ p \\ q \end{pmatrix} \xrightarrow{MixColumn} \begin{pmatrix} \{02\} \bullet m \oplus \{03\} \bullet n \oplus p \oplus q \\ m \oplus \{02\} \bullet n \oplus \{03\} \bullet p \oplus q \\ m \oplus n \oplus \{02\} \bullet p \oplus \{03\} \bullet q \\ \{03\} \bullet m \oplus n \oplus p \oplus \{02\} \bullet q \end{pmatrix} \tag{2.10}$$

where $\oplus$ and $\bullet$ represent respectively addition and multiplication in $GF(2^8)$.

AddRoundKey

In the AddRoundKey stage, the round key is simply combined with the state using an exclusive or operation. The expansion mechanism of the secret key to the different round keys is explained in [Sta11, pp. 190–193].

Decryption

The AES decryption performs the inverse of the four main transformations on the ciphertext, using the round keys in reverse order. InvShiftRows rotates the last 3 rows of the state right instead of left. InvSubBytes uses the inverse SBOX. An inverse SBOX is constructed by taking the inverse of $AffineTransformation$ and then the multiplicative inverse in $GF(2^8)$. InvMixColums multiplies the columns by $a(x)^{-1} = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}$. Finally AddRoundKey is its own inverse. The AES decryption algorithm is shown in Algorithm 2.2.

Algorithm 2.2: AES decryption algorithm

Input: *ciphertext*, *roundkey*, number of rounds $ROUNDS$
Output: *plaintext*

```
1 state = ciphertext
2 state = AddRoundKey(state, roundkey[ROUNDS])
3 for round = ROUNDS − 1 downto 0 do
4     state = InvShiftRows(state)
5     state = InvSubBytes(state)
6     state = AddRoundKey(state, roundkey[round])
7     if round > 0 then state = InvMixColumns(state)
8 end
9 plaintext = state
```

Security

Given the long key lengths for AES (128 bits, 192 bits or 256 bits), a brute-force attack is not computationally feasible as of today. A related-key attack (requiring ciphertext obtained by encrypting some plaintext under several keys related to each others) can break the full 12 rounds of 192-bit AES with time complexity of 2^{176} and data complexity of 2^{123}, and another attack can break the full 14 rounds of 256-bit AES with $2^{99.5}$ time and data complexity [BK09]. 256-bit AES is more vulnerable to related-key attacks than 192-bit AES due to weaknesses in the 256-bit AES key schedule. The first key-recovery attack better than brute-force was published in 2011 [BKR11]. It has a time complexity of $2^{126.1}$ for 128-bit AES, $2^{189.7}$ for 192-bit AES and $2^{254.4}$ for 256-bit AES. These improved cryptanalytic attacks are still computationally infeasible. However practical side-channel attacks such as timing attacks [Ber05,OST06] and power attacks [OGOP04] have been demonstrated against AES. Side-channel attacks are described in section 2.4.

2.3 Public-key cryptography

This section presents public-key cryptography and in particular the RSA algorithm, which is commonly used for key management and digital signature.

2.3.1 Principles

Public-key cryptography was publicly introduced by Diffie and Hellman in 1976 [DH76a,DH76b]. It has two main features that differentiate it from conventional cryptography [Sta11, pp. 291–299]:

- It is based on mathematical problems rather than on substitution and permutation ciphers.
- It is asymmetric which involves the use of two separate keys, as opposed to symmetric encryption which only uses one key.

A public-key crypto-system has six main components:

- The plaintext P which is the non-encrypted message.
- The encryption algorithm E used to encrypt the plaintext.
- The public key KU and the private key KR. The public key is made public and is used for encryption. The private key is kept secret and is used for decryption.
- The ciphertext C which is the encrypted message produced by the encryption algorithm.
- The decryption algorithm D used to decrypt the ciphertext.

The main steps of public-key encryption are the following:

1. Each user generates a public/private key pair for encryption and decryption.
2. Each user makes its public key accessible in a public register and keeps its private key secret. Therefore, each user maintains a public-key ring containing the public keys of the other users.
3. If Alice wants to send a confidential message P to Bob, Alice encrypts the message using Bob's public key. The ciphertext is therefore:

$$C = E(KU_b, P) \tag{2.11}$$

4. After receiving the message, Bob can decrypt it using its private key as follows:

$$P = D(KR_b, C) \tag{2.12}$$

The message can only be decrypted by Bob as he is the only one who knows the required private key. This encryption process is summarised in Fig. 2.11.

Public-key encryption can also be used to provide authentication. As a matter of fact, Alice can sign a message P she wants to send to Bob with her private key KR_a:

$$S = E(KR_a, P) \tag{2.13}$$

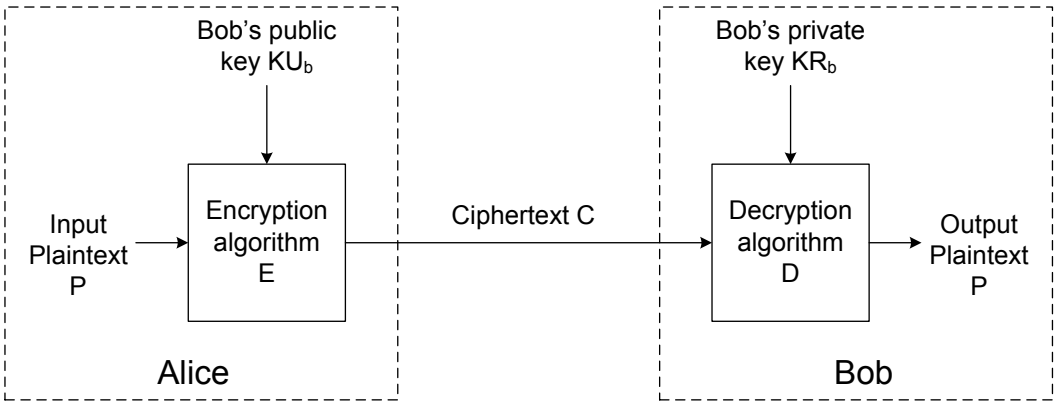


Figure 2.11: Public-key encryption

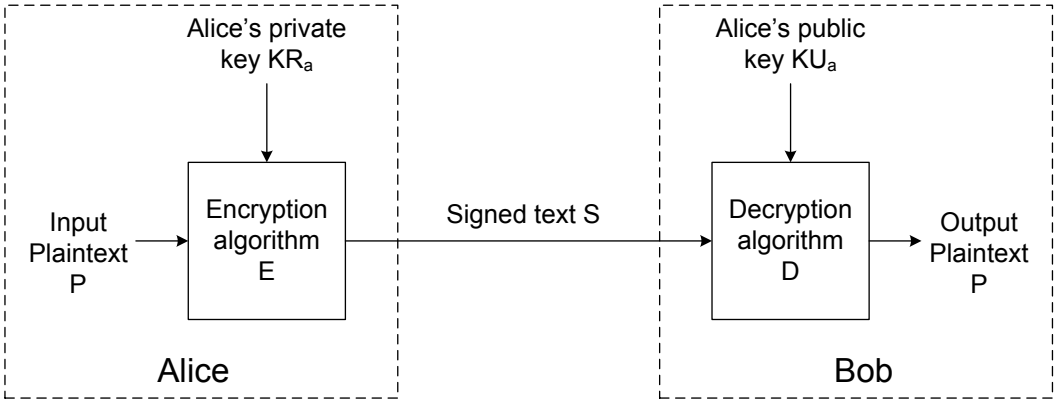


Figure 2.12: Public-key authentication

Then Bob can verify the signed message using Alice's public key:

$$P = D(KU_a, S) \tag{2.14}$$

As the message was encrypted using Alice's private key, only Alice could have prepared it. Moreover, nobody can alter the message without access to Alice's private key. Then this process provides authentication in terms of source and data integrity. It is depicted in Fig. 2.12.

2.3.2 Requirements

A public-key crypto-system must fulfil the following requirements [DH76b]:

1. It is computationally easy to generate a pair (public key KU_x, private key KR_x).

2. Knowing the public key and the message to be encrypted, it is computationally easy to generate the corresponding ciphertext.

3. Knowing the private key it is computationally easy to decrypt the ciphertext.

4. It is computationally infeasible, knowing the public key, to determine the private key.

5. It is computationally infeasible, knowing the public key and the ciphertext, to recover the original message.

Hence a public-key crypto-system relies on the discovery of a trap-door one-way function, that is a family of invertible function f_k such that:

$Y = f_k(X)$ is easy to compute, if k and X are known

$X = f_k^{-1}(Y)$ is easy to compute, if k and Y are known

$X = f_k^{-1}(Y)$ is infeasible to compute, if Y is known but k is not known

2.3.3 Key sizes

To prevent brute-force attacks, the keys used in public-key encryption have to be large. However, the public-key crypto-system relies on an invertible mathematical function that can be complex to compute. Hence, the key size must also be small enough to allow encryption and decryption to be performed in a reasonable amount of time. In practice, the key sizes needed to prevent brute-force attacks make public-key encryption too slow to be used for general encryption. The main applications of public-key cryptography are therefore key management and digital signature which only require encryption of short messages.

2.3.4 RSA

RSA (Rivest-Shamir-Adleman) [RSA78] is a public-key encryption algorithm whose strength relies on the difficulty of solving the following number-theoretic problems:

- Given $N = p.q$ with p and q two large primes, try to factorise N.

- Find P such that $P^E = C \bmod N$ given integers N, E, and C such that $N = p.q$ where p and q are two large primes, $1 < E < (p-1)(q-1)$ is coprime to $(p-1)(q-1)$, and $0 \leq C < N$.

The RSA algorithm consists of three parts: key generation, encryption and decryption.

RSA key generation

To compute the public key used for encryption and the private key used for decryption the following steps are performed [Sta11, pp. 302-303]:

1. Select two large prime numbers p and q such that $p \neq q$ and keep them secret.
2. Calculate $N = p.q$.
3. Calculate the Euler's totient function[2] $\phi(N) = (p-1)(q-1)$ and keep it secret.
4. Select integer E such that $gcd(\phi(N), E) = 1$ and $1 < E < \phi(N)$. We often choose a small value for E (common values are 3, 5, 17 or 65537 [FSK10]) to facilitate encryption.
5. Calculate D such that $E.D = 1 \mod \phi(N)$.

The public key is (E, N) and the private key is (D, N).

Encryption/Decryption

The ciphertext C corresponding to the encryption of the plaintext P with the public key (E, N) is:

$$C = P^E \mod N \tag{2.15}$$

To decrypt the ciphertext C, one has to own the private key (D, N). C can then be decrypted as follows:

$$P = C^D \mod N \tag{2.16}$$

As a matter of fact, Euler proved that:

$$P = P^{k.\phi(p.q)+1} \mod p.q \text{ for } k \in \mathbb{N} \tag{2.17}$$

[2]Number of numbers less than N relatively prime to N.

As E and D are chosen so that $E.D = k.\phi(p.q) + 1$ (step 5 of the RSA key generation), we have:

$$\begin{aligned} C^D \bmod N &= (P^E)^D \bmod N \\ &= P^{E.D} \bmod N \\ &= P^{k.\phi(p.q)+1} \bmod N = P \end{aligned}$$

Chinese Remainder Theorem (CRT)

The Chinese Remainder Theorem (CRT) is a useful tool to simplify RSA encryption/decryption. The CRT states that it is possible to reconstruct integers from their residues modulo a set of pairwise relatively prime numbers [Sta11, pp. 278–281]. More formally, let:

$$M = \sum_{i=1}^{k} m_i \tag{2.18}$$

where $gcd(m_i, m_j) = 1$ for $1 \leq i, j \leq k$ and $i \neq j$, that is the m_i are pairwise relatively prime. Any integer x in $\mathbb{Z}_M$ (ring of integers modulo M) can be represented by the tuple $(x_1, x_2, ..., x_k)$ where:

$$x_i = x \bmod m_i \tag{2.19}$$

Moreover, x can be reconstructed from the x_i as follows:

$$x = \left(\sum_{i=1}^{k} x_i c_i \right) \bmod M \tag{2.20}$$

$$c_i = M_i.(M_i^{-1} \bmod m_i) \tag{2.21}$$

$$M_i = M/m_i \tag{2.22}$$

Note that $(M_i^{-1} \bmod m_i)$ is well defined as M_i is relatively prime to m_i.

Applied to RSA, the CRT shows that in order to compute $P^E \bmod N$, we can compute $P^{E \bmod (p-1)} \bmod p$ and $P^{E \bmod (q-1)} \bmod q$ (the same applies to the decryption $C^D \bmod N$).

The reduction of the exponent comes from Fermat's theorem, which states that $a^{p-1} \bmod p = 1$ if p and a are relatively prime. If P, E and N are n-bit long, the computation of $P^E \bmod N$ requires about $3n/2$ multiplications modulo N, assuming that around half of the bits of E are 1 and that the square-and-multiply exponentiation algorithm is used. The computation of $P^{E \bmod (p-1)} \bmod p$ and $P^{E \bmod (q-1)} \bmod q$ requires $2.3n/4 = 3n/2$ multiplications modulo one of the primes of size $n/2$. This saves at least a factor of 2 in computing time.

Key size

As of today a key size of at least 2048 bits is recommended. More precisely in [Lab04, FSK10, NIS12], the use of a 2048 bit key is stated to be secure at least until 2030. For data that need to be protected beyond 2031 a 3072 bit key is recommended. Most of the experiments involving RSA encryption and key generation in this thesis are performed for 1024-bit inputs. This is equivalent to 2048-bit security using the CRT.

2.4 Side-channel attacks

A side-channel attack is an attack based on the information gained from the physical implementation of a cryptosystem. It differs from brute-force or theoretical weaknesses that are based on the analysis of the algorithm itself. A side-channel attack is implementation-dependent whereas conventional cryptanalytic attacks are algorithm-dependent. Side-channel attacks are classified in different categories according to the side-channel exploited.

Timing attacks are one of the most common side-channel attacks together with power attacks, which are studied in more detail in the following sections. Timing attacks are based on measuring the time various computations take to complete. All encryption algorithms perform operations which depend on the value of the key. A common example is modular exponentiation in a naive implementation of RSA. If the exponentiation algorithm uses the square-and-multiply algorithm as shown in Algorithm 3.4, only a squaring operation is performed if the key bit is 0 whereas a squaring followed by a multiplication are performed if the key bit is 1. Hence the execution time for modular exponentiation depends linearly on the number of bits at 1 in the key. Repeated executions with the same key and different inputs can be used to recover the key

completely using statistical analysis as shown in [Koc96]. Cache-timing attacks exploit timing variability or inter-process leakage of memory access patterns created by cache misses. They are particularly efficient on lookup-table based implementations of encryption algorithms and have been successfully demonstrated against AES [Ber05, OST06].

Electromagnetic (EM) attacks are based on the leaked electromagnetic radiations of a device. The analysis techniques for EM attacks are very similar to what is used in power analysis. EM attacks have been successfully demonstrated on implementations of DES, RSA and Elliptical Curve Cryptography among others [QS01, GMO01, DMOPV07].

Other less common side-channel attacks exist. For instance, differential fault analysis tries to discover secrets by introducing faults in the circuit whereas acoustic cryptanalysis exploits sound produced during computation.

The following sections focus on power attack.

2.4.1 FPGA power consumption

Complementary Metal-Oxide Semiconductor (CMOS)

The most popular logic style is Complementary Metal-Oxide Semiconductor (CMOS), which is used to implement logic cells in most FPGAs. CMOS technology is based on NMOS and PMOS transistors which are arranged in a complementary structure. The PMOS transistors are located between V_{DD} and the output of the cells and form a pull-up network. The NMOS transistors are located between the ground and the output of the cells and form a pull-down network. CMOS gates are built so that the pull-up and pull-down networks are never conducting at the same time. The diagram of a CMOS NAND gate is shown in Fig. 2.13. If at least one of the inputs A or B is low (GND), at least one transistor of the pull-up network is conducting and one transistor of the pull-down network is insulating. Hence the output *out* is high. If both A and B are high (V_{DD}), both transistors of the pull-up network are insulating and both transistors of the pull-down network are conducting. Hence the output *out* is low.

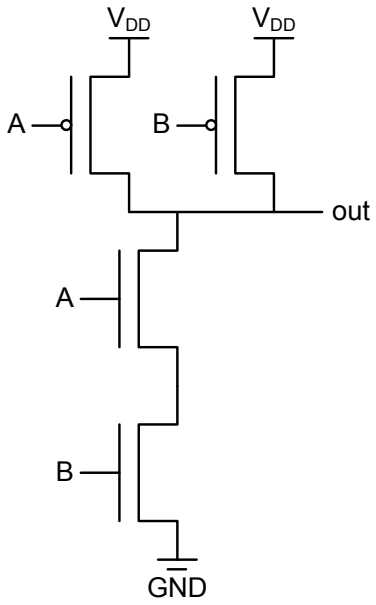


Figure 2.13: CMOS NAND gate

Static Power Consumption

The power consumption of an FPGA is the sum of the static and dynamic power consumption of all its transistors. The static power is caused by the leakage current I_{leak} flowing through the MOS transistors that are turned off. The static power consumption is therefore:

$$P_{stat} = I_{leak}.V_{DD} \tag{2.23}$$

Dynamic Power Consumption

The dynamic power consumption is mainly caused by the switching of the output signal of a logic cell. The dynamic power consumption is data-dependent. It is often the dominant factor in the total power consumption even if static power is catching up because of process shrinking. Two factors contribute to the dynamic power consumption: the output capacitance of the cell and the temporary short circuit that occurs during the switching of the output.

The output capacitance of a cell C_L depends on the physical properties of the process technology, the length of the wires to the following cells and the number of following cells (fanout). When a $0 \rightarrow 1$ output transition occurs, C_L charges which causes a current to flow from the power supply. It can be shown that the average charging power consumption

is [MOP07]:

$$P_{chrg} = \alpha_{0\to1}.f.C_L.V_{DD}^2 \tag{2.24}$$

f is the clock frequency and $\alpha_{0\to1}$ is the average number of $0 \to 1$ transitions occurring at the output of the cell per clock cycle.

Short-circuit current is due to the finite rise and fall times of PMOS and NMOS transistors. Hence for both $0 \to 1$ and $1 \to 0$ transitions, there is a period of time during which a current flows from V_{DD} to ground. The short-circuit power consumption can be approximated by:

$$P_{sc} = (\alpha_{0\to1} + \alpha_{1\to0}).f.V_{DD}.I_{peak}.t_{sc} \tag{2.25}$$

I_{peak} is the current peak caused by the short-circuit and t_{sc} is the time of the short circuit. To obtain this result, we make the simple assumption that the short-circuit current for one switching event is a triangle with base t_{sc} and height I_{leak}.

The dynamic power consumption P_d is the sum of the charging power consumption P_{chrg} and the short-circuit power consumption P_{sc}:

$$P_d = P_{chrg} + P_{sc} \tag{2.26}$$

2.4.2 Power analysis

Power analysis relies on the fact that the dynamic power consumption of a circuit depends on the switching activity of its transistors as shown in equations 2.24 and 2.25. Hence, by measuring the power consumed by a chip performing a given cryptographic operation, an attacker can recover information about the data being processed and the secret keys used.

Simple power analysis (SPA) proceeds by direct observation of a power trace. An implementation whose power consumption is different depending on which bit of the secret key is being processed is vulnerable to SPA. This is for instance the case for some implementations of the square-and-multiply modular exponentiation algorithm used in RSA or in the Diffie-Hellman key exchange protocol. The power trace of an unsecured implementation of this algorithm,

in which squaring and multiplication operations have significantly different power signatures, is presented in [Roh10]. In this example, the private key can be recovered easily with only a single power measurement. Scalar multiplication in elliptic curve cryptography (ECC) is also vulnerable to SPA. In this algorithm, either a point doubling or a point addition operation is performed, depending on the value of the key.

Differential power analysis (DPA) performs statistical analyses of multiple power traces for different inputs. This method is introduced in [KJJ99] where a DPA on a smartcard implementation of the DES algorithm is successfully performed. DPA is generally more powerful than SPA. The secret key is partitioned into smaller subkeys. Intermediate values or transitions depending on the analysed subkey and known inputs or outputs (message m) are mapped into estimated power consumptions (h_m) using a power model. The attacker usually has a limited knowledge of the device attacked and he cannot perform precise simulations at the analog, logic or behavioural level. Hence, the simple Hamming weight (HW) or Hamming distance (HD) models are usually chosen.

The Hamming weight model estimates the power consumption by counting the number of ones in the intermediate value of interest. This is the simplest model and is often applied if the attacker only knows one intermediate value and cannot have access to the preceding or succeeding value. This model is usually not very suited to CMOS circuits whose power consumption depends on the output transitions.

The Hamming distance model counts the number of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions that occur in a part of the circuit during a certain period of time. Usually the number of transitions is approximated by the number of bit flips in the intermediate value of interest. If the intermediate value changes from v_0 to v_1, the Hamming distance model computes $HD(v_0, v_1) = HW(v_0 \oplus v_1)$, where $HW(v)$ represents the Hamming weight of word v. In [SOQP04], it is shown that the Hamming distance model is very suited to estimate the power consumption of an FPGA device by predicting the number of bit transitions inside the device's registers.

Once the power model is chosen, the attacker uses statistical methods to estimate the most likely secret subkey from all the possible subkeys in the search space. The correlation power analysis attack (CPA) is often used to this end [KJJ99]. CPA uses the Pearson correlation as

the statistical distinguisher. Suppose that we have K key hypothesis, M different messages for which we have captured power traces of length T. $h_{m,i}$ is the estimated power consumption for message m and key hypothesis i and $p_{m,j}$ is the recorded value of the power trace for message m at time j. A KxT correlation matrix R is computed. Its elements $r_{i,j}$ are defined by [MOP07]:

$$r_{i,j} = \frac{\sum_{m=1}^{M}(h_{m,i} - \overline{h}_i).(p_{m,j} - \overline{p}_j)}{\sqrt{\sum_{m=1}^{M}(h_{m,i} - \overline{h}_i)^2 . \sum_{m=1}^{M}(p_{m,j} - \overline{p}_j)^2}} \tag{2.27}$$

$\overline{h}_i$ is the mean value of the vector $\mathbf{h}_i$ of estimated power consumptions for all messages under key hypothesis i. $\overline{p}_j$ is the mean value of the vector $\mathbf{p}_j$ of recorded power consumptions for all messages at time j. The most likely secret subkey is simply determined by identifying the row that contains the extreme value of matrix R.

2.4.3 Countermeasures

In designing a secured hardware-based crypto-system one needs to incorporate protections against SPA and DPA. In [MOP07], these countermeasures are divided into two groups: masking and hiding countermeasures.

Masking countermeasures randomize the intermediate values processed by the cryptographic device. The main goal is to make the power consumption independent of the intermediate values. They have been successfully applied to several encryption algorithms [RWS11, RM08] but usually lead to area and performance overheads. As a matter of fact, in order to get the correct result, that is the same result as the one obtained without masking, the output has to be corrected. This correction step can double the data-path of the implementation. For example, if we consider Boolean masking on AES, one data-path is required to process the plaintext and intermediate values masked by a random number, while another datapath processes the random number itself.

Hiding countermeasures aim at decoupling the data-specific power consumption from the processed intermediate values at the physical level. When the attacker does not have physical access to the device, filtering the power supply is a straightforward solution. Noise can be added to the power trace using on-chip noise generation [GM11]. The mapping, placement

and routing algorithms can also be made security-aware. In [TV04, YS07], wave dynamic differential logic (WDDL) and symmetrical routing are used to reduce the power consumption fluctuations. These techniques require specific placement and routing algorithms and can lead to area overhead of 3 times or more. Random dynamic voltage and frequency switching have also been proposed as hiding contermeasures [YWV+04, BZ07]. However, these techniques introduce a performance overhead and for current commercial FPGAs, voltage switching would need to be implemented off-chip, compromising the security of the system. No actual hardware implementation of these two solutions have been presented.

Both types of countermeasures focus on making an attack more difficult by decreasing the correlation between the power consumption and the actual computation. In practice no countermeasure alone can guarantee the security of a cryptographic system and several countermeasures should be used simultaneously.

2.5 Summary

In this chapter we first introduce Field Programmable Gate Arrays (FPGAs). Then we describe the two main types of encryption schemes, namely symmetric-key and public-key cryptography. The Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) are discussed as practical applications of symmetric-key cryptography. RSA, the public-key encryption algorithm commonly used for key management and digital signature, is also presented. Finally, we present side-channel attacks and power attacks in particular. For an unprotected implementation, power attacks are generally much more efficient at recovering cryptographic keys than cryptanalytic attacks.

Chapter 3

Parametric Montgomery Multiplier Architecture

In this chapter, we present new hardware architectures for Montgomery modular multiplication and exponentiation. After introducing Montgomery arithmetic, we focus on designing a parametric Montgomery multiplier that can explore a large design space in terms of speed-area trade-off. Our Montgomery multiplier architecture is applied to modular exponentiation, which is the main operation behind RSA encryption/decryption. Finally we compare our designs to existing architectures and discuss their scalability.

3.1 Motivation

In section 2.3, we show that most public-key algorithms consist of two main stages: the key generation stage which requires the ability to generate large prime numbers, and the encryption/decryption stage. Modular multiplication and modular exponentiation are core operations of several public-key crypto-systems such as the Diffie-Hellman key exchange protocol and RSA. They are also extensively used in probabilistic primality tests, which are studied in chapter 5.

As FPGAs are quickly increasing in size, it is becoming more of a challenge to fully utilize the design space available for a given budget. This introduces the need for parametric designs capable of exploring the entire design space, especially in terms of speed-area trade-offs.

The Montgomery algorithm is the most suited algorithm for hardware implementation of

modular multiplication due to its relatively simple structure. In this chapter, after introducing Montgomery arithmetic, we present new scalable and parametric hardware architectures for Montgomery modular multiplication and exponentiation. Chapter 4 will extend our architecture to a general class of algorithms. In chapter 5, we will use our modular multiplier and exponentiator designs to create scalable architectures for primality testing.

3.2 Introduction to Montgomery Arithmetic

This section presents additional background on Montgomery arithmetic which is relevant to the understanding of the chapter.

3.2.1 Montgomery multiplication algorithm

Montgomery multiplication is commonly used to speed up modular multiplication. In particular, by using the Montgomery algorithm we can greatly accelerate modular exponentiation. The Montgomery algorithm was first introduced by Peter Montgomery in 1985 in [Mon85].

Let us explain the principles behind the Montgomery algorithm by first describing Montgomery reduction. Let R and N be two integers with R greater than N and relatively prime to N. Since $gcd(R, N) = 1$, there exist R^{-1} and N' such that $0 < R^{-1} < N$ and $0 < N' < R$ and:

$$RR^{-1} - NN' = 1 \tag{3.1}$$

Note that this equation gives:

$$N' = (-N)^{-1} \bmod R \tag{3.2}$$

For an integer X, $X.R^{-1} \bmod N$ can be computed efficiently using Algorithm 3.1. This algorithm works as follows [Gua03, Mon85]. We have:

$$XR^{-1} = XRR^{-1}/R = X(NN' + 1)/R \tag{3.3}$$

Moreover, for any integer l:

$$((XN' + lR)N + X)/R \bmod N = (XN'N + lRN + X)/R \bmod N \tag{3.4}$$

$$= (XN'N + X)/R \bmod N \tag{3.5}$$

Hence instead of computing $q = XN'$, we can compute $q = XN' \bmod R$ without changing the result of a. Now assuming $0 \leq X < RN$, the value of $a = (X + qN)/R$ is less than $2N$. Hence only one extra subtraction by N might be needed to obtain $a \bmod N$.

Algorithm 3.1: Montgomery reduction algorithm

Input: X, N, R^{-1} and N' such that $0 < R^{-1} < N$, $0 < N' < R$ and $R.R^{-1} - N.N' = 1$
Output: $a = X.R^{-1} \bmod N$

1 $q = (X \bmod R)N' \bmod R$
2 $a = (X + qN)/R$
3 **if** $a \geq N$ **then** $a = a - N$

The idea behind the Montgomery multiplication algorithm is to map the numbers $0 \leq A \leq N$ and $0 \leq B \leq N$ to their residues $\overline{A} = AR \bmod N$ and $\overline{B} = BR \bmod N$. It can be shown that this mapping is a one-to-one mapping. The product $AB \bmod N$ is mapped to:

$$(AB)R \bmod N = (AR)(BR)R^{-1} \bmod N \tag{3.6}$$

This is therefore equivalent to applying Algorithm 3.1 on the product $(AR)(BR)$.

The reduction and multiplication steps can be interleaved to produce an efficient algorithm. Let:

$$A = \sum_{k=0}^{m-1} a_k r^k, \qquad B = \sum_{k=0}^{m-1} b_k r^k, \qquad N = \sum_{k=0}^{m-1} n_k r^k \tag{3.7}$$

A, B and N are written as m-word numbers in radix r. We want to compute $P = ABR^{-1} \bmod N$ with $R = r^m$ [1]. In this case $gcd(R, N) = 1$ only if $gcd(r, N) = 1$.

Algorithm 3.2 iterates on the words a_i of A. At iteration i, we interleave the computation of the partial product $a_i B$ with a reduction step performed by Algorithm 3.1. Hence we compute

[1] A and B would previously be converted to their respective residue representations $\overline{A}$ and $\overline{B}$. Here we drop the $\overline{A}$ and $\overline{B}$ notations to simplify the equations

Algorithm 3.2: Radix-r Montgomery modular multiplication algorithm

Input: $A = \sum_{i=0}^{m-1} a_i r^i$, $B = \sum_{i=0}^{m-1} b_i r^i$, $N = \sum_{i=0}^{m-1} n_i r^i$, $N' = (-N)^{-1} \bmod r$, $0 \leq A, B \leq N$

Output: $P = ABr^{-m} \bmod N$

```
1 P = 0
2 for i = 0 to m − 1 do
3     q = (p_0 + a_i b_0) N' mod r
4     P = P + a_i B + qN
5     P = P/r
6 end
7 if P ≥ N then  P = P − N
```

$P = P + a_i B + qN$ followed by $P = P/r$. The value of q is computed as:

$$q = (P + a_i B)N' \bmod r = (p_0 + a_i b_0)N' \bmod r \tag{3.8}$$

The simplification in equation 3.8 is possible because $(P - p_0)$ and $(B - b_0)$ are multiples of r. Note that:

$$N' = (-N)^{-1} \bmod r = (r - n_0)^{-1} \bmod r \tag{3.9}$$

This is due to the choice of $R = r^m$ together with equation 3.1. Intuitively, each iteration of Algorithm 3.2 can be seen as a partial Montgomery reduction on the partially computed product. After m iterations we obtain:

$$P = (a_0 B + a_1 Br + ... + a_{m-1} Br^{m-1})r^{-m} \bmod N = ABr^{-m} \bmod N \tag{3.10}$$

If $r = 2$ and $n_0 = 1$, the calculation of q can be simplified as $q = (p_0 + a_i b_0)$, which is equal to the LSB of $(P + a_i B)$. This also ensures that $gcd(r, N) = 1$. The corresponding radix-2 Montgomery multiplication algorithm is given in Algorithm 3.3. In this chapter, we focus on designing a scalable hardware architecture for this algorithm. Many other variations of the Montgomery multiplication algorithm exist. A widely cited review of Montgomery multiplication algorithms is presented in [KABSK96].

Algorithm 3.3 only consists of simple additions, subtractions, comparisons and shifts. In

Algorithm 3.3: Radix-2 Montgomery modular multiplication algorithm

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $a_i,\ b_i,\ n_i \in \{0, 1\}$, $n_0 = 1$, $0 \leq A, B \leq N$

Output: $P = A.B.2^{-n} \bmod N$

```
1 P = 0
2 for i = 0 to n − 1 do
3     P = P + a_i.B
4     P = P + p_0.N
5     P = P/2
6 end
7 if P ≥ N then  P = P − N
```

particular, it does not need any multiplication or division unit. Interleaving the multiplication and the reduction steps also saves storage space as no large product needs to be stored.

The radix-2 Montgomery algorithm actually computes $A.B.2^{-n} \bmod N$ in $O(n)$ operations where n is the bitwidth of the inputs. It introduces an extra 2^{-n} factor compared to a classical modular multiplication. As explained in the general case, the inputs have to be converted to N-residue as follows:

$$\overline{A} = A.2^n \bmod N$$

$$\overline{B} = B.2^n \bmod N$$

It can be done by Montgomery multiplying the inputs by the constant $C = 2^{2n} \bmod N$.
The result of the Montgomery multiplication of $\overline{A}$ by $\overline{B}$ becomes:

$$\overline{P} = A.B.2^n \bmod N$$

$\overline{P}$ is converted back to a normal representation by Montgomery multiplying it by one. The calculation of the constant C, the initial conversions to N-residue and the conversion back to normal representation are also in $O(n)$ but introduce a computational overhead. Hence for a single modular multiplication, the Montgomery algorithm can be less efficient than a classical multiplication followed by a modular reduction due to the effect of constant factors. However, for applications that compute a sequence of modular multiplications, we only need to perform the conversions once. In this case, the speedup introduced by the use of the Montgomery

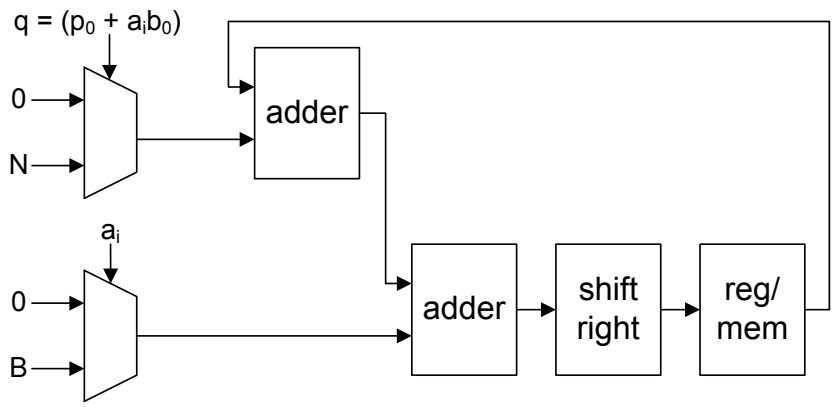


Figure 3.1: Montgomery multiplication with two ripple-carry adders

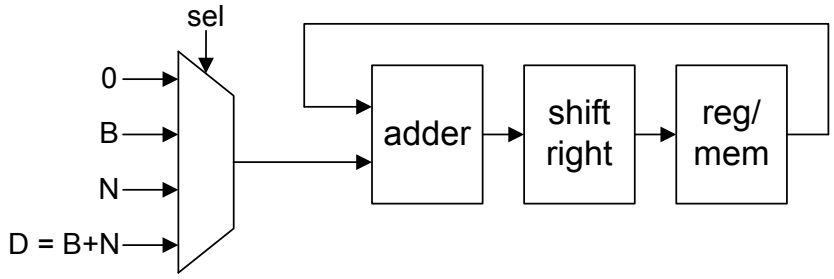


Figure 3.2: Montgomery multiplication with one ripple-carry adder

algorithm often outperforms the computational overhead especially for large inputs. The right-to-left exponentiation algorithm presented in section 3.2.3 is an example of such an application.

3.2.2 Hardware implementations of the Montgomery algorithm

A relevant review of different modular multiplication techniques together with their hardware implementations is presented in [NdMM06]. Many FPGAs implementations of the Montgomery algorithm have been presented in the literature. We describe here some major contributions.

In [FL05] the authors present the ARSA core, a scalable RSA architecture taking advantage of the high-speed adder/subtractor logic of Altera FPGAs in arithmetic mode. The Montgomery multiplier of the ARSA core computes the Montgomery product, decomposed into blocks, using two simple adders along with two multiplexers for the input selection as shown in Fig. 3.1. The quicker version of the ARSA core runs at 200 MHz and takes an area of 900 Altera Logic Elements (LE). It is capable of computing a 1024-bit modular exponentiation in about 80 ms using Montgomery multiplication. This design is very area-efficient but slow.

The design presented in [DM02] also takes advantage of the dedicated carry-chain logic of

FPGAs. The authors use a processing element similar to Fig. 3.2. This design comes from the observation that at iteration i, only four different values can be added to P depending on the values of a_i and p_0: 0, B, N or $B + N$. For bitwidths exceeding the size of an FPGA column, the authors pipeline the carry-chain to reduce performance loss.

Another interesting design is presented by Blum in [BP99]. The full multiplier uses a systolic array of processing elements, each of them being a Montgomery multiplier cell. This implementation is much more area consuming than a one-block implementation but it is also faster. According to [FL05], this implementation takes more than ten times the area of an ARSA core for a bitwidth of 1024, and is two times faster. A radix-16 version of the systolic array design of [BP99] is presented in [BP01]. A scalable pipelined Montgomery multiplier whose number of processing elements can be chosen according to the area availability and the speed requirements is presented in [CBLC04]. Every pipeline element of this design undergoes two stall operations. Moreover, the way the pipeline is organised requires a complex RAM decoder. In all these designs the processing element is also based on the ripple-carry adder-based architecture shown in Fig. 3.2.

In [BST02] and [NAPP+05], a comparison between a Montgomery multiplier implementation with two carry-save adders (CSA) and one with a single CSA is made. The architectures are shown in Fig. 3.3 and Fig. 3.4 respectively. A carry-save adder takes three numbers x, y and z and add them together to obtain two numbers: the sum S and the carry C. A n-bit CSA consists of n full-adders and performs the addition $x+y+z = S+C$ in $O(1)$ time. For comparison, a standard ripple-carry adder has a latency in $O(n)$ due to the need for carry propagation. The redundant representation (S, C) used by the CSA prevents the need for carry propagation. On recent FPGAs with dedicated carry-lookahead logic, a CSA is faster than an ordinary adder for large enough inputs. For example, on a Spartan-6 XC6SLX150T this is the case for inputs larger than 32 bits. The one-CSA Montgomery multiplication implementation turns out to take half the area of the two-CSA implementation with better speed performance. The carry-save Montgomery algorithm is explained in more detail in section 3.3. Other CSA-based Montgomery architectures are presented in [MMM03]. The main weakness of these designs is that they are not parametric and therefore do not scale well with the increase in size and

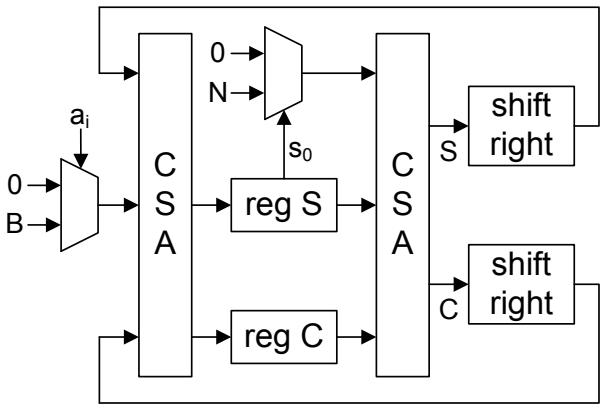


Figure 3.3: Montgomery multiplication with two CSAs

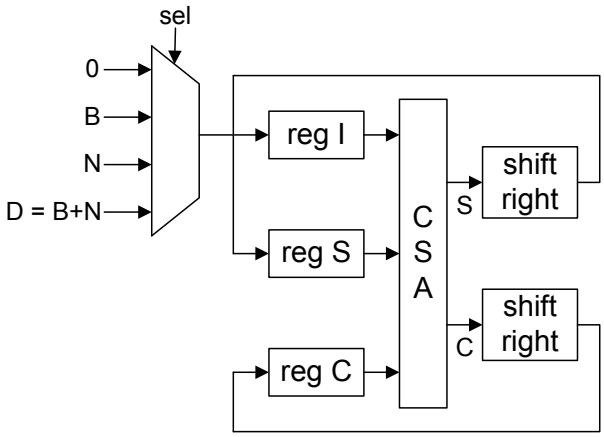


Figure 3.4: Montgomery multiplication with one CSA

speed of FPGAs. Our design adds pipeline and replication capabilities to this implementation to explore as much design space as possible.

All of the designs presented so far perform multiplication by repeated additions and are implemented in LUTs. For a bitwidth n, they have a complexity of $O(n^2)$ in terms of Area $\times$ Time (AT) product [BST02]. Montgomery multipliers that take advantage of the embedded multipliers (or DSPs) of recent FPGAs also exist [TTL03, Suz07, OS08, CELL10]. In [OS08], the authors develop a parametric Montgomery multiplier architecture targeting low time-area products. It consists of an array of processing elements. Each processing element is based on two embedded multipliers. The parameters of the architecture are the number of processing elements, the radix and the number of words used. This paper also makes an exhaustive speed and area comparison of 1024-bit Montgomery multiplier architectures. All these multiplier-based designs perform Montgomery multiplication by blocks. If s is the number of blocks in an n-bit input, they perform $O(s^2)$ multiplications. In [CELL10] the authors present a

Table 3.1: Comparison of hardware implementations of Montgomery multiplication

Design	Radix	LUT/DSP based	Scalability parameters
ARSA [FL05]	2	LUT	Size of adders
Daly [DM02]	2	LUT	Pipeline depth of carry chain
Bunimov [BST02]	2	LUT	No
Amanor [NAPP+05]	2	LUT	No
McIvor [MMM03]	2	LUT	No
Blum [BP99]	2	LUT	Number of PEs
Blum [BP01]	16	LUT	No
Cheung [CBLC04]	2	LUT	Number of PEs
Tang [TTL03]	2^{17}	DSP	No
Suzuki [Suz07]	2^{17}	DSP	No
Oksuzoglu [OS08]	2^{17}	DSP	Number of PEs
Chow [CELL10]	2^{32}	DSP	No
Ours	2	LUT	Number of pipeline stages Number of replications

Montgomery multiplier based on the Karatsuba algorithm with a $O(n^{log3/log2})$ time complexity. A 512-bit implementation of this design running at 300 MHz on a Virtex-6 FPGA is 60 times faster than an optimised multi-threaded software implementation on an Intel Xeon 2.5 GHz CPU. However, this implementation is extremely area-consuming. For a bitwidth of more than 1024 bits, it cannot fit in any Virtex-6 FPGA.

Table 3.1 summarises the differences between the different Montgomery architectures presented in this section. Our Montgomery multiplier architecture is presented in section 3.3. Speed, area and additional scalability results are given in section 3.5. All our architectures are LUT-based in order to allow easier customisation and better scalability across different generations of FPGAs.

3.2.3 Montgomery modular exponentiation

Equations 2.15 and 2.16 show that the basic operation at the core of RSA encryption and decryption is modular exponentiation. A simple but commonly used algorithm for modular exponentiation is given in Algorithm 3.4.

This algorithm is called the right-to-left binary exponentiation algorithm. It requires $O(n)$ modular multiplications for n-bit inputs. The algorithm iterates on the bits of E from the LSB to the MSB. At each iteration i, the variable $P_i = X^{2^i} \bmod N$ is squared modulo N to obtain

Algorithm 3.4: Right-to-left binary exponentiation algorithm

Input: X,E,N with $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0, 1\}$
Output: $Z_n = X^E \bmod N$

```
1 Z_0 = 1, P_0 = X
2 for i = 0 to n − 1 do
3     P_{i+1} = P_i^2 mod N                          /* Square */
4     if e_i = 1 then
5         Z_{i+1} = Z_i.P_i mod N                    /* Multiply */
6     else
7         Z_{i+1} = Z_i
8 end
```

$P_{i+1} = X^{2^{i+1}} \bmod N$ (squaring operation). If $e_i = 1$, the accumulated product Z_i is multiplied by P_i modulo N (multiplication operation), otherwise it remains the same. This step relies on the following formula:

$$X^E \bmod N = X^{\sum_{i=0}^{n-1} e_i 2^i} \bmod N \tag{3.11}$$

$$= \prod_{i=0}^{n-1} X^{e_i 2^i} \bmod N \tag{3.12}$$

After n iterations, n being the bitwidth of E, Z_n contains $X^E \bmod N$. In practice, if we perform the test of e_i first, only two variables P and Z, updated at each iteration, are needed.

Other exponentiation algorithms exist. A detailed survey of existing exponentiation methods is presented in [Gor98].

3.3 Scalable Montgomery multiplier architecture

Our work on Montgomery multiplication is first presented in [Mas09] and an improved version is published in [LMLEC10]. Our goal is to design a scalable Montgomery multiplier whose speed and area can be adapted to the application.

We choose a one-CSA based Montgomery multiplier as the basic block of our design. As discussed earlier, a CSA is faster than a ripple-carry adder with no area overhead. Algorithm 3.5 from [BST02] is used. This algorithm is a straightforward adaptation of Algorithm 3.3 to the carry-save format. The diagram of a multiplier cell is given in Fig. 3.5. The bitwidth of

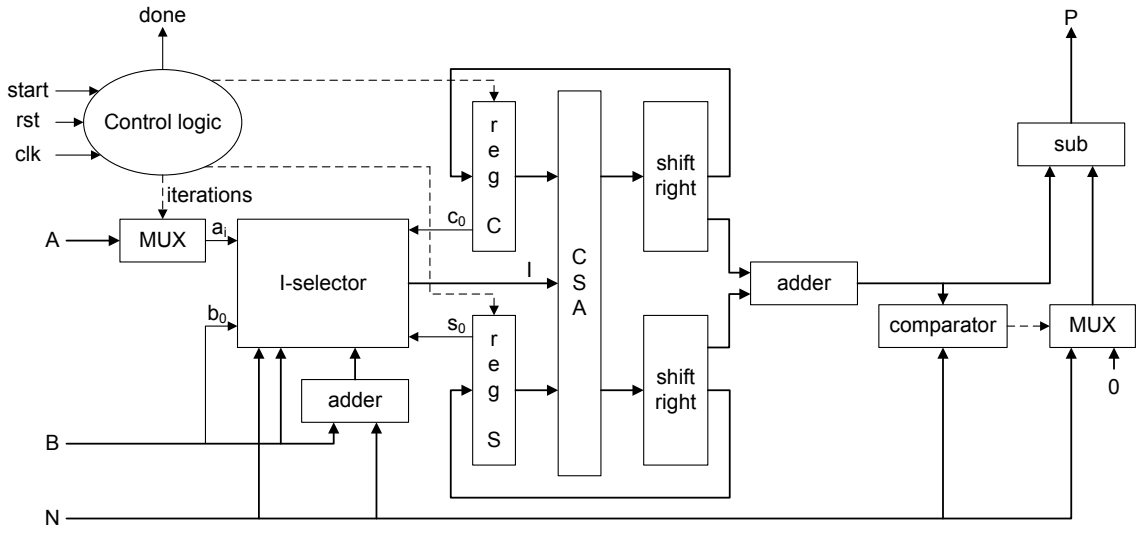


Figure 3.5: Diagram of a Montgomery multiplier cell

this cell is set as a design parameter.

Algorithm 3.5: Carry-save Montgomery algorithm for modular multiplication

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $(a_i, b_i, n_i) \in \{0,1\}^3$
Output: $P = A.B.2^{-n} \bmod N$

1 $S = 0$
2 $C = 0$
3 $D = B + N$ /* Pre-computation of B+N */
4 **for** $i = 0$ **to** $n-1$ **do**
5 **if** $(s_0 = c_0)$ *and* $a_i = 0$ **then** $I = 0$ /* Equivalent to $a_i = p_0 = 0$ */
6 **if** $(s_0 \neq c_0)$ *and* $a_i = 0$ **then** $I = N$ /* Equivalent to $a_i = 0$, $p_0 = 1$ */
7 **if** $(s_0 \oplus c_0 \oplus b_0) = 0$ *and* $a_i = 1$ **then** $I = B$ /* Equivalent to $a_i = 1$, $p_0 = 0$ */
8 **if** $(s_0 \oplus c_0 \oplus b_0) = 1$ *and* $a_i = 1$ **then** $I = D$ /* Equivalent to $a_i = 1$, $p_0 = 1$ */
9 $S, C = S + C + I$ /* Carry-save addition */
10 $S = S \; div \; 2$ /* Shift of the sum */
11 $C = C \; div \; 2$ /* Shift of the carry */
12 **end**
13 $P = S + C$ /* Full addition */
14 **if** $P \geq N$ **then** $P = P - N$

We apply two techniques to our design: pipelining and serial replication. Pipelining improves the throughput of the design and serial replication improves its latency. Three main challenges have to be addressed to design a parametric Montgomery multiplier using these techniques:

Challenge 1. Algorithm 3.5 cannot be easily parallelised due to the data dependencies between the consecutive values of the sum S and the carry C in the main loop. At iteration $i+1$, the values of S and C from iteration i are needed to compute the new value of I and the

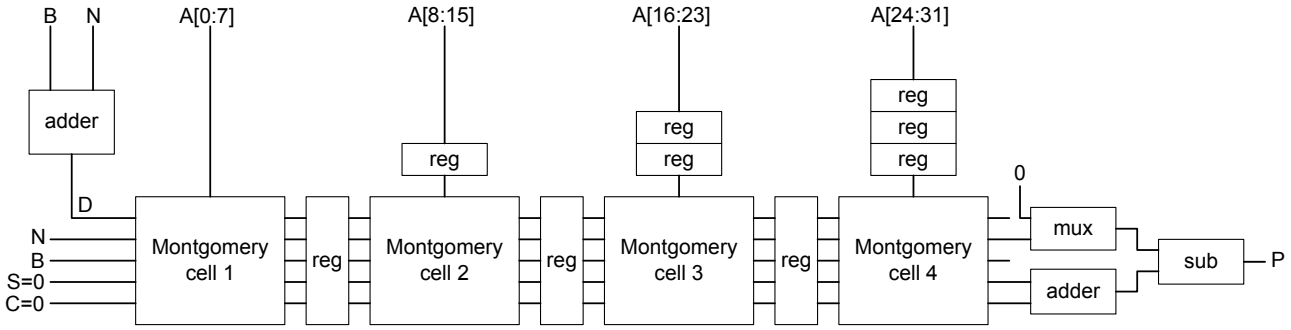


Figure 3.6: Structure of a 32 bit pipelined Montgomery multiplier with 4 pipeline stages

new values of S and C.

Challenge 2. To explore as much design space as possible, the bitwidth, the number of replications and the number of pipeline stages should be parameters.

Challenge 3. The control logic should adapt to the values of these parameters.

Fig. 3.6 shows the basic structure of our pipelined design. We call p the number of pipeline stages. Each Montgomery cell is a modified version of the basic cell presented earlier. We cope with Challenge 1 by allowing each basic cell to perform a consecutive part of the iterations. For this principle to work, the final addition and subtraction blocks are removed from this cell and the number of iterations performed by each cell becomes a parameter. The basic cell is enhanced with the ability to load the S and C registers from the inputs. The current values of S and C are also available at the output of each cell.

Inside each pipeline block, the CSA can be replicated as many times as needed. We call r the total number of CSAs in series. Part of a basic cell for $r = 2$ is shown in Fig. 3.7. The data dependencies problem prevents us from simply duplicating the CSA and perform several iterations in parallel. Instead, several CSAs along with the shift logic are put in series. This is equivalent to unrolling the loop of Algorithm 3.5 $(r-1)$ times. The I-selector is also replicated as the values of I differ for each CSA and at each iteration. Replicating the CSA $(r-1)$ times reduces the number of clock cycles required to perform a multiplication by a factor of r. The area overhead is less than $(r-1)$ times the area of the basic cell because only a part of the basic cell is replicated. In practice, replication also increases the latency between the outputs

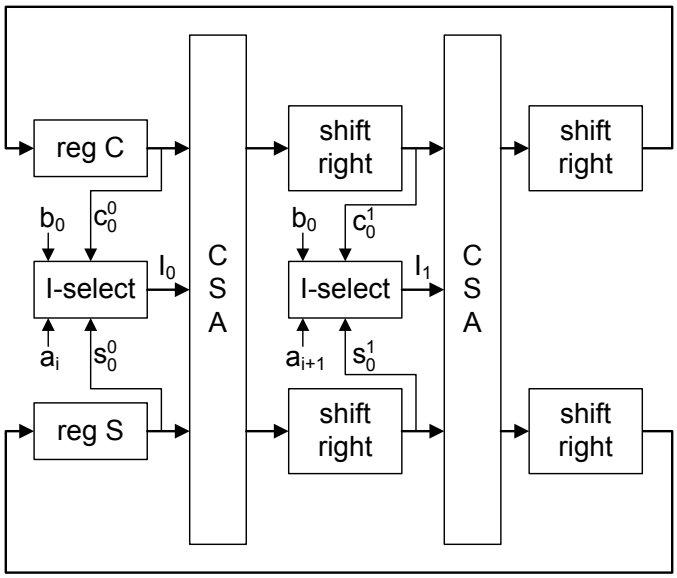


Figure 3.7: Focus on the CSA and I-selector of a basic cell for $r = 2$

of registers S and C and their respective inputs. This reduces the maximum clock frequency at which the multiplier can run.

Allowing the bitwidth (n) and the pipeline depth (p) to take any value makes it more difficult to divide the number of iterations between blocks. When n is not a multiple of p, each block cannot perform the same number of iterations. We address this challenge by adding an extra iteration to the first n mod p pipeline blocks. This leads to n mod p blocks computing $\lfloor n/p \rfloor + 1$ iterations, and $(n - n \bmod p)$ blocks computing $\lfloor n/p \rfloor$ iterations.

Inside a pipeline block, in order to allow the number of replications (r) to take any value, the result can be extracted from any CSA. This solution deals with the case when the number of iterations the cell has to perform is not a multiple of the number of replications.

A flexible pipeline control is implemented. This control deals with the updates of two types of registers: the registers between blocks and the triangular register array. The register triangular structure consists of arrays of registers controlled as FIFO queues. The inputs enter all the FIFOs at the same time when the `done` signal of the very first cell is triggered. The element at the head of the FIFO of a given cell leaves the queue when the corresponding cell has finished to use it, that is when its `done` signal is triggered.

Inside a pipeline block, the control logic manages the extraction of the result from the correct CSA, depending on the number of replications chosen and the number of iterations this

particular block has to perform. This is implemented through a multiplexer whose select line is set according to the value of an iteration counter.

Let us take a full example summarizing this section with $n = 1024$, $p = 5$ and $r = 7$. As $\lfloor n/p \rfloor = 204$ and $n \bmod p = 4$, the first 4 pipeline blocks compute $204 + 1 = 205$ iterations and the last one computes 204 iterations. Our flexible pipeline control deals with this issue. Inside each block, we want 7 replications. In the first 4 pipeline blocks, we loop through the 7 CSAs 30 times ($\lceil 205/7 \rceil$). The result is extracted from CSA number 2 (205 mod 7) after the last iteration. In the last pipeline block, we loop through the 7 CSAs 30 times ($\lceil 204/7 \rceil$). The result is extracted from CSA number 1 (204 mod 7) after the last iteration.

3.4 Application to modular exponentiation

In this section, we demonstrate how our scalable Montgomery multiplier can be used as the main building block of a modular exponentiator. In particular, we show how the pipeline of the multiplier can speed up the exponentiation speed.

3.4.1 Basic Montgomery exponentiator

Our basic Montgomery exponentiator is also presented in [Mas09,LMLEC10]. The exponentiator implements Algorithm 3.4 using the pipelining and replication capabilities of the multiplier.

Two main problems have to be solved for this design to be efficient in terms of speed and area. First, the pipeline of the multiplier has to be optimally used. The number of pipeline stages of the multiplier giving best performance for use with the exponentiator is equal to two. This is due to the fact that in Algorithm 3.4, P_{i+1} depends on P_i and Z_{i+1} depends on both P_i and Z_i. If we use more than two pipeline stages, the multiplier's pipeline cannot be kept full due to these data dependencies.

Second, integrating our multiplier in a bigger design can reduce its running frequency due to critical path problems. The latency of the adders and subtractors used in the multiplier are indeed a bottleneck for large bitwidths. To reduce the critical path, all the ripple-carry adders and the subtractors are pipelined.

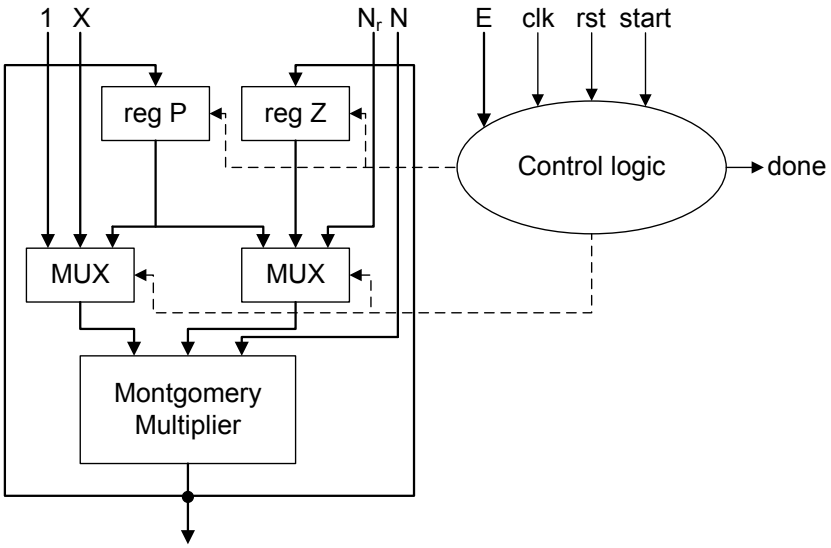


Figure 3.8: Montgomery exponentiator

The design of our exponentiator is represented in Fig. 3.8. It has three main parameters: the number of multiplier's pipeline stages, the number of replications for the multiplier's pipeline cells and the pipeline depth of the adders/subtractors. At each iteration, the current values of P and Z are stored in registers. The control logic manages the inputs to give to the multiplier and the data to write back. It also controls the multiplier.

3.4.2 Adding interleaved exponentiation support

As said before, the data dependencies in the exponentiation algorithm prevent us from efficiently using a multiplier with more than two pipeline stages to compute a single exponentiation. However in a crypto-system several modular exponentiations might be needed to encrypt or decrypt a piece of data. For example, using RSA and the CRT to sign a $2n$-bit number with a $2n$-bit key already requires two n-bit exponentiations. The idea is to interleave the iterations of several exponentiations in the multiplier's pipeline. This technique enables us to cut the data dependencies of the different exponentiations that are running simultaneously. Fig. 3.9 shows the benefits of interleaving several exponentiations. This figure compares the number of clock cycles needed to perform one exponentiation in a 4-stage pipeline to the number of clock cycles needed to perform two interleaved exponentiations in the same pipeline. To simplify we suppose that each pipeline stage takes one clock cycle and that 4 multiplications are needed to complete an exponentiation. Recall that at iteration i, P_{i+1} depends on P_i, and Z_{i+1} depends on both P_i

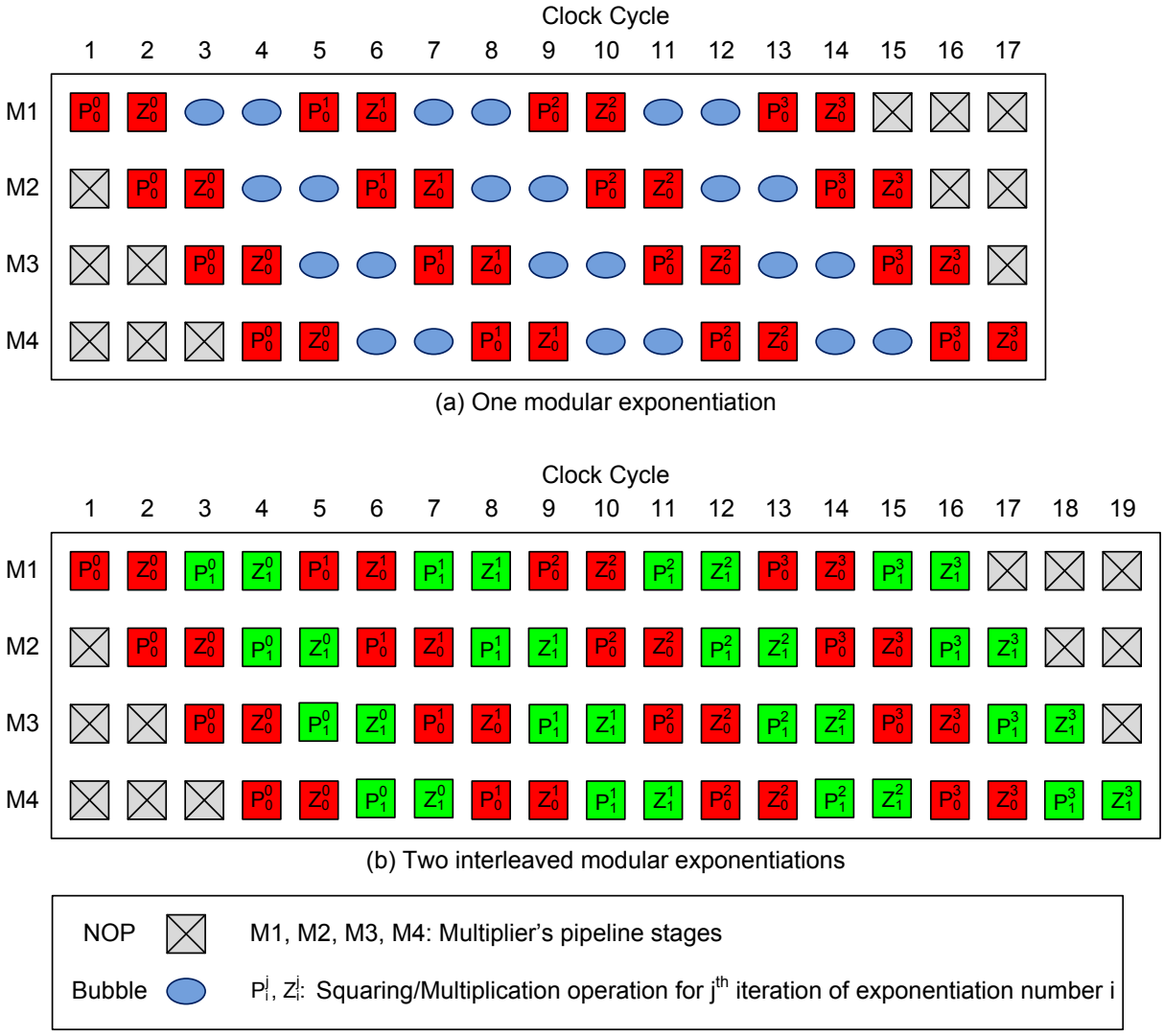


Figure 3.9: Modular exponentiations in a 4-stage pipeline

and Z_i. Hence we cannot perform a single exponentiation in a 4-stage pipeline without stalling the pipeline as shown in Fig. 3.9a. In that case performing a single exponentiation takes 17 clock cycles. If we interleave two exponentiations in the 4-stage pipeline (Fig. 3.9b), we see that the pipeline is kept full. It amounts to replace the stalls of Fig. 3.9a with actual computation. Only two extra clocks are needed to drain the pipeline and we can perform two exponentiations in 19 clock cycles. This increases the throughput of the exponentiator by almost a factor of 2 compared to the non-interleaved method.

We modify our exponentiator to add interleaved exponentiation support. The diagram of the new exponentiator is shown in Fig. 3.10. The values of the inputs corresponding to different exponentiations are stored in RAMs. The maximum number of interleaved exponentiations is a parameter of our design. It determines the depths of the different RAMs. Two multiplexers

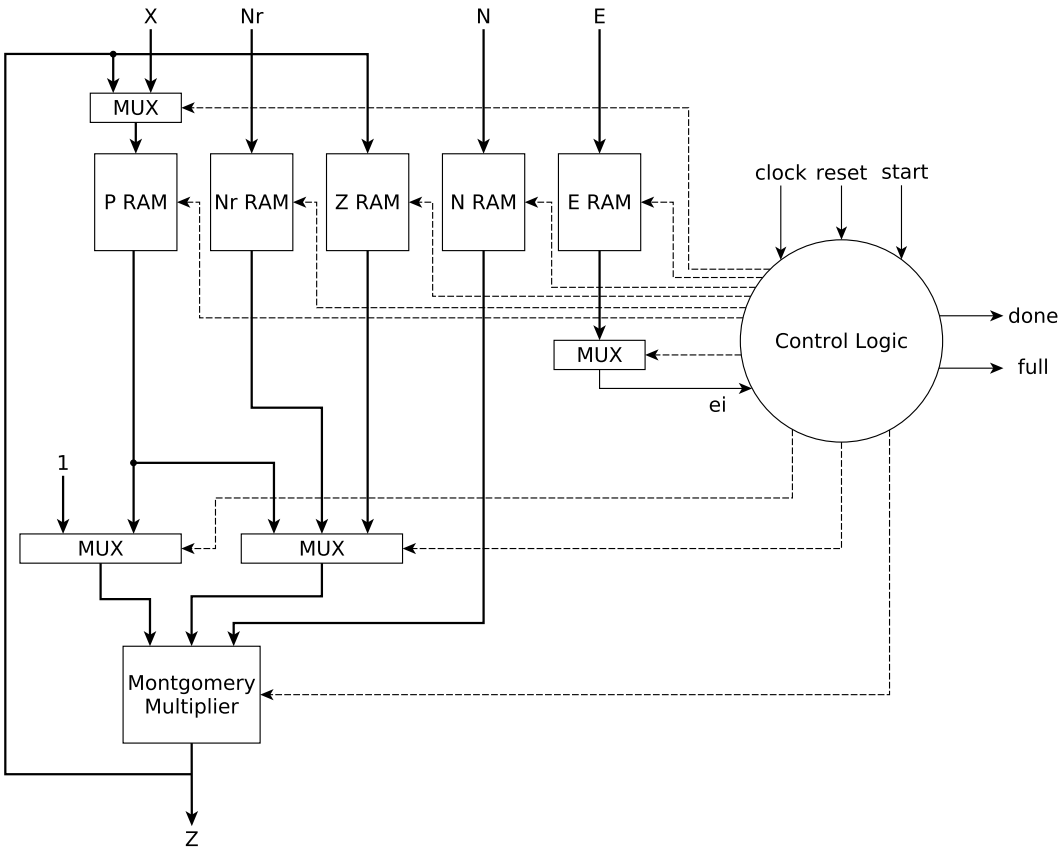


Figure 3.10: Modular exponentiator with interleaved exponentiation support

enable the selection of the inputs to the multiplier according to the operation performed. The operation can be one of the following:

- Conversion of X to N-residue
- Conversion of 1 (initial value of Z) to N-residue
- Calculation of the new value of P (at each iteration)
- Calculation of the new value of Z (at each iteration)
- Conversion of Z back to normal representation

New inputs corresponding to a new exponentiation can be loaded at any time by the user provided that the maximum number of interleaved exponentiations is not reached (this is indicated by the `full` signal).

At each iteration, the value at the address of the P-RAM corresponding to the current exponentiation is updated, followed by the value at the same address in the Z-RAM. Note that

the P-RAM can be loaded with either the value of the input X or the output of the Montgomery multiplier. The exponentiation are interleaved following the model shown in Fig. 3.9b. The simultaneous exponentiations are numbered. The control algorithm used to fill the multiplier's pipeline consists of the following steps:

1. Choose a new exponentiation number
2. Start the computation of the values of P and Z for the current iteration of the current exponentiation
3. Update the iteration number of the current exponentiation
4. Goto 1.

Step 1 is implemented through an arbiter. The arbiter ensures that the iterations of each exponentiation enter the multiplier's pipeline in a round-robin fashion and that the exponentiations finish in order.

Finally, in order to write-back the results given by the multiplier to the correct memory addresses, a FIFO keeps track of the label (exponentiation and iteration numbers) of the computation being performed in each stage of the pipeline.

3.5 Design space exploration and comparison to existing architectures

In this section, we give implementation results for our parametric designs. We compare our multiplier to the best FPGA implementations in the literature and show how it scales with the number of replications and the number of pipeline stages. We also give results for our exponentiator and confirm the benefits of interleaving exponentiations. In all the tables, we use the designs presented in section 3.2.2 for comparison, provided that enough relevant data are available in the corresponding papers. Our architectures are coded in Verilog. Appendix C discusses our testing methodology. We place and route our designs with Xilinx ISE 14.1 for Xilinx Virtex-5 FPGAs. The implementation parameters are given in appendix D.1.1.

In particular we perform place-and-route in performance evaluation mode, which means that instead of using external timing constraints, the tool automatically generates timing constraints for each internal clock separately and adjusts the constraints during execution [Xil12b, p. 123]. The performance achieved by this mode is not necessarily the best possible performance. Better clock frequencies might be obtained by running multiple place-and-route instances for each design with different clock constraints and cost tables. However using the performance evaluation mode is a good compromise between speed of the fully routed design and place-and-route time.

Table 3.2 compares the execution time of our multiplier with other implementations for $n = 1024$ bits. When comparing the reported performance, one should take into account that they do not all target the same FPGA family. In particular our implementations target more recent FPGAs. Virtex-5 FPGAs have 6-input LUTs whereas the other devices reported in this table have 4-input LUTs. Hence we expect the other implementations to occupy slightly less LUTs on Virtex-5. Moreover given the advances in process technology, the same architecture is likely to run faster on Virtex-5 than on earlier devices. On the other hand, in order to allow a fair comparison with the software implementation, we report place-and-route results whereas the authors of some of the other implementations ([OS08] and possibly [NAPP$^+$05, MMM03, DM02]) report synthesis results. The area reported after synthesis is generally close to reality. However, the reported frequency is a very optimistic guess of the actual achievable frequency after place-and-route. Hence these results only give a rough idea of how our implementations perform compared to other implementations in the literature. For the software version, we report the mean and the standard deviation (σ) of the execution time for one million multiplications of random numbers. For 1024 bits the GMP library uses optimised assembly implementations of the schoolbook methods for multiplication and divisions, which are $O(n^2)$ algorithms.

Our multiplier achieves latencies in range with other implementations. In particular for $p = 1$ and $r = 8$, the latency of our multiplier is almost 3 times lower than the software implementation and only 30% slower than Tang's design. However, real-world applications, such as exponentiation, often execute many multiplications. Hence comparing throughputs is more relevant. Our multiplier without any pipelining and replication is faster than most

Table 3.2: Performance comparison of 1024 bit multipliers

Design	Device	Clock (MHz)	Area (LUT)	Latency (μs)	Throughput (Mb/s)
Ours ($p = 8$, $r = 7$)	XC5VLX330T-2	51.48	204 923	4.23	2510.26
Ours ($p = 2$, $r = 8$)	XC5VLX110T-3	70.83	60 629	2.57	1098.94
Ours ($p = 1$, $r = 8$)	XC5VLX110T-3	90.19	32 937	2.00	710.42
Tang [TTL03]	XC2V3000-6	90.11	N/A	1.49	688.60
Ours ($p = 2$, $r = 4$)	XC5VLX110T-3	82.47	36 038	3.76	649.60
Ours ($p = 1$, $r = 4$)	XC5VLX110T-3	147.34	20 641	2.09	584.79
Ours ($p = 2$, $r = 1$)	XC5VLX110T-3	170.56	17 585	6.32	339.79
Ours ($p = 1$, $r = 1$)	XC5VLX110T-3	195.31	11 415	5.51	194.93
GMP 4.2.4 [gmp] `mpz_mul/mpz_mod`	Intel Core 2 Duo E7400	2800	N/A	mean: 5.45 σ: 0.53	mean: 189.65 σ: 17.19
Oksuzoglu [OS08] (1020 bit)	XC3S500E-4FG320C	119	6 906	7.62	134.35
McIvor [MMM03]	XC2V3000	75.23	23 234	13.46	76.08
Daly [DM02]	XCV1000	54.61	10 116	19.03	54.45
Amanor [NAPP+05]	XVC2000E-6	49.0	8 064	21.00	48.86

existing implementations with a throughput of 194.93 Mb/s. It is also faster than the software implementation of modular multiplication using the very optimised GMP library on one core of an Intel Core 2 Duo E7400 running at 2.8 GHz. For $p = 8$ and $r = 7$, our multiplier is 4 times faster than the best hardware implementation and 13 times faster than the optimised software implementation. We can still get better performance by increasing r and p if we target an FPGA with enough available area. Our design scales as the device scales and can therefore adapt to future FPGA families.

Table 3.3 compares our multipliers in terms of Latency $\times$ Area and Area / Throughput to existing designs. The latency is reported in clock cycles and the throughput in (clock cycles)$^{-1}$. By doing so we make the results independent of the speed of the device on which the designs are implemented. The Latency $\times$ Area and Area / Throughput results are normalised to our best architectures for both metrics respectively. For $p = 4$ and $r = 8$ our multiplier ranks first in terms of Area / Throughput. It is 58% more efficient than Oksuzoglu's. It is interesting to see that $(p, r) = (4, 8)$ is a design point favouring throughput over area, whereas Oksuzoglu's design clearly favours area over throughput. For $p = 1$ and $r = 8$, our design is very close to Oksuzoglu's in terms of Latency $\times$ Area product and better than all the other designs.

Fig. 3.11 to 3.16 show how our multiplier scales with r and p. In these figures, we only

Table 3.3: Latency × Area and Area / Throughput normalised to our best designs for 1024 bit multiplication

Design	Area (LUTs)	Latency (clock cycles)	Throughput $(\text{clock cycles})^{-1}$	Latency × Area	Area / Throughput
Ours ($p = 4$, $r = 8$)	116 789	186	1/34	3.66	**1**
Ours ($p = 1$, $r = 8$)	32 938	180	1/130	**1**	1.08
Oksuzoglu [OS08] (1020 bit)	6 906	907	1/907	1.06	1.58
Amanor [NAPP$^+$05]	8 064	1027	1/1027	1.40	2.09
ARSA 128 [FL05]	900	10404	1/10404	1.58	2.36
Daly [DM02]	10 116	1039	1/1027	1.77	2.62
Ours ($p = 1$, $r = 1$)	11 415	1076	1/1026	2.07	2.95
ARSA 64 [FL05]	700	18564	1/18564	2.19	3.27
ARSA 32 [FL05]	500	34980	1/34980	2.95	4.40
ARSA 16 [FL05]	300	67860	1/67860	3.43	5.13
McIvor [MMM03]	23 234	1026	1/1026	4.02	6.00

report synthesis results. This is justified as we are only interested in the relative speed and area consumption of our architectures. Note that the outliers for $(p, r) = (7, 7)$ and $(p, r) = (8, 6)$ in Fig. 3.11 to Fig. 3.14 are due to the inability of the synthesis tool to perform a logic optimisation on the I-selector of the basic cells for a specific pipeline block. This optimisation is performed for all other values of p and r on all the pipeline blocks. It is also performed if we synthesize the problematic pipeline block alone with proper wrapping logic. Not enough detail is given on the proprietary synthesis algorithm to explain this behaviour.

As expected, Fig. 3.11 and Fig. 3.12 show that the latency of the design depends mostly on r. However, as shown in Fig. 3.12 the latency does not decreases linearly with r. In fact increasing r also decreases the maximum frequency of the design by slowing down the latency of the critical path.

The throughput of the design increases linearly with p as shown in Fig. 3.13. As expected the benefits of increasing r becomes marginal for large r as the slow down of the critical path overcomes the gain in clock cycles. In that sense the multiplier scales better with p than with r. The scalability of our multiplier in terms of area consumption is shown in Fig. 3.15 and Fig. 3.16. For small values of r ($r \leq 4$), the area overhead of doubling the number of pipeline stage is less than 100% whereas it is around 100% for greater values of r. This is due to the fact that for small values of r the area taken by the logic required at the input and output of

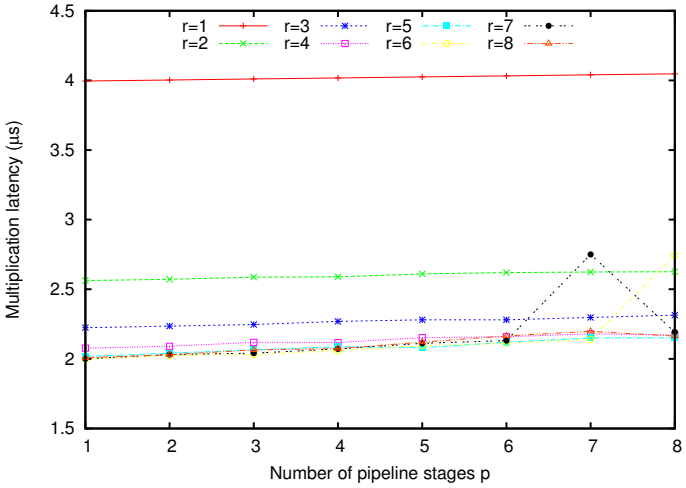


Figure 3.11: Multiplication latency after synthesis on a XC5VLX330T against p for different values of r

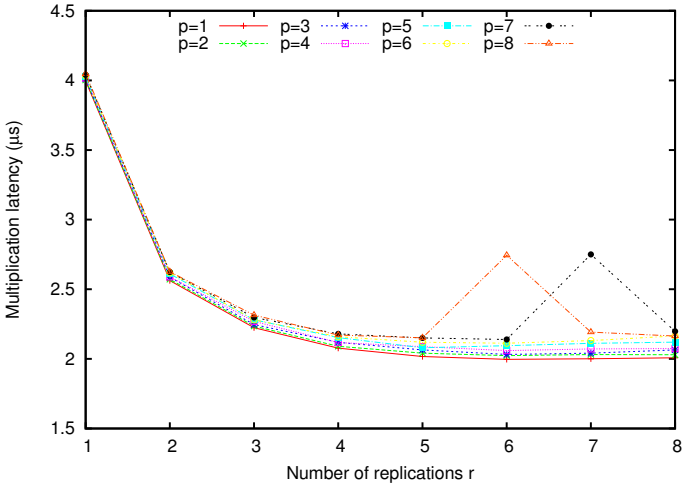


Figure 3.12: Multiplication latency after synthesis on a XC5VLX330T against r for different values of p

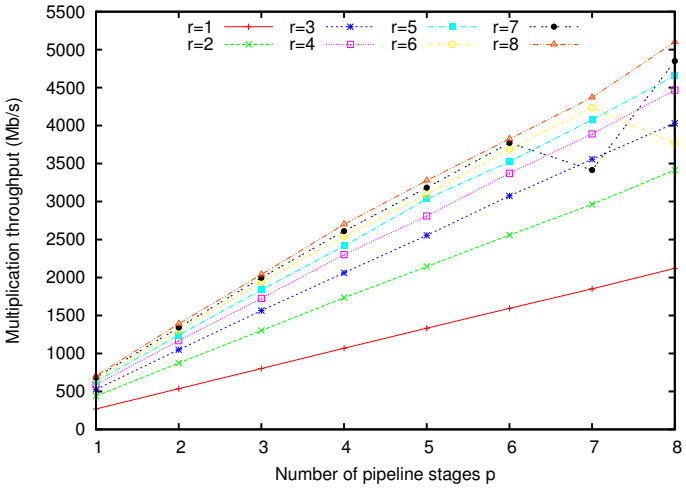


Figure 3.13: Multiplication throughput after synthesis on a XC5VLX330T against p for different values of r

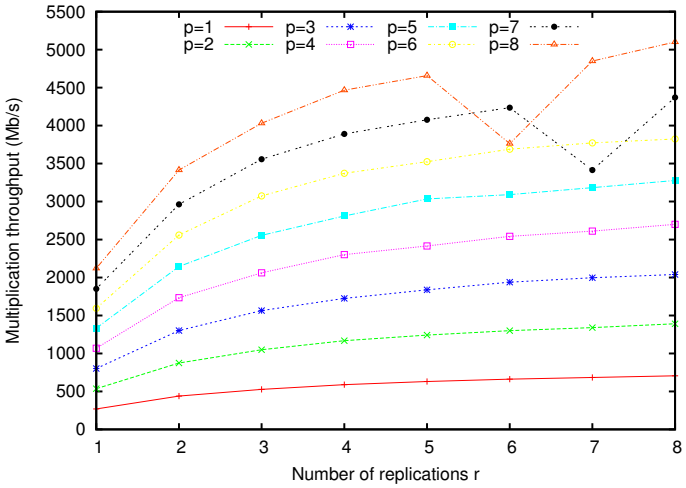


Figure 3.14: Multiplication throughput after synthesis on a XC5VLX330T against r for different values of p

the pipeline is not negligible compared to the area of the multiplier cells. Keeping p constant, if we double r the corresponding area overhead is much less than 100%. Increasing the number of replications is less area-consuming than increasing the number of pipeline stages as a smaller part of the basic cell (only the CSA and I-selector logic) is duplicated.

Table 3.4 compares the execution time of our exponentiator with other implementations for $n = 1024$ bits. For the hardware results we assume that half of the bits of the exponent are set and we report place-and-route results. The pipeline depth of all ripple-carry adders and all subtractors is fixed to 16 in order to reduce the critical path. For $p > 2$ we use the interleaved

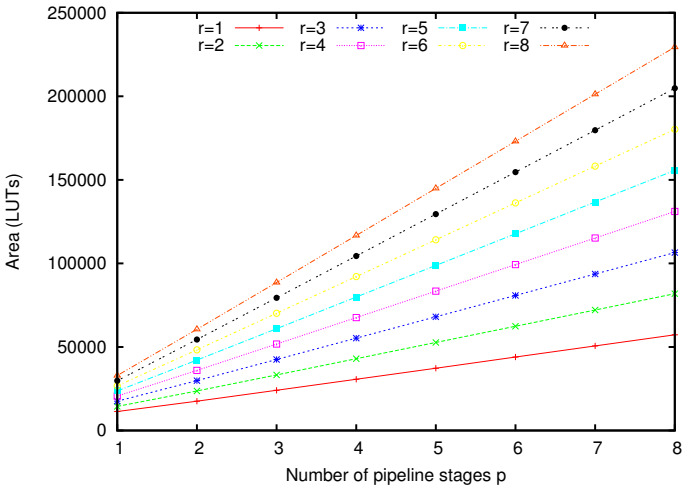


Figure 3.15: Area taken by our multiplier after synthesis on a XC5VLX330T against p for different values of r

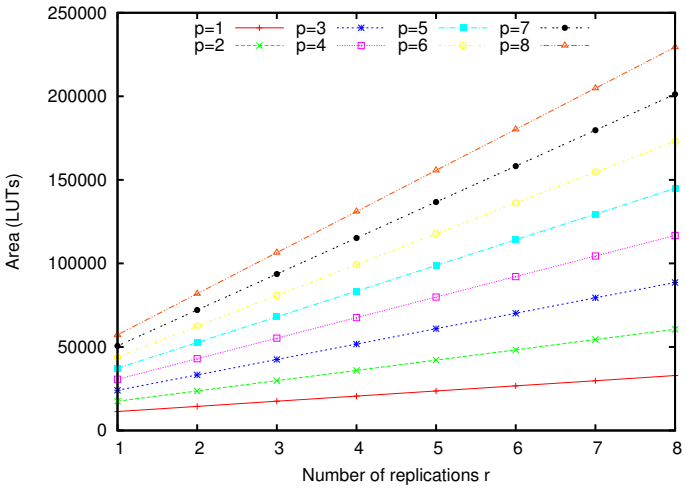


Figure 3.16: Area taken by our multiplier after synthesis on a XC5VLX330T against r for different values of p

version of the exponentiator. The execution time is normalised to the minimum number of interleaved exponentiations required to keep the multiplier's pipeline full [2]. For the software version, we report the mean and standard deviation for one million exponentiations of random numbers.

For $p = 2$ and $r = 7$, our non-interleaved implementation running at 94.09 MHz is 2 times faster than the optimised software implementation using the GMP library on one core of an Intel Core 2 Duo E7400 running at 2.8 GHz. By interleaving multiple exponentiations ($p = 8$, $r = 5$) we can run more than 5 times faster than the software and 2 times faster than the best Montgomery modular exponentiator in the literature which uses DSPs.

Table 3.5 compares our best non-interleaved and interleaved designs in terms of the Time $\times$ Area product with the other designs. As in Table 3.3, the results are independent of the clock speed of the device used. Our exponentiator is more efficient than all the designs but Blum's and Tang's. It is interesting to note that both designs use high-radix Montgomery multipliers. Blum's design use a radix-16 multiplier. Tang's uses a radix-2^{17} multiplier based on 96 embedded multipliers. For Tang's design, the multipliers area is not taken into account in our calculation of the Time $\times$ Area product.

Fig. 3.17 reports the exponentiation time against r for exponentiators with and without interleaving support and for different values of p after synthesis. For $p = 4$ (respectively

[2] 2 interleaved exponentiations for $p = 4$, 4 interleaved exponentiations for $p = 8$

Table 3.4: Performance comparison of 1024 bit exponentiators

Design	Device	Clock (MHz)	Area	Ex. Time (ms)
Ours ($p = 8$, $r = 5$)	XC5VLX330T-3	95.37	157 674 LUTs	0.75
Suzuki [Suz07]	XC4VFX12-10SF363	200/400	7 874 LUTs + 17 DSP48	1.71
Ours ($p = 2$, $r = 7$)	XC5VLX110T-3	94.09	60 687 LUTs	2.24
Tang [TTL03]	XC2V3000-6	90.11	28 668 LUTs + 62 multipliers	2.28
GMP 4.2.4 [gmp] `mpz_powm`	Intel Core 2 Duo E7400	2800	N/A	mean: 4.23 σ: 0.12
Ours ($p = 2$, $r = 2$)	XC5VLX110T-3	88.95	29 940 LUTs	6.57
Oksuzoglu [OS08] (1020 bit)	XC3S500E-4FG320C	119	15 398 LUTs + 20 multipliers	7.95
Ours ($p = 2$, $r = 1$)	XC5VLX110T-3	138.68	23 791 LUTs	8.00
Ours ($p = 1$, $r = 2$)	XC5VLX110T-3	102.70	20 695 LUTs	10.29
Blum [BP01]	XC4000	45.7	13 266 LUTs	11.94
Ours ($p = 1$, $r = 1$)	XC5VLX110T-3	114.40	17 623 LUTs	18.41
Daly [DM02]	XCV1000	49.63	26 738 LUTs	21.21
Blum [BP99]	XC4000	52.1	9 730 LUTs	40.49

Table 3.5: Time × Area product normalised to our best design for 1024 bit exponentiation

Design	Area (LUTs)	Latency (clock cycles)	Time × Area
Tang [TTL03]	28 668	205824	0.59
Blum [BP01]	13 266	545832	0.73
Ours ($p = 8$, $r = 3$)	94 016	106015	**1**
Ours ($p = 1$, $r = 8$)	39 143	268783	1.06
ARSA 128 [FL05]	900	15980544	1.44
ARSA 64 [FL05]	700	28514304	2.00
Blum [BP99]	9 730	2109456	2.06
ARSA 32 [FL05]	500	53729280	2.70
Daly [DM02]	26 738	1052637	2.82
ARSA 16 [FL05]	300	104232960	3.14

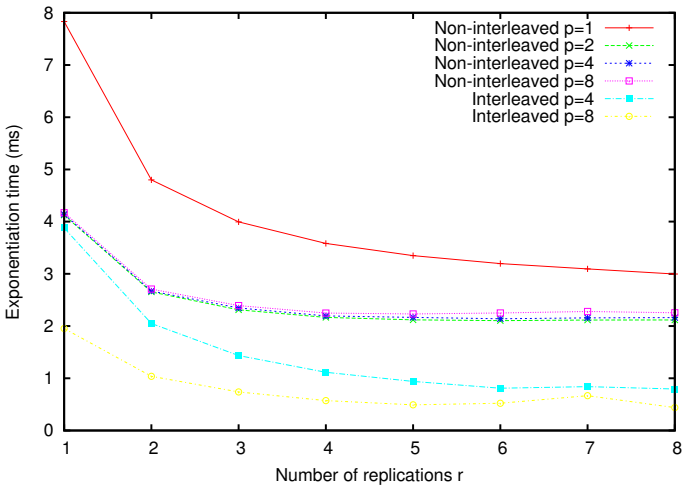


Figure 3.17: Exponentiation time after synthesis on a XC5VLX330T against r for different values of p

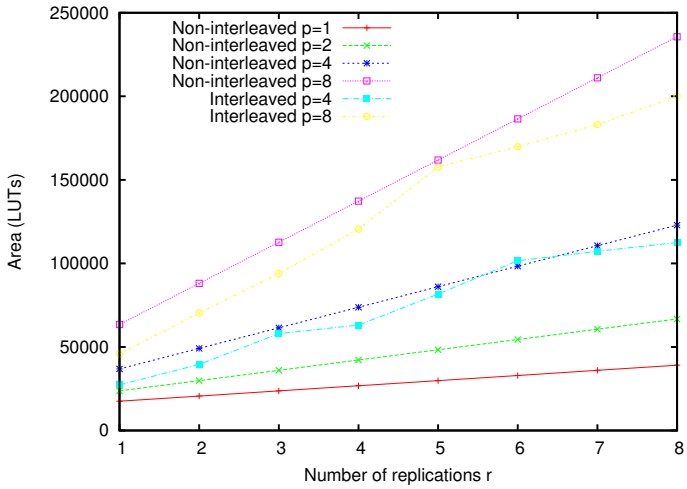


Figure 3.18: Area taken by our exponentiator after synthesis on a XC5VLX330T against r for different values of p

$p = 8$) with interleaving support, two (respectively four) exponentiations are interleaved in the pipeline. The reported execution time is then normalised to the number of exponentiations performed. This plot shows that without interleaving support, using more than two pipeline stages is useless. In fact, it even slightly increases the exponentiation time because extra clock cycles are introduced for pipeline management without any increase in the multiplier's throughput. On the other hand, we see that using interleaved exponentiation greatly improves the performance of our exponentiator. The interleaved exponentiator using 4 pipeline stages is faster than the 2-stage non-interleaved exponentiator. It is actually less than two times faster for $r \leq 4$ due to better achievable running frequencies for the non-interleaved version. For $r \leq 5$, we can still get another two times speedup by increasing the number of pipeline stages from 4 to 8. For $r > 5$ and $p = 8$, the high utilization of the FPGA lead to a stiff decrease in the maximum clock frequency due to increased routing delays.

Fig. 3.18 shows that our interleaving exponentiator does not require more logic area than our basic exponentiator[3]. As a matter of fact, the only area-consuming hardware added to the exponentiator are the RAMs used to store the operands and the intermediate values of each interleaved exponentiation. These can either be implemented in block RAMs or/and in registers, which are not a bottleneck in terms of area. For example our implementation for

[3]The interleaved exponentiator is actually smaller than the non-interleaved version. The non-interleaved version is based on the regular architecture presented in chapter 4. The interleaved version is recoded from scratch.

$r = 8$ and $p = 8$ only makes use of 101 Block RAMs out of 324 available in the XC5VLX330T, leaving enough Block RAMs to interleave 16 extra exponentiations. This is more than enough given that for $p = 8$, interleaving more than 4 exponentiations will not improve the normalised exponentiation time.

Therefore, these results show that with no logic area overhead, our interleaving exponentiator is faster than our basic exponentiator. More importantly, the new exponentiator scales with both r (up to the limits previously described) and p.

3.6 Summary

In this chapter, we present new hardware architectures for Montgomery modular multiplication and exponentiation. The loop-carried dependency in the Montgomery algorithm prevents a parallel hardware implementation. In order to explore a large design space in terms of speed-area trade-off, our Montgomery multiplier architecture uses variable pipeline stages and variable serial replications. These two techniques are both equivalent to unrolling the loop of the Montgomery algorithm. Serial replication duplicates the main computational element of the design in order to perform several iterations of the loop in the same clock cycle. This effectively reduces the latency of a multiplication, that is the number of clock cycles needed to complete one multiplication. However replication increases the critical path of the design and therefore its running frequency as confirmed in section 3.5. Pipelining duplicates the entire multiplier cell and registers the output of each cell. This reduces the number of iterations performed in each cell, but not the overall number of clock cycles needed to complete one multiplication. Hence pipelining only increases the throughput of the multiplier. Moreover, apart from the possible routing congestions created by increasing the complexity of the design, pipelining does not have a direct negative effect on the running frequency of the design. In that sense, the multiplier scales better with the number of pipeline stages than it does with the number of replications. These observations are developed further in the following chapter, which will extend our architecture to a general class of algorithm with loop dependencies carried from one iteration to the next.

Our Montgomery multiplier architecture is applied to modular exponentiation, which is

the main operation behind RSA encryption/decryption. Due to the data dependencies in the exponentiation algorithm, for a single exponentiation the multiplier can only be used at full throughput for 2 pipeline stages or less. In order to overcome this limitation, we interleave several exponentiations in the multiplier's pipeline. This technique amounts to replacing the pipeline bubbles by actual computation and has no area overhead. Applications of modular exponentiation to probabilistic primality testing are presented in chapter 5.

Our scalable Montgomery multiplier implemented on a Virtex-5 FPGA can achieve throughputs 4 times higher than the best existing hardware implementation and 13 times higher than an optimised software implementation. Our multiplier is also more efficient than existing designs (for which clock cycles data are available) in terms of Latency $\times$ Area and Area / Throughput. By interleaving multiple exponentiations our exponentiator can run more than 5 times faster than optimised software and 2 times faster than the best modular exponentiation hardware reported in the literature. Only designs based on high-radix multipliers are more efficient than our exponentiator in terms of Time $\times$ Area product. Our Montgomery multiplier and exponentiation architectures scale well with the number of pipeline stages and replications. Hence better performance will be achieved on upcoming devices such as the Virtex-7, which will be faster and larger.

Chapter 4

Mapping Loop Structures onto Parametrised Hardware Pipelines

This chapter generalises the idea used to scale our Montgomery multiplier presented in the previous chapter. We present a coarse-grained method to automatically map a general class of algorithms consisting of a loop with loop dependencies carried from one iteration to the next to a parametric hardware design with pipelining and replication features similar to the Montgomery multiplier architecture. The general algorithm consists of three stages. The pre-computation stage initialises the result vector. The main stage iterates on the bits of some of the inputs to update the result vector according to its previous value and the values of the inputs. Finally, the post-computation stage terminates the algorithm. A technology-independent parametric model of the proposed design is developed to capture the variations of area and throughput with the number of pipeline stages and replications. Our model allows rapid optimisation of design parameters by a few pre-synthesis operations. We present an optimisation method based on the model. We show that our method facilitates design space exploration by quickly predicting the area taken by the design as well as its maximum frequency and throughput.

4.1 Motivation

In chapter 3, we introduced two parametric designs. The main advantages of parametric designs are their scalability and their reusability. They can be reused as IP cores in many projects

with different speed-area trade-offs. However, finding the optimal values for the parameters to meet a given design goal can be difficult and time-consuming. An exhaustive approach implementing designs for all the values of the parameters in a given search interval cannot be achieved in most projects where the time-to-market is an issue. Therefore we need a model of the design that facilitates estimating its parameters under speed and area constraints. Such area and speed estimation models are often application-dependent [BLC09, YLS+08] or technology-dependent [EJCT00, SJ08].

Previous work on resource and performance estimations for reconfigurable devices has mainly been focused on high-level RTL estimation. The common method used for resource estimation involves identifying the basic operations in the algorithm which can be represented as a Data Flow Graph (DFG) [EJCT00] or an RTL netlist [SJ08]. A library of operators characteristic of the device is then used to estimate the resources needed by the algorithm. This process usually requires approximations in order to fit the parameters of the identified operation (bitwidth, number of inputs, etc) to the operator in the library [KNRK06]. Such estimation tools are readily available in commercial software such as Xilinx System Generator [MQF91] and PlanAhead [SJ08]. The RTL resource estimation tool used by Xilinx is claimed to be 60 times faster than synthesis [SJ08], which allows fast design space exploration. Similar methods can be used to estimate the performance of the design by using a device-dependent library of delays for the estimated resources.

These methods are general and inherently fine-grained. A most specific and coarse-grained approach focused on energy-consumption is developed by Becker et. al in [BLC09, BLC10]. The authors target runtime reconfigurable software-defined radio applications. Their approach needs a priori information about the implementation, usually obtained by running a few synthesis operations.

The literature on algorithm mapping onto reconfigurable hardware and associated design space exploration is vast. For instance, in [SHD02] a mixed compiler/synthesis tools approach is presented. Several compiler optimisations are performed on the algorithm before translation to RTL. The design space is then searched using resource estimations obtained from synthesis results.

Mapping of unrolled loop onto pipelines has also been extensively studied [Wei97, BP98, STL04]. In particular the authors of [STL04] propose the Reconfigurable Dataflow approach that maps loops with loop-carried dependencies onto efficient pipelines. This method is more fine-grained and general than our approach to the extent that it supports loop-carried dependencies of various sizes and loop indices determined at runtime. However the authors do not provide any resource and performance model for the mapped implementation, which makes design space exploration difficult.

Unlike other more fine-grained loop pipelining and resource estimation methods, the method presented in this chapter is restrained to a particular application domain and uses a coarse-grained model of the corresponding hardware mapping. As a matter of fact, our base module is a coarse-grained cell which already consists of many fine-grained operators. Our method is particularly useful for quick optimisation of design parameters through design space exploration. To this extent it is close to the method presented in [BLC09, BLC10]. However, the authors of [BLC09, BLC10] focus on a parallelizable class of algorithms while the focus of this work is on algorithms that cannot be parallelized due to loop-carried dependencies. The restrained application domain of our model makes the resource and performance estimations as well as the design space exploration more accurate than general estimation methods.

4.2 Main idea

We focus on a class of algorithms with loop dependencies carried from one iteration to the next. Common applications falling into this category are bitwise algorithms for which the result is updated at each iteration based on the corresponding bit of one or several inputs. This is for instance the case for the Montgomery multiplication algorithm and the square-multiply exponentiation algorithm presented in chapter 3 as well as the Lucas primality test presented in chapter 5. Some important arithmetic algorithms such as the integer square root [Par00, Cre] and the restoring division algorithms also follow this pattern. Algorithms performing a reduce operation also exhibit such loop dependencies. This is the case for several algebraic algorithms such as matrix-vector and matrix multiplications, for which products of row and column elements are iteratively accumulated. Finally, loop dependencies carried from

one iteration to the next are also common in approximation algorithms for which a solution is refined at each iteration based on the result from the previous one. The Newton-Raphson method is one of the best known algorithms of this type.

Usually algorithms with loop-carried dependencies cannot be easily parallelized. However, several techniques can be used for efficient hardware implementation. Pipelining and replication are introduced in chapter 3. To recall these concepts, let us take a simple example using the carry-save adder (CSA) based Montgomery multiplication of Algorithm 3.5. Let us assume that we want to perform a $n = 32$ bit Montgomery multiplication. A simple hardware architecture for the loop would reuse the same processing logic (Montgomery cell) 32 times, storing the intermediate values of S and C in registers. This architecture is compact but slow as the latency of one multiplication is 32 clock cycles.

If we are willing to trade area for performance, the Montgomery cell can be replicated. For example if we replicate the Montgomery cell once, we obtain $r = 2$ copies of the cell in series in a Montgomery block, as shown in Fig. 3.7. Hence the latency of one multiplication is now $32/2 = 16$ clock cycles. The area is slightly less than doubled as some control logic can be shared. However by replicating the cell, we increase the delay through the block and hence the critical path delay. In practice, replication is only useful if the minimum period that we want to achieve is higher than the critical path delay.

Now let us assume that we want to perform several multiplications. In our replicated design we would have to wait for the first multiplication to terminate before starting a new one, leading to a throughput of one multiplication every 16 clock cycles. To increase the throughput of the design, we can pipeline the Montgomery block as shown in Fig. 3.6. As opposed to replication, pipelining buffers the output of each duplicated block. Here the entire block is duplicated, and therefore doubling the number of pipeline stages doubles the logic area. For $p = 4$ pipeline stages, each block is assigned $32/4 = 8$ iterations. Each block contains $r = 2$ cells. Hence it takes $8/2 = 4$ clock cycles for a block to compute its part of the iterations. The latency of the architecture is still 16 clock cycles but a new multiplication can now be started every 4 clock cycles, leading to a throughput of one multiplication every 4 clock cycles. The following sections formalise these ideas.

4.3 General bit by bit processing algorithm with loop-carried dependency

In this analysis, we consider algorithms that can be described in the general form presented in Algorithm 4.1.

Algorithm 4.1: General bit by bit processing algorithm

Input:
$\mathbf{A} = \{A_0, ..., A_{k-1}\}$ with $A_j = \sum_{i=0}^{n-1} a_{(j,i)} 2^i$ for $0 \leq j < k$
$\mathbf{a_i} = \{a_{(0,i)}, ..., a_{(k-1,i)}\}$ for $0 \leq i < n$
$\mathbf{B} = \{B_0, ..., B_{l-1}\}$ with $B_j = \sum_{i=0}^{\beta_j - 1} b_{(j,i)} 2^i$ for $0 \leq j < l$

Output:
$\mathbf{R} = \{R_0, ..., R_{m-1}\}$ with $R_j = \sum_{i=0}^{\rho_j - 1} r_{(j,i)} 2^i$ for $0 \leq j < m$

```
1 R = pre(A, B)                          /* Variables initialisation */
2 for i = 0 to n − 1 do
3   R = main(a_i, B, R)                  /* Loop-carried dependency through R */
4 end
5 R = post(A, B, R)                      /* Final additions, shifts, etc */
```

Table 4.1: Parameters of the general algorithm

Parameters	Description
k	Size of vector $\mathbf{A}$ of inputs $\{A_0, ..., A_{k-1}\}$
n	Bitwidth of inputs $\{A_0, ..., A_{k-1}\}$
l	Size of vector $\mathbf{B}$ of inputs $\{B_0, ..., B_{l-1}\}$
β_j	Bitwidth of input B_j
m	Size of vector $\mathbf{R}$ of inputs $\{R_0, ..., R_{m-1}\}$
ρ_j	Bitwidth of input R_j

The different parameters are described in Table 4.1. This algorithm has an optional pre-computation stage (line 1). This stage can consist of initialisation of variables, initial shifts, etc. Then, the algorithm iterates on the number of bits n of the elements $\{A_0, ..., A_{k-1}\}$ of input vector $\mathbf{A}$. At each iteration of the loop, the result vector $\mathbf{R}$ is updated through the `main()` function (line 3). At iteration i, this function takes the previous value of vector $\mathbf{R}$, the input vector $\mathbf{B}$ and the i-th bits $\mathbf{a_i} = \{a_{(0,i)}, ..., a_{(k-1,i)}\}$ of each element of vector $\mathbf{A}$. A possible post-computation stage terminates the algorithm (line 5).

In practice, every algorithm with one loop that has loop dependencies carried from one

iteration to the next, and for which some inputs are processed bit by bit, can be represented in this general form. Table 4.2 shows how we can map the Montgomery multiplication algorithm, the exponentiation algorithm and the integer square root algorithm to our general algorithm. For each algorithm the inputs, the outputs, the number of iterations (n) and the three functions `pre()`, `main()` (at iteration i) and `post()` are identified. Note that for the integer square root algorithm to conform to our general algorithm, its input bits have to come from two elements of input vector **A**. In fact, the bit by bit processing of the elements of **A** does not limit the generality of our approach as k n-bit physical elements can be zipped together to form n k-bit logical inputs if required. In section 4.7 we choose these three reference designs to evaluate the accuracy of our model.

4.4 Mapping of the algorithm to the parametric hardware description

We focus on the main loop of Algorithm 4.1 and map it to the general hardware design described in Fig. 4.1 and Fig. 4.2.

The n iterations are divided between p blocks organised in a pipeline fashion as shown in Fig. 4.1. Pipeline block number i performs $iter(i)$ successive iterations of the loop where for all $i \in [1, p]$:

$$iter(i) = \left\lfloor \frac{n}{p} \right\rfloor + extra(i) \tag{4.1}$$

$$extra(i) = \begin{cases} 1 & \text{if } i < n \bmod p \\ 0 & \text{if } i \geq n \bmod p \end{cases} \tag{4.2}$$

The number of iterations are evenly distributed between pipeline blocks and if p does not divide n, the first $n \bmod p$ blocks perform an extra iteration. Therefore, block number i

Table 4.2: Specialisation of the general algorithm for three applications

	CSA Montgomery	Exponentiation	Integer Square root
Inputs	$\mathbf{A} = \{A_0\}$, $\mathbf{B} = \{B_0, B_1\}$ $A_0 = A = \sum_{i=0}^{s-1} a_i 2^i$ $B_0 = B = \sum_{i=0}^{s-1} b_i 2^i$ $B_1 = N = \sum_{i=0}^{s-1} n_i 2^i$ $(a_i, b_i, n_i) \in \{0,1\}^3$	$\mathbf{A} = \{A_0\}$, $\mathbf{B} = \{B_0\}$ $A_0 = E = \sum_{i=0}^{s-1} e_i 2^i$, $e_i \in \{0,1\}$ $B_0 = X$, $B_1 = N$	$\mathbf{A} = \{A_0, A_1\}$, $\mathbf{B} = \emptyset$ $A_0 = (a_1 a_3 ... a_{s-1})_2$ $A_1 = (a_0 a_2 ... a_{s-2})_2$ where $A = (a_{s-1} ... a_1 a_0)_2$ is the number we want to take the square root of and $a_i \in \{0,1\}$
Outputs	$\mathbf{R} = \{R_0, R_1, R_2\}$ $R_0 = S = A.B.2^{-n} \bmod N$ $R_1 = C$ $R_2 = D$	$\mathbf{R} = \{R_0, R_1\}$ $R_0 = Z = X^E \bmod N$ $R_1 = P$	$\mathbf{R} = \{R_0, R_1\}$ $R_0 = Rem$, $R_1 = Root$
n	s	s	$s/2$
$pre()$	$S = 0$, $C = 0$, $D = B + N$	$Z = 1$, $P = X$	$Rem = 0$, $Root = 0$

$main()$	**if** $(s_0 = c_0)$ *and* $a_i = 0$
	then
	$I = 0$
	if $(s_0 \neq c_0)$ *and* $a_i = 0$
	then
	$I = N$
	if $(s_0 \oplus c_0 \oplus b_0) = 0$ *and* $a_i = 1$ **then**
	$I = B$
	if $(s_0 \oplus c_0 \oplus b_0) = 1$ *and* $a_i = 1$ **then**
	$I = D$
	$S, C = S + C + I$
	$S = S$ *div* 2
	$C = C$ *div* 2
		if $e_i = 1$ **then**
		$Z = Z.P \bmod N$
		$P = P^2 \bmod N$
			$Root = (Root << 1)$
			$a = (a_{(0,i)} a_{(1,i)})_2$
			$Rem = (Rem << 2) + a$
			$Div = (Root << 1) + 1$
			if $Div \leq Rem$ **then**
			$Rem = Rem - Div$
			$Root = Root + 1$

$post()$	$S = S + C$ **if** $S \geq N$ **then** $S = S - N$	-	-

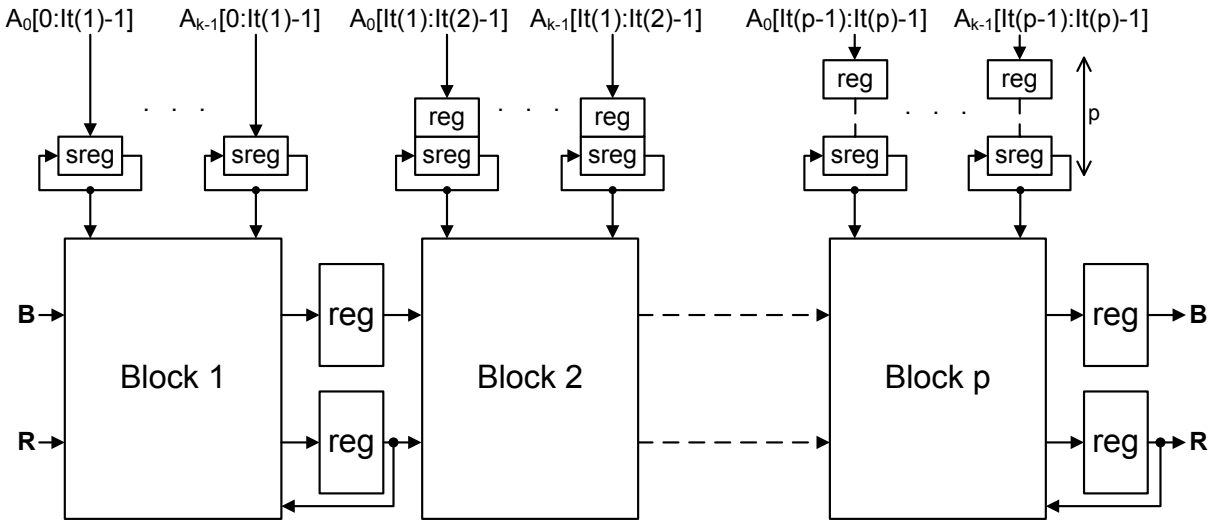


Figure 4.1: Pipeline of the parametric hardware description

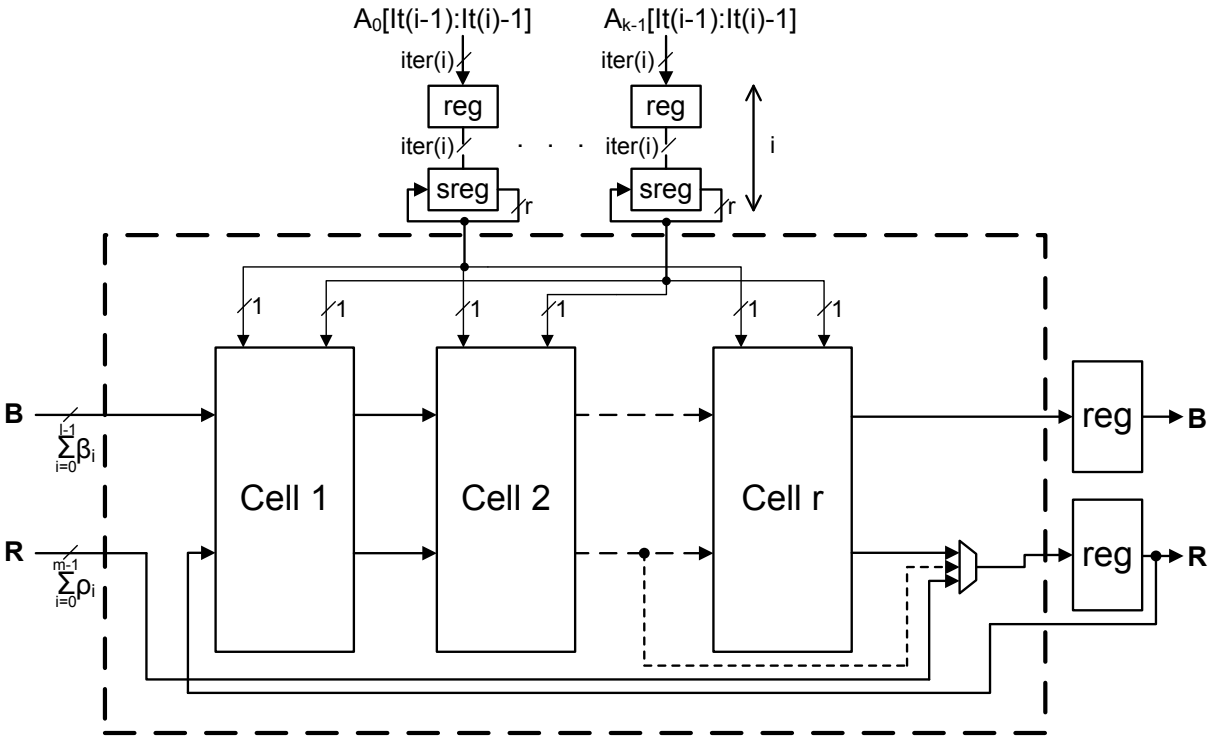


Figure 4.2: Inside pipeline block i

processes the bits $It(i-1)$ to $(It(i)-1)$ of inputs A_0 to A_{k-1} where:

$$\begin{aligned} It(0) &= 0 \\ It(i) &= \sum_{k=1}^{i} iter(k) \quad \forall i \in [1,p] \end{aligned} \tag{4.3}$$

The other inputs to block number i are:

- the current value of result vector $\mathbf{R}$, calculated by block number $(i-1)$ if $i \geq 2$ or given by the pre-computation stage if $i = 1$,
- the value of input vector $\mathbf{B}$.

Each pipeline block consists of a processing cell performing the actual computation, some registers, and extra logic needed for the control. As well as organising the basic blocks in a pipeline fashion, we allow the processing cell of each block to be replicated in series without buffering as shown in Fig. 4.2. This feature is relevant if the processing cell consists only of combinational logic. If the operations performed by a cell take several clock cycles, replication will not improve the number of clock cycles spent in each pipeline block. For instance let us assume that the operations of a cell take 10 clock cycles and we need to perform 2 iterations in each pipeline block. With one cell, the single cell performs both iterations one after the other for a total of 20 clock cycles. If we replicate the cell once, the first cell performs the first iteration. The second cell waits for the first cell to terminate then it performs the second iteration. Hence the total time through the block is still 20 clock cycles and no speed improvement is achieved.

If each pipeline block has r main processing elements, r iterations are processed in series during one clock cycle. Hence the number of clock cycles spent in pipeline block i per operation is:

$$lat(i) = \left\lceil \frac{iter(i)}{r} \right\rceil \tag{4.4}$$

We call this number the latency of pipeline block i. $lat(i)$ is expressed in clock cycles and is therefore independent of the frequency of the design. This equation shows that an extra clock cycle is needed if r does not divide $iter(i)$. In this case the results of block i need to be

extracted from cell number $iter(i)$ mod r during this extra clock cycle.

Between each block, registers save the values of vectors **R** and **B** to be given as inputs to the next block. The registers for **R** are updated at each clock cycle with the results of cell number r or cell number $iter(i)$ mod r. The registers for **B** are only updated each time the operation of the preceding block is done. Finally, a triangle of registers at the top ensures that the pipeline blocks are always given the correct values from their corresponding elements of vector **A**. In the triangle of registers, the bottom registers connecting to each pipeline block are circular shift registers (denoted `sreg` in Fig. 4.2). They provide the cells in the pipeline block with the inputs corresponding to the current iteration.

Mapping our general algorithm to this particular hardware structure has three main advantages. First, the parametric nature of this structure (through the two parameters r and p) allows the designer to explore a large design space and therefore to consider different speed/area trade-offs. Second, we will show that the throughput of the design can be improved by increasing the values of r and p, the biggest limitation being the area available. This is particularly interesting for applications targeting high-speed performance. Finally, our structure is regular and the generation of the pipeline and replication logic is automated.

4.5 Modelling of the architecture

We develop a model of our parametric design presented in the previous section enabling rapid optimisation for throughput under area and speed constraints. The use of such a model reduces development time as fewer synthesis operations are needed to obtain satisfactory values for r and p.

4.5.1 Latency and throughput

Our module needs to perform n iterations to complete the main computation if no replication is introduced. Let $t_{p,e}$ be the time to perform such an iteration, corresponding to c clock cycles

at a frequency f:

$$t_{p,e} = \frac{c}{f} \tag{4.5}$$

If $c = 1$, the main processing element of a block can be replicated. If $c > 1$, as we cannot replicate the main processing element it takes $n.c$ clock cycles to perform the n iterations. Therefore the total time to perform the main operation is:

$$t_p = \begin{cases} \sum_{i=1}^{p} lat(i).t_{p,e} & \text{if } c = 1 \\ n.t_{p,e} & \text{if } c > 1 \end{cases} \tag{4.6}$$

This corresponds to the time taken by the system to compute the first result, that is the latency of the hardware module in seconds. Note that in practice, it is clear that replication increases the delay through a pipeline block and therefore reduces the maximum frequency f of the clock as cells are put in series without buffering. This is considered in section 4.5.2.

To simplify the equations, we assume that the latency of the pre-computation and post-computation operations are shorter than the latency of a pipeline block. If not the case, the throughput bottleneck is in the slower of the two stages, not in the main computation. The throughput ϕ of the design (number of operations per unit of time when the pipeline is full) depends on both the pipeline length p and the number of replications r. It is inversely proportional to the maximum number of clock cycles spent in a pipeline block:

$$\phi = \begin{cases} \dfrac{1}{\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil} & \text{if } c = 1 \\ \dfrac{1}{\left\lceil \frac{n}{p} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{n}{p} \right\rceil c} & \text{if } c > 1 \end{cases} \tag{4.7}$$

Replicating and pipelining a design reduce the effective number of iterations that need to be computed in each block, leading to the following constraint:

$$p.r \leq n \tag{4.8}$$

4.5.2 Frequency

To simplify, we assume that the pre-processing and post-processing modules are not a frequency bottleneck. Hence the critical path is in a pipeline block and the minimum period is equal to the delay through a block. In our parametric description, the size of the main processing element in each block is around the same for all p. Even if the size of the control hardware managing the iterations in a block decreases with p, the delay through a block is not likely to decrease significantly. Therefore, we assume that the minimum period of the design does not depend on p. On the contrary, replication has a negative effect on the critical path as the replicated cells are connected together in series. The minimum period can therefore be approximated by:

$$T_p(r) \approx d_l + r d_c \tag{4.9}$$

where d_c is the delay through a cell, r is the number of cells in a block and d_l is the sum of the delays through the logic located before and after the cells (mostly multiplexers). Hence the frequency of the design is:

$$f(r) = \frac{1}{T_p(r)} = \frac{1}{d_l + r d_c} \tag{4.10}$$

Let us define:

$$f_0 = 1/T_p(1) \tag{4.11}$$

the frequency of the design for $r = 1$ and:

$$\lambda = d_c/T_p(1) \tag{4.12}$$

the ratio of the delay through the basic cell to the minimum period for $r = 1$. We can rewrite the frequency as:

$$f(r) = \frac{1/T_p(1)}{1 + d_c/T_p(1)(r-1)} = \frac{f_0}{1 + \lambda(r-1)} \tag{4.13}$$

4.5.3 Area

We decompose the area taken by our design into two parts:

- A_l the area taken by logic [1]

- A_r the area taken by registers

Let us first consider the area taken by the logic A_l. Let A_{cell} be the logic area of a cell for $r = 1$, $A_{mux,2}$ and $A_{mux,3}$ be the area taken respectively by a 2-to-1 and a 3-to-1 bit multiplexer. We recall that ρ_i is the bitwith of the output R_i and m is the number of such outputs. As shown in Fig. 4.2, the multiplexers required at the inputs of the **R** registers are either 2-input or 3-input multiplexers, depending on the need for an extra clock cycle in pipeline block i. Hence, the logic area of pipeline block i is:

$$A_{block}(i) = r.A_{cell} + \sum_{k=0}^{m-1} \rho_k.A_{mux}(i) \tag{4.14}$$

where:

$$A_{mux}(i) = \begin{cases} A_{mux,2} & \text{if } iter(i) \bmod r = 0 \\ A_{mux,3} & \text{if } iter(i) \bmod r \neq 0 \end{cases} \tag{4.15}$$

Each register of the top register triangle needs either a 2-input or a 3-input multiplexer (not shown on Fig. 4.1 and Fig. 4.2) depending on whether they are simple registers (to choose between the initialisation value or the value from the previous register) or circular shift registers (to choose between the initialisation value, the value from the previous register, or the shifted output). The part of the register triangle supplying inputs to pipeline block number i is of size $iter(i)$. We recall that k is the size of vector **A**. Hence the logic area of the register triangle is:

$$A_{triangle} = k \sum_{i=1}^{p} iter(i).((i-1).A_{mux,2} + A_{mux,3}) \tag{4.16}$$

[1] We do not explicitly consider coarse-grained blocks such as DSPs and BRAMs in this model as all our architectures are implemented on fine-grained logic for scalability issues explained in section 3.2.2. However the model could be easily extended to coarse-grained resources by taking into account the occupation of these resources individually.

Let A_{pre} and A_{post} be the fixed logic area taken respectively by the pre-computation and post-computation modules. We can deduce the total logic area:

$$A_l = A_{pre} + A_{triangle} + \sum_{i=1}^{p} A_{block}(i) + A_{post} \tag{4.17}$$

To calculate A_r we need to derive the total register size S. The total size of the registers between two pipeline blocks is:

$$S_d = \sum_{i=0}^{l-1} \beta_i + \sum_{i=0}^{m-1} \rho_i \tag{4.18}$$

The size of a register in the top triangle of registers is for pipeline block i:

$$S_p(i) = k.iter(i) \tag{4.19}$$

Let S_{cell} be the total size of the registers internal to a cell, S_{pre} and S_{post} be respectively the total size of the registers of the pre-computation and post-computation modules. The total register size is:

$$S = S_{pre} + p.(S_d + r.S_{cell}) + \sum_{i=1}^{p} i.S_p(i) + S_{post} \tag{4.20}$$

Finally, we have:

$$A_r = S.A_{r,e} \tag{4.21}$$

where $A_{r,e}$ is the area taken by a one-bit register.

4.5.4 Constraints

The following constraints are used:

- Maximum area available for logic $A_{l,max}$

- Maximum area available for registers $A_{r,max}$

- Minimum frequency f_{min} at which we want the design to run

4.5.5 Throughput optimisation

The problem of optimising the throughput of the design can be formulated as follows:

maximise:

$$\phi = \begin{cases} \dfrac{1}{\left\lceil \frac{\lceil \frac{n}{p} \rceil}{r} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{\lceil \frac{n}{p} \rceil}{r} \right\rceil} & \text{if } c = 1 \\ \dfrac{1}{\left\lceil \frac{n}{p} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{n}{p} \right\rceil c} & \text{if } c > 1 \end{cases}$$

such that: (4.22)

$$A_l \leq A_{l,max}$$

$$A_r \leq A_{r,max}$$

$$f \geq f_{min}$$

$$p.r \leq n$$

Given a search interval for r and p, this problem is solved easily by exhaustive search.

4.6 Optimising the design

We show how the parameters of our model can be determined and we present an optimisation process based on our model.

4.6.1 Determining the values of the parameters

Our model contains many parameters which affect the values of r and p maximising the throughput. Table 4.3 summarises how each parameter is determined. The number of A inputs (k), the number of B inputs (l), the number of outputs (m) and their respective bitwidths (n, β_0, ... β_{l-1}, ρ_0, ..., ρ_{m-1}), and the number of clock cycles c needed to complete an iteration, depend on the application. Hence, we can obtain them easily once the design of the hardware mod-

Table 4.3: Determination of the design parameters

Parameters	Determination
A inputs number (k), bitwidth (n)	application-dependent
B inputs number (l), bitwidths (β_0, ... β_{l-1})	application-dependent
R outputs number (m), bitwidths (ρ_0, ... ρ_{m-1})	application-dependent
Iteration clock cycles (c)	application-dependent
Minimum frequency f_{min}	design choice
Maximum logic area ($A_{l,max}$)	device-dependent
Maximum register area ($A_{r,max}$)	device-dependent
Basic cell area (A_{cell})	pre-synthesis
Basic cell registers (S_{cell})	pre-synthesis
Pre/post-computation area (A_{pre}, A_{post})	pre-synthesis
Pre/post-computation registers (S_{pre}, S_{post})	pre-synthesis
Area of a 1-bit register ($A_{r,e}$)	pre-synthesis
Multiplexers area ($A_{mux,2}$ and $A_{mux,3}$)	pre-syntheses
Frequency parameters (f_0, λ)	interpolation or direct

ule is fixed. The minimum frequency f_{min} at which our module runs is a design choice. The maximum area available for registers $A_{r,max}$ and for logic $A_{l,max}$ are device-dependent. For an FPGA, $A_{r,max}$ is the maximum number of registers and $A_{l,max}$ is the maximum number of LUTs available. For an ASIC, $A_{r,max}$ and $A_{l,max}$ can be grouped together to provide the maximum area available given in μm^2, in number of cells or in the technology-independent notion of a register bit equivalent (rbe) [MQF91].

The area of a basic cell A_{cell}, of the pre-computation and post-computation modules (A_{pre} and A_{post}) can be approximated by using a priori knowledge of the design or found by running a single synthesis operation for $r = 1$ and $p = 1$. This is also the case for the registers internal to a cell S_{cell} and the registers of the pre-computation and post-computation modules (S_{pre} and S_{post}). The area of a 1-bit register ($A_{r,e}$), 2-to-1 and 3-to-1 multiplexers ($A_{mux,2}$ and $A_{mux,3}$) are easily determined by running short synthesis operations.

The frequency parameters f_0 and λ can be obtained by two different means. The first solution is to run some synthesis operations in order to get the values of f for a small set of chosen r and find f_0 and λ by interpolation. The second solution is to run a single synthesis operation to find the value of f_0 and compute λ directly using equation 4.12, provided that we can estimate the delays through a pipeline block and a replicated cell. In our current implementation of the optimisation process these parameters need to be extracted by the designer.

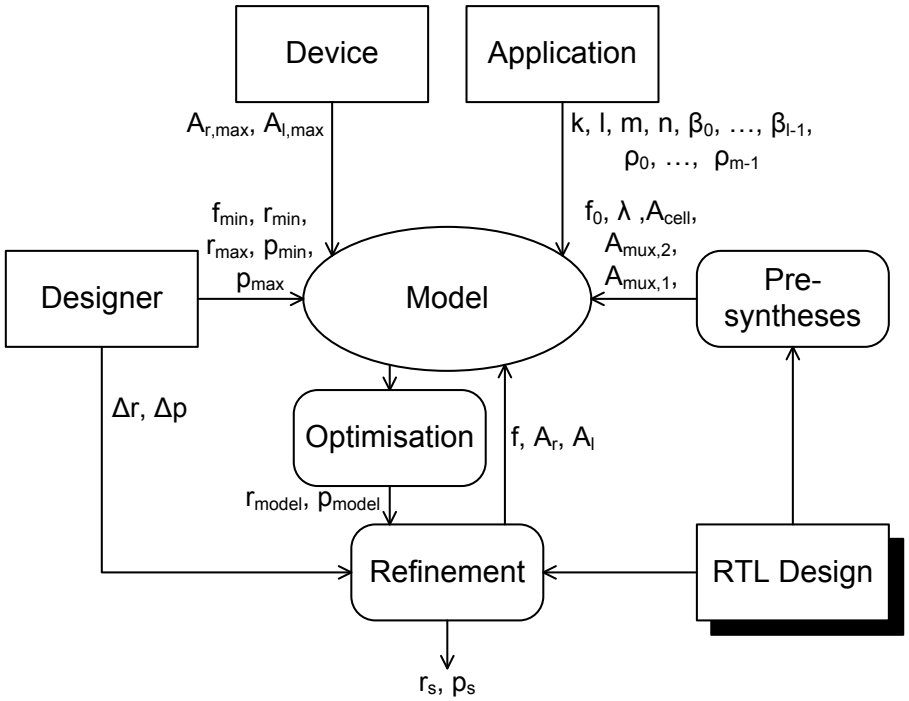


Figure 4.3: Optimisation process

If the search intervals for r and p are large enough, the time taken by the extra synthesis operations (we call them pre-synthesis operations) is negligible compared to the time it would take to synthesize the design for every possible value of the tuple (r, p) in the search intervals. More precisely, running 3 or 4 pre-synthesis operations for uniformly distributed values of r in the search interval is usually enough to get a good approximation of λ by interpolation. The pre-synthesis time needed to determine the multiplexers area is negligible. Moreover one of the pre-synthesis operations used to determine the frequency parameters also gives us the basic cell area A_{cell}. Hence for $r.p > 4$, using our model should be faster than a complete search through the design space. Speedup results are given in section 4.7.2.

4.6.2 Design generation and optimisation process

Given the different application-dependent parameters of the design and the number of pipeline stages p and replications r, a script automatically generates RTL code for our general pipelined and replicated hardware mapping as well as the corresponding control logic. This greatly simplifies the work of the designer. Only the basic cell, the pre-computation and post-computation logic are defined by the designer for each application. In the examples presented in section 4.7,

these modules are coded in Verilog. A higher level hardware language such as Handel-C or Xilinx HLS could be used to reduce the design time further.

The complete optimisation process of our parametric design is depicted in Fig. 4.3. The aim of this process is to find satisfactory values for the parameters r and p, that is values that are close to the real optima, in a short amount of time. First the different parameters of the model are determined using the methods described in Table 4.3. Given a search interval for r and p, respectively $[r_{min}, r_{max}]$ and $[p_{min}, p_{max}]$, the throughput optimisation of equation 4.22 gives us the optimal values of r and p according to the model. We call them r_{model} and p_{model}. The values r_{model} and p_{model} are then given to the `Refinement` process. This optional process performs a small number of synthesis operations around r_{model} and p_{model} to refine the solution by giving the real post-synthesis values of f, A_r and A_p to the model. The refinement intervals for r and p are respectively $[r_{model} - \Delta r, r_{model} + \Delta r]$ and $[p_{model} - \Delta p, p_{model} + \Delta p]$. The outputs of this process are r_s and p_s.

The `Pre-synthesis`, `Optimisation` and `Refinement` processes are implemented as a Python software tool which interfaces with Xilinx synthesis tools.

4.7 Accuracy and design space exploration results

In this section, we evaluate the accuracy and the benefits of our model against a complete search through the design space.

4.7.1 Accuracy of the model

We consider three applications presented in Table 4.2:

- 512-bit CSA-based Montgomery multiplication
- 128-bit modular exponentiation
- 512-bit integer square root

The hardware mappings of these designs are synthesized for a Xilinx Spartan 6 XC6SLX45T-FGG484-3 with XST 13.2 optimising for area, without resource sharing and equivalent register

removal. All the synthesis operations are run on an Intel Core i7 950 CPU @ 3.07GHz machine with 6GB of DDR3 memory. The application-dependent parameters for these three applications are summarized in Table 4.4.

The basic cells of the Montgomery multiplier and the integer square root modules only consist of combinational logic ($c = 1$). Hence replication can be introduced. We synthesize these two designs for all the combinations of the number of replications (r) and pipeline stages (p) in the respective intervals $[1, 16]$ and $[1, 16]$. The basic cell of the modular exponentiation module, which is basically a modular multiplier, is synchronous ($c > 1$). Therefore replicating this cell is irrelevant. We synthesize this design for $r = 1$ and p in $[1, 32]$. Table 4.5 shows the different design parameters obtained after the pre-synthesis stage. For the multiplier and the square root modules we determine the frequency parameters by interpolation using three points: $(p, r) = (1, 1)$, $(p, r) = (1, 8)$ and $(p, r) = (1, 16)$. The frequency of the exponentiation module is fixed to its value for $p = 1$.

For our three applications Table 4.6 reports the maximum, average and standard deviation of the relative prediction errors of our model across all the combinations of the parameters p and r for the number of look-up tables (LUTs), the number of registers, the maximum frequency and the throughput. The maximum errors for the number of LUTs and registers are very low over the three applications (1.18% to 4.43%, and 0.42% to 1.24% respectively). The maximum frequency error is more substantial for the Montgomery multiplication and the exponentiation module. We suspect that this error is due to uncontrollable optimisations performed by the synthesis tool on routing delays that sometimes make the frequency depend on the number of pipeline stages p. This dependency on p does not follow a general trend and therefore cannot be modelled effectively. For the Montgomery multiplier, this frequency error is only localised to a few (r, p) tuples. In fact the maximum frequency error is equal to 21% but the mean and standard variation of the frequency error are low (2-3%). This is not the case for the exponentiation module whose average frequency error is high and close to the maximum frequency error. In that case the reference point chosen to determine f_0 ($(r, p) = (1, 1)$) has a maximum frequency which is around 15% lower than for the other values of p, leading to a constant error.

Table 4.4: Application-dependent parameters for our three applications

	512-bit CSA Montgomery	128-bit exponentiation	512-bit integer square root
A inputs number (k)	1	1	2
A inputs bitwidth (n)	512	128	256
B inputs number (l)	2	2	0
B inputs bitwidths (β_0, ... β_{l-1})	512, 512	128, 128	-
R outputs number (m)	3	2	2
R outputs bitwidths (ρ_0, ... ρ_{m-1})	512, 512, 513	128, 128	512, 512
Iteration clock cycles (c)	1	314	1

Table 4.5: Design parameters obtained after pre-synthesis

	512-bit CSA Montgomery	128-bit exponentiation	512-bit integer square root
Basic cell area A_{cell} (LUTs)	1536	1343	774
Basic cell registers S_{cell}	0	1210	0
Pre-computation area A_{pre} (LUTs)	531	0	0
Post-computation area A_{post} (LUTs)	1580	0	0
Pre-computation registers S_{pre}	2584	0	0
Post-computation registers S_{post}	2087	0	0
Area of a 1-bit register $A_{r,e}$ (flip-flops)	1	1	1
Multiplexers area $A_{mux,2}$ and $A_{mux,3}$ (LUTs)	1	1	1
Initial frequency f_0 (MHz)	166.20	137.83	119.20
Stiffness of frequency decrease λ	0.66	0	0.83

Table 4.6: Accuracy of our model

	LUTs error (%)			Registers error (%)			Frequency error (%)			Throughput error (%)			Corrected throughput error (%)		
	Max	Mean	Std	Max	Mean	Std	Max	Mean	Std	Max	Mean	Std	Max	Mean	Std
CSA Montgomery	1.18	0.32	0.20	0.42	0.26	0.09	21.03	1.87	2.87	128.85	17.38	26.03	21.03	2.33	3.43
Exponentiation	4.43	3.32	0.68	1.23	1.19	0.02	15.73	14.75	2.96	15.73	14.75	2.96	15.73	14.75	2.96
Integer square root	3.41	0.60	0.66	1.24	0.97	0.12	0.68	0.34	0.14	0.68	0.34	0.14	0.68	0.34	0.14

Table 4.7: Evaluation of prediction capabilities of the model

	Design space size	Total synthesis time (min)	Before refinement			After refinement			
			Max throughput error (%)	Average synthesis time (min)	Average synthesis speedup (x)	Δr	Δp	Average synthesis time (min)	Average synthesis speedup (x)
CSA Montgomery	256	2489	25.48	7	419	2	2	33	96
Exponentiation	32	49	37.88	2	34	0	1	5	13
Integer square root	256	698	9.09	4	200	0	1	8	91

Table 4.8: Design constraints

f_{min} (MHz)	$(A_{l,max}, A_{r,max})$ (LUTs,FF)
0	$(6822, 13644)$
25	$(13644, 27288)$
50	$(27288, 54576)$
100	(∞, ∞)

The throughput error of the Montgomery multiplier is very high. Our model supposes that the latency of the pre-computation stage is shorter than the latency of any pipeline block. However, in this application the pre-computation stage performs a pipelined addition which takes 8 clock cycles. For r and p such that $\lceil \lceil n/p \rceil / r \rceil < 8$, the latencies of the pipeline blocks are less than 8 and the throughput of the module is fixed and only determined by the latency of the pre-computation stage. This invalidates equation 4.7. These values of r and p should be avoided as they lead to pointless area-consuming designs. The last column of Table 4.6 shows the corrected throughput errors, that is when not taking into account these designs. In that case, the throughput errors are similar to the frequency errors.

4.7.2 Design space exploration

We evaluate the prediction capabilities of our model. The Spartan 6 used has 27288 LUTs and 54576 flip-flops (FFs). For each of our three applications, we use our model to find the design with maximum throughput for all the combinations of f_{min} (in MHz) and $(A_{l,max}, A_{r,max})$ in Table 4.8. Hence for each application, we perform 16 different optimisations. Note that setting $f_{min} = 0$ MHz corresponds to not putting any constraint on the frequency. Setting $(A_{l,max}, A_{r,max}) = (\infty, \infty)$ corresponds to assuming that the FPGA has unlimited area.

Table 4.7 present our results. The design space size corresponds to the number of synthesis operations needed for a full search through the design space without using our method. The total synthesis time is the time taken to perform this search. The two other synthesis times reported are the average total synthesis time over the 16 optimisations performed using our model, respectively before and after refinement. Without refinement, our method can speed up the design space exploration by an average of 419 times for the CSA Montgomery application and 200 times for the integer square root. The lower average synthesis speedup of 34 times for

the exponentiation application is directly linked to the smaller size of the design space explored.

Without refinement the optimum design is not always found by our model. For the Montgomery multiplier module, our worst estimation is 2 pipeline stages and 2 replications off the optimum design. For the exponentiation and the integer square root modules, it is only 1 pipeline stage off. This leads to a throughput error (error between the optimum throughput and the throughput of the design given by the model) between 9% and 38%. In order to find the optimum design, extra syntheses need to be performed. This reduces the average synthesis speedups by a factor of 2 to 4. However, even with this speedup decrease using our method still saves hours of synthesis time. In particular the design space exploration time for the CSA Montgomery application is reduced from almost 2 days to only half an hour. Note that in practice we cannot know how many extra synthesis operations are needed to find the optimum design. Hence there is a trade-off between speed of the design space exploration and accuracy of the result.

4.8 Summary

In this chapter, we generalise our work on scalable Montgomery multiplication. We describe how we can map algorithms with the same structure as Montgomery multiplication to a parametric hardware design using pipelining and replication. A technology-independent model of our general design is developed. It allows rapid hardware mapping and optimisation of the algorithm by only running a few number of pre-synthesis operations and therefore reduces time-to-market. We present an optimisation method based on the model and we evaluate our method on three different applications implemented on a Xilinx Spartan 6 XC6SLX45T FPGA: the Montgomery multiplier, the modular exponentiatior and a integer square root module. Our model facilitates design space exploration by quickly predicting the area taken by the designs with less than 5% of error and their maximum frequencies and throughputs with less than 22% of error. As a matter of fact, our optimisation method is up to 96 times faster than a full search through the design space.

Chapter 5

Novel Architectures for Probabilistic Primality Testing

This chapters presents the first hardware architecture for the NIST approved Lucas primality test and a novel architecture for the Miller-Rabin primality test. Our architectures are based on the Montgomery multiplier and the Montgomery exponentiator described in chapter 3 and generated using the tools presented in chapter 4. After introducing probabilistic primality testing and our Miller-Rabin architecture briefly, we focus on the Lucas primality test. We present a hardware architecture to compute the Jacobi symbol based on the binary Jacobi algorithm. We analyse the dependencies in the Lucas sequences computation and propose a schedule for the different operations. Finally we compare the performance of our Miller-Rabin and Lucas architectures with an optimised software implementation in terms of speed, area and energy efficiency. Our Lucas architectures are implemented on a Xilinx Virtex-5 FPGA and synthesized for TSMC 65 nm and 45 nm ASICs.

5.1 Motivation

Many crypto-systems require the ability to test large numbers for primality. For instance, the key generation process of the RSA algorithm presented in section 2.3.4 needs two prime numbers of up to 1536 bits to allow secure data encryption beyond 2031 according to the RSA Laboratory recommendations [Lab04]. Algorithm 5.1 shows a typical method for finding large

primes. The main idea behind this algorithm is to test random numbers in the appropriate range until a prime number is found. In the neighbourhood of u, about one in $ln(u)$ numbers is prime. Hence choosing a limit $m = 100(\lfloor log_2 u \rfloor + 1)$ for the number of tests ensures that we will almost certainly find a prime [FSK10]. The $isPrime()$ function is our main interest in this chapter.

Algorithm 5.1: Generation of large prime numbers

Input: l lower bound of range in which prime should lie
u upper bound of range in which prime should lie
Output: p random prime in the interval $\{l, ..., u\}$ or -1 if no prime found

```
1 m = 100(⌊log2 u⌋ + 1)          /* Ensures a prime is almost certainly found */
2 for i = 0 to m − 1 do
3     Choose a random integer n in {l, ..., u}
4     if isPrime(n) then
5         return n
6 end
7 return −1
```

In [AKS02], the problem of finding whether a number is prime has been proven to be solvable in polynomial time using the AKS primality. This test is deterministic, meaning that it deterministically distinguishes whether the number is prime or composite. It is also unconditional, meaning that it does not rely on any unproven hypothesis. However, this test is complex and impractical to implement in hardware.

Probabilistic methods determine whether or not a number is prime with a certain probability of error. More precisely, all numbers declared composite[1] by these tests are not prime whereas a number declared prime can be composite with a small probability. By repeatedly running the test with different parameters, we can reduce this probability. Probabilistic primality tests are in use in most crypto-systems.

In a typical crypto-system, new keys are generated much less frequently than data are encrypted or signed. This is particularly true for public/private key pairs. Hence accelerating the key generation process is less important than accelerating encryption. However, the two main probabilistic tests recommended by NIST [ITLN09], namely the Miller-Rabin and the Lucas test, are mostly based on modular multiplication and modular exponentiation. The

[1] A composite number is a number that can be written as a product of more than one prime factor, that is a non-prime number.

Montgomery multiplication and exponentiation hardware already present for RSA encryption can therefore be reused.

5.2 Probabilistic primality testing

This section presents additional background on probabilistic primality testing, which is useful to understand the chapter.

5.2.1 Fermat primality test

The simplest probabilistic test is based on Fermat's little theorem which states that for a prime p and a positive integer a such that $gcd(p, a) = 1$:

$$a^{p-1} = 1 \bmod p \tag{5.1}$$

Using the contrapositive of this theorem, we can design a pseudo-primality test as shown in Algorithm 5.2.

Algorithm 5.2: Fermat primality test

Input: n odd integer, k parameter determining the accuracy of the test
Output: *composite* if n is composite, *prime* if n is probably prime

```
1 for i = 0 to k − 1 do
2     Choose a random integer a with 1 ≤ a ≤ n − 1
3     if a^{n−1} = 1 mod n then
4         continue
5     else
6         return composite
7 end
8 return prime
```

For a given a, a composite number n such that $gcd(n, a) = 1$ and $a^{n-1} = 1 \bmod n$ is called a base-a pseudoprime. By performing many tests with different random a we decrease the probability to declare a composite number prime. However, some numbers are pseudoprime to all bases [MD00]. Such numbers are called Carmichael numbers.

5.2.2 Miller-Rabin primality test

The Miller-Rabin strong pseudoprime test [Rab80] is an improvement over the Fermat test. This test is given in Algorithm 5.3. It relies on the fact that if we can find an integer a such that:

$$a^d \neq 1 \bmod n$$

and

$$a^{2^j d} \neq -1 \bmod n \text{ for all } 0 \leq j \leq l-1$$

then an odd integer n, written as $n = 2^l d + 1$, is composite.

Miller and Rabin have shown that if k tests are performed on an odd composite number, then the probability that this number passes each test is less or equal to $1/4^k$. When the strategy for finding primes presented in Algorithm 5.1 is used in conjunction with the Miller-Rabin test, bounds on the probability of returning a composite number are given in [DLP93, ITLN09].

Algorithm 5.3: Miller-Rabin strong pseudoprime test (from [Rab80])

Input: $n = 2^l d + 1$ odd integer, k parameter determining the accuracy of the test
Output: *composite* if n is composite, *prime* if n is probably prime

```
1 for i = 0 to k − 1 do
2     Choose a random integer a with 1 ≤ a ≤ n − 1
3     if a^d = 1 mod n or a^(2^j d) = −1 mod n for some 0 ≤ j ≤ l − 1 then
4         continue
5     else
6         return composite
7 end
8 return prime
```

Instead of using random numbers, the first k prime numbers can be used. This is shown in Algorithm 5.4. A scalable hardware implementation of this test is given in [CBLC04].

Algorithm 5.4: Miller-Rabin strong pseudoprime test deterministic variant

Input: $n = 2^l d + 1$ odd integer, set P of $|P|$ first primes
Output: *composite* if n is composite, *prime* if n is probably prime

```
1 for i = 0 to |P| − 1 do
2     a = P[i]                          /* Choose (i+1)th prime as the witness */
3     if a^d = 1 mod n or a^(2^j d) = −1 mod n for some 0 ≤ j ≤ l − 1 then
4         continue
5     else
6         return composite
7 end
8 return prime
```

5.2.3 Lucas primality test

Principle

The Lucas sequences $U(a, b)$ and $V(a, b)$ of the pair (a, b) are the sequences:

$$U(a,b) = (U_0(a,b), U_1(a,b), U_2(a,b), ...) \text{ and}$$
$$V(a,b) = (V_0(a,b), V_1(a,b), V_2(a,b), ...)$$

such that for each $k \geq 0$:

$$U_k(a,b) = \frac{\alpha^k - \beta^k}{\alpha - \beta} \tag{5.2}$$
$$V_k(a,b) = \alpha^k + \beta^k \tag{5.3}$$

where α and β are the two roots of the quadratic equation $x^2 - ax + b = 0$, with a, b chosen such that $a, b \neq 0$ and the discriminant $D = a^2 - 4b \neq 0$ [MD00].

For a integer and p prime, the Legendre symbol is defined as:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } a = 0 \bmod p \\ 1 & \text{if } a \neq 0 \bmod p \text{ and } x^2 = a \bmod p \text{ is soluble for some integer } x \\ -1 & \text{if } a \neq 0 \bmod p \text{ and } x^2 = a \bmod p \text{ is not soluble} \end{cases}$$

For any integer a and any positive odd integer n with prime decomposition $n = p_1^{\gamma_1} p_2^{\gamma_2} ... p_k^{\gamma_k}$, the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\gamma_1} \left(\frac{a}{p_2}\right)^{\gamma_2} ... \left(\frac{a}{p_k}\right)^{\gamma_k}$$

The Lucas theorem is defined as follows [MD00].

Lucas Theorem. Let a, b, D, and U_k be as above. If p is prime, $\gcd(b, p) = 1$ and $\left(\frac{D}{p}\right) = -1$, then p divides U_{p+1}.

The Lucas test is based on the contrapositive of the Lucas Theorem stating that if, under the previous assumptions, an odd positive integer n does not divide U_{n+1}, then n is composite.

Lucas Test algorithms

A simple algorithm performing a Lucas Test is given in Algorithm 5.5.

Algorithm 5.5: Simple Lucas test algorithm

Input: n odd integer
Output: *composite* if n is composite, *probably prime* if n is probably prime

```
1 Choose a, b ≠ 0 such that D = a^2 − 4b ≠ 0, gcd(b, n) = 1 and (D/n) = −1
2 Compute U_{n+1}(a, b)                          /* Directly or recursively */
3 if U_{n+1}(a, b) mod n = 0 then
4     return probably prime
5 else
6     return composite
```

Consider line 1 of Algorithm 5.5. Some methods to choose a, b and D are given in [BW80, PSJ80]. A method proposed by Selfridge [PSJ80] consists of choosing D as the first element in the sequence $\{5, -7, 9, -11, 13, ...\}$ such that $\left(\frac{D}{n}\right) = -1$, $a = 1$ and $b = \frac{1 - D}{4}$. If n is square, it can be shown that $\left(\frac{D}{n}\right) > -1$ for any D. Hence, so that the algorithm terminates, we have to check whether n is a perfect square[2] either before looking for a correct D or after a few trials

[2] A number n is a perfect if there exists a positive integer a such that $n = a^2$.

for D.

Consider line 2. Several methods can be used to compute U_{n+1}. A simple method is to use equation 5.2 directly. However, this method is not efficient for hardware implementation as it requires solving a quadratic equation and performing a big integer division. It is better to use a recurrence relation. For example, we can easily show that for $k > 1$:

$$U_k(a,b) = aU_{k-1} - bU_{k-2} \tag{5.4}$$

$$V_k(a,b) = aV_{k-1} - bV_{k-2} \tag{5.5}$$

We can also show that for $k > 0$:

$$U_k(a,b) = \frac{aU_{k-1} + V_{k-1}}{2} \tag{5.6}$$

$$V_k(a,b) = \frac{aV_{k-1} + DU_{k-1}}{2} \tag{5.7}$$

In [JQ96], a more complex recurrence relation is used to come up with an efficient method to compute the Lucas sequences:

$$U_{i+j}(a,b) = U_iV_j - b^jU_{i-j} \tag{5.8}$$

$$V_{i+j}(a,b) = V_iV_j - b^jV_{i-j} \tag{5.9}$$

A particular case of this recurrence relation useful for calculating U_k and V_k by processing k bit by bit is:

$$U_{2k}(a,b) = U_kV_k \tag{5.10}$$

$$V_{2k}(a,b) = V_k^2 - 2b^k = \frac{V_k^2 + DU_k^2}{2} \tag{5.11}$$

as:

$$b^k = \alpha^k\beta^k\frac{V_k + U_k\sqrt{D}}{2}\frac{V_k - U_k\sqrt{D}}{2} = \frac{V_k^2 - DU_k^2}{4} \tag{5.12}$$

This recurrence relation is used in the American National Institute of Standards and Technology (NIST) approved Lucas probabilistic primality test given in Algorithm 5.6.

Algorithm 5.6: Lucas probabilistic primality test (from [ITLN09])

Input: n odd integer
Output: *composite* if n is composite, *probably prime* if n is probably prime

1 **if** n *is a perfect square* **then**
2 **return** *composite*
3 Find the first D in the sequence $\{5, -7, 9, -11, 13, -15, 17, ...\}$ for which the Jacobi symbol $\left(\frac{D}{n}\right) = -1$.
4 **if** $\left(\frac{D}{n}\right) = 0$ *for any* D *in this sequence* **then**
5 **return** *composite*
6 $k = n + 1$
7 Let $k_l k_{l-1} ... k_0$ be the binary expansion of k, with $k_l = 1$
8 $U = 1$, $V = 1$ /* Initialisation */
9 **for** $i = l - 1$ **to** 0 **do**
10 $U_{temp} = UV \bmod n$ /* Equation 5.10 */
11 $V_{temp} = (V^2 + DU^2)/2 \bmod n$ /* Equation 5.11 */
12 **if** $k_i = 1$ **then**
13 $U = (U_{temp} + V_{temp})/2 \bmod n$ /* Equation 5.6 */
14 $V = (V_{temp} + DU_{temp})/2 \bmod n$ /* Equation 5.7 */
15 **else**
16 $U = U_{temp}$
17 $V = V_{temp}$
18 **end**
19 **if** $U = 0$ **then**
20 **return** *probably prime*
21 **else**
22 **return** *composite*

The NIST algorithm first checks if n is a perfect square. If not, it uses the method described by Selfridge to find a correct value for D. This test is sometimes called the Lucas-Selfridge probabilistic primality test. Lines 4-5 rely on the fact that if $\left(\frac{D}{n}\right) = 0$, a factor of n exists [BW80] and we can therefore stop the test. Lines 6 to 18 computes $U_{n+1}(a, b)$ using an algorithm similar to left-to-right binary exponentiation. It starts with $U = U_1(a, b) = 1$ and $V = V_1(a, b) = 1$. Let us call U_{K_i} and V_{K_i} the terms of the Lucas sequences held respectively in U and V at the beginning of iteration i. At iteration i, if bit $k_i = 0$, the current values of U and V are updated through equations 5.10 and 5.11. Hence at the end of iteration i, we have $U = U_{K_{i-1}} = U_{2K_i}$ and $V = V_{K_{i-1}} = V_{2K_i}$ This is similar to the squaring operation in binary exponentiation. If bit $k_i = 1$, U and V are first updated through equations 5.10 and 5.11 and then through equations

5.6 and 5.7. Hence at the end of iteration i, we have $U = U_{K_{i-1}} = U_{2K_i+1}$ and $V = V_{K_{i-1}} = V_{2K_i+1}$. This is similar to the square-and-multiply operation in binary exponentiation. We deduce the following recurrence relation:

$$K_l = k_l = 1 \tag{5.13}$$

$$K_i = 2K_{i+1} + k_{i+1} \quad \forall i < l \tag{5.14}$$

from which we obtain:

$$K_0 = 2(2...(2k_l + k_{l-1}) + k_{l-2}...) + k_0 \tag{5.15}$$

$$= 2^l k_l + 2^{l-1} k_{l-1} + 2^{l-2} k_{l-2} + ... + k_0 = k \tag{5.16}$$

In [ITLN09], the NIST gives an algorithm to determine if a number is a perfect square. This algorithm is complex for hardware implementation as it requires divisions and squaring. A much simpler binary algorithm is presented in [Cre]. We give the binary integer square root algorithm in Algorithm 5.7. This algorithm returns the integer square root of an n-bit number together with the remainder using only shifts, additions and subtractions. At each iteration, a digit of the root is determined using two consecutive digits of the input a (from the MSB to the LSB) and the remainder is updated accordingly. The number tested is a perfect square if the remainder is equal to zero after $n/2$ iterations.

Line 3 of Algorithm 5.6 requires the ability to compute the Jacobi symbol $\left(\frac{D}{n}\right)$. As before, the algorithm given by the NIST in [ITLN09] is not fit for fast hardware implementation as it requires several modular reductions. Some more relevant binary Jacobi algorithms using only shifts, additions/subtractions and comparisons are presented in [SS93, PPV06, ES98, Ved06].

No FPGA implementation of the Lucas primality test has been reported in the literature so far.

5.2.4 Combining tests

A primality test can be made stronger by combining two or more probabilistic methods. The Baillie-PSW Primality Test consists of a single Miller-Rabin test with base 2 followed by a

Algorithm 5.7: Binary integer square root algorithm (from [Cre])

Input: $a = \sum_{i=0}^{n-1} a_i 2^i$, $a_i \in \{0, 1\}$
Output: *rem* (remainder), *root* (square root)

```
1  rem = 0
2  root = 0
3  divisor = 0
4  for i = 0 to n/2 − 1 do
5      root = (root << 1)
6      rem = (rem << 2) + (a >> (n − 2))
7      a = (a << 2)
8      divisor = (root << 1) + 1
9      if divisor ≤ rem then
10         rem = rem − divisor
11         root = root + 1
12 end
```

Lucas test [PSJ80]. We could not find any mention of a composite number passing this test in the literature. Moreover in [ITLN09], the authors claim that there is no known composite number passing the number of Miller-Rabin tests with random bases prescribed by the NIST[3] followed by a single Lucas test.

5.3 Miller-Rabin primality test architecture

Our parametric Miller-Rabin prime tester is based on both our multiplier and our exponentiator. The challenge is to use the multiplier and the exponentiator optimally, keeping the design relatively simple. Our hardware design is based on Algorithm 5.4 but could easily be adapted to Algorithm 5.3.

To save area, one single multiplier is shared by the exponentiator (to perform the multiplications needed to compute $a^d \bmod n$) and the prime tester (to perform the consecutive modular multiplications needed to compute the $a^{2^j d} \bmod n$). In a scenario where random numbers are given to the module to be tested for primality, on average the time to compute $a^d \bmod n$ largely dominates the calculation of the $a^{2^j d} \bmod n$. To demonstrate this property, let Y be the random variable representing the number of trailing 0 in a uniformly distributed random

[3]This number depends on the bitwidth of the key to generate.

even variable X of size k. Let $X[i]$ represent the i^{th} bit of X. We have:

$$p(Y = 0) = 0 \tag{5.17}$$

$$p(Y = 1) = p(Y \geq 1) \cap p(X[1] = 1) = 1.\frac{1}{2} = \frac{1}{2} \tag{5.18}$$

$$p(Y = i) = p(Y \geq i) \cap p(X[i+1] = 1) = \left(\frac{1}{2}\right)^{i-1}.\frac{1}{2} = \left(\frac{1}{2}\right)^{i} \quad \forall i \geq 2 \tag{5.19}$$

Hence the expected value of Y is:

$$E(Y) = \sum_{i=0}^{k-1} i.p(Y = i) = \sum_{i=0}^{k-1} i.\left(\frac{1}{2}\right)^{i} = \frac{1}{2}\sum_{i=1}^{k-1} i.\left(\frac{1}{2}\right)^{i-1} \tag{5.20}$$

This series is strictly monotonically increasing and converges towards 2. Hence the value of l in Algorithm 5.4 is less than 2 on average. This shows that on average the multiplier is used less than once directly by the prime tester at each iteration.

A diagram containing the important blocks, signals and connections of our prime tester is presented in Fig. 5.1. The values of the first prime numbers are stored in a ROM. At each iteration, the control logic selects the prime to use as a witness for the test and the inputs to give to the multiplier, to the comparator and to other intermediate registers. The control logic contains the state machine of the prime tester which controls the multiplier and the exponentiator.

5.4 Lucas primality test

Our Lucas Primality tester is based on Algorithm 5.6 recommended by the NIST. We divide our Lucas hardware into three sub-modules:

1. Perfect square test module (lines 1-2)
2. Jacobi symbol calculator module (lines 3-5)
3. Lucas sequence (or $U_{n+1}(a, b)$) calculator module (lines 6-22)

Module 1 is an implementation of Algorithm 5.7. In our implementation, the iterations of the loop are performed by the square root cell. The architecture of the square root cell is given

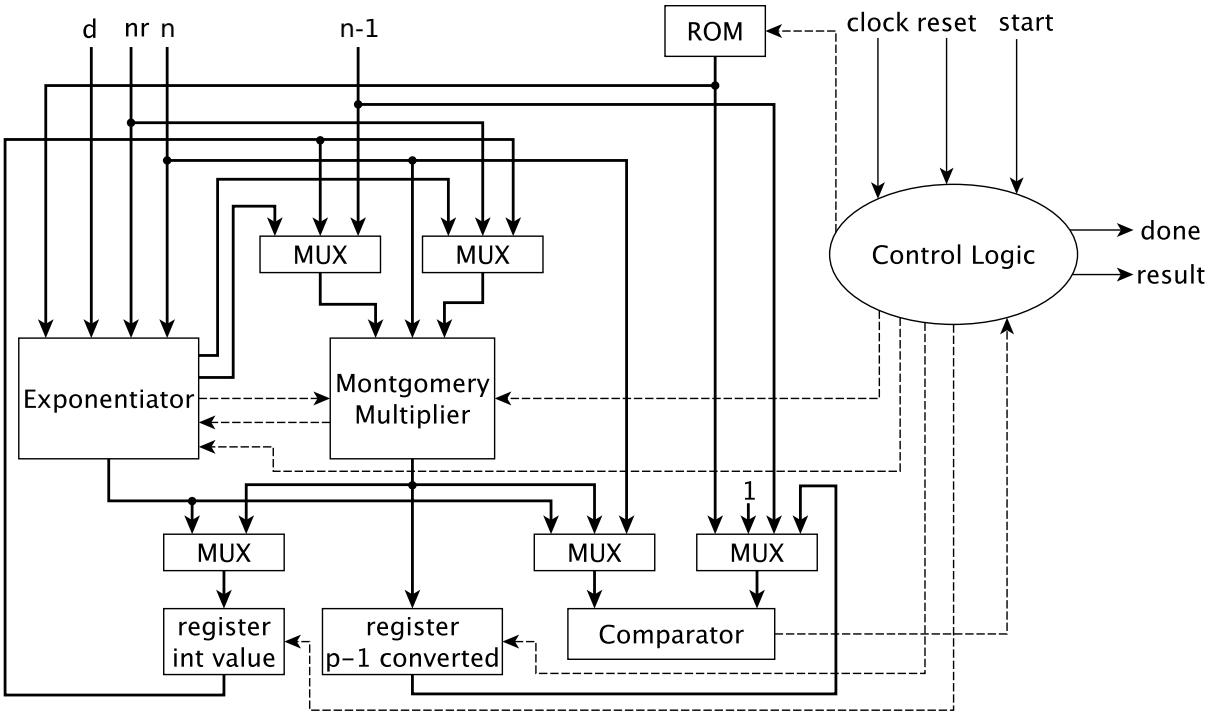


Figure 5.1: Miller-Rabin prime tester

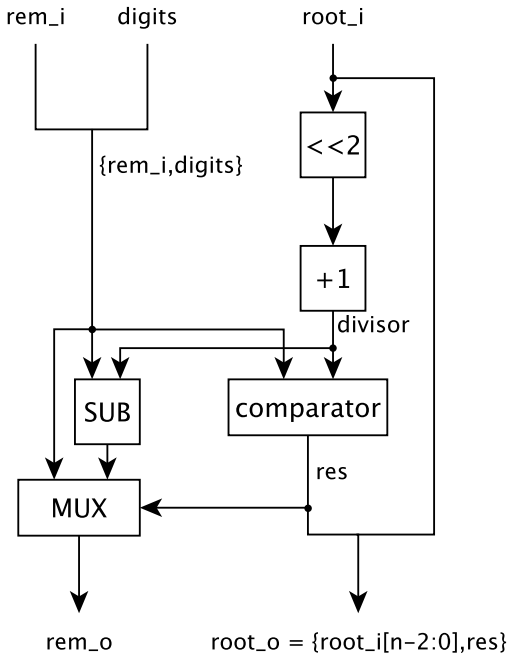


Figure 5.2: Square root cell

in Fig. 5.2. The values of *rem* and *root* are stored in registers. a is stored in a shift register and shifted by 2 at each iteration. The diagram of the square root cell is given in Fig. 5.2.

In the following, we focus on module 2 and 3 which are the most interesting ones. Three main challenges have to be addressed when designing these modules:

Challenge 1. The Jacobi symbol calculator module has to work with negative integers and must not contain any complex modular reduction that would greatly increase the area taken by the design.

Challenge 2. We need to design parametric and efficient modules for the two main operations of the Lucas calculator: (2.a) modular multiplication and (2.b) modular add-shift.

Challenge 3. These two operations have to be rescheduled to optimise the speed of the design, making the most of the pipeline capabilities of the modules.

5.4.1 Challenge 1: Jacobi symbol calculator

Classical Jacobi algorithms require a full modular reduction at each iteration. This operation can only be implemented with a divider which is area consuming. This is not the case for the binary Jacobi algorithm presented in [SS93]. We modify this algorithm to compute $\left(\frac{a}{b}\right)$ for negative a by using the following property of the Jacobi symbol:

$$\left(\frac{-a}{b}\right) = \begin{cases} \left(\frac{a}{b}\right) & \text{if } b \bmod 4 = 1 \\ -\left(\frac{a}{b}\right) & \text{if } b \bmod 4 = 3 \end{cases} \tag{5.21}$$

Our modified binary Jacobi algorithm is presented in Algorithm 5.8.

All the modular reductions use a fixed modulo which is a power of two. In that case, $x \bmod y = (x \;\&\; (y-1))$. These operations are therefore very simple to implement in hardware. To reduce the critical path and therefore increase the maximum clock frequency, the comparators and the subtractors are pipelined.

5.4.2 Challenge 2: Lucas sequence calculator

Our Lucas sequence calculator performs the operations of lines 6 to 22 of Algorithm 5.6. This part of the algorithm consists of several modular multiplications and modular add-shift operations. The operations performed in the for loop can be decomposed as shown in Algorithm 5.9.

Algorithm 5.8: Binary Jacobi algorithm

Input: a integer, b odd positive integer
Output: $\left(\frac{a}{b}\right)$

```
1  t = 1
   // Support for negative a
2  if a < 0 then
3      a = −a
4      if b mod 4 = 3 then t = −t
5  end
   // Original algorithm from [SS93]
6  while a ≠ 0 do
7      while a mod 2 = 0 do
8          a = a/2
9          if (b mod 8 = 3) or (b mod 8 = 5) then t = −t
10     end
11     if a < b then
12         interchange(a, b)
13         if (a mod 4 = 3) and (b mod 4 = 3) then t = −t
14     end
15     a = (a − b)/2
16     if (b mod 8 = 3) or (b mod 8 = 5) then t = −t
17 end
18 if b = 1 then
19     return t
20 else
21     return 0
```

Algorithm 5.9: Decomposition of the operations performed in the for loop of the Lucas primality test algorithm

```
1  U_temp = U.V mod n
2  V_temp = V^2 mod n
3  temp1 = U^2 mod n
4  temp1 = D.temp1 mod n
5  V_temp = (V_temp + temp1)/2 mod n
6  if k_i = 1 then
7      U = (U_temp + V_temp)/2 mod n
8      temp2 = D.U_temp mod n
9      V = (V_temp + temp2)/2 mod n
10 else
11     U = U_temp
12     V = V_temp
```

If k is an (l+1)-bit number with $k_l = 1$, the algorithm performs between $4l$ and $5l$ modular multiplications. We reuse our parametric Montgomery multiplier to perform these modular

multiplications and solve challenge 2.a. All the operations are performed in n-residue representation, with n the number under test.

Algorithm 5.10: Modular addition with right shift

Input: N odd integer, $A < N$ and $B < N$ positive integers
Output: $S = (A + B)/2 \bmod N$

1 **if** $(A + B)$ *mod* $2 = 1$ **then**
2 $S = (A + B + N)/2$ /* N added to make sum even */
3 **if** $S \geq N$ **then** $S = S - N$ /* The sum can now be greater than N */
4 **else**
5 $S = (A + B)/2$

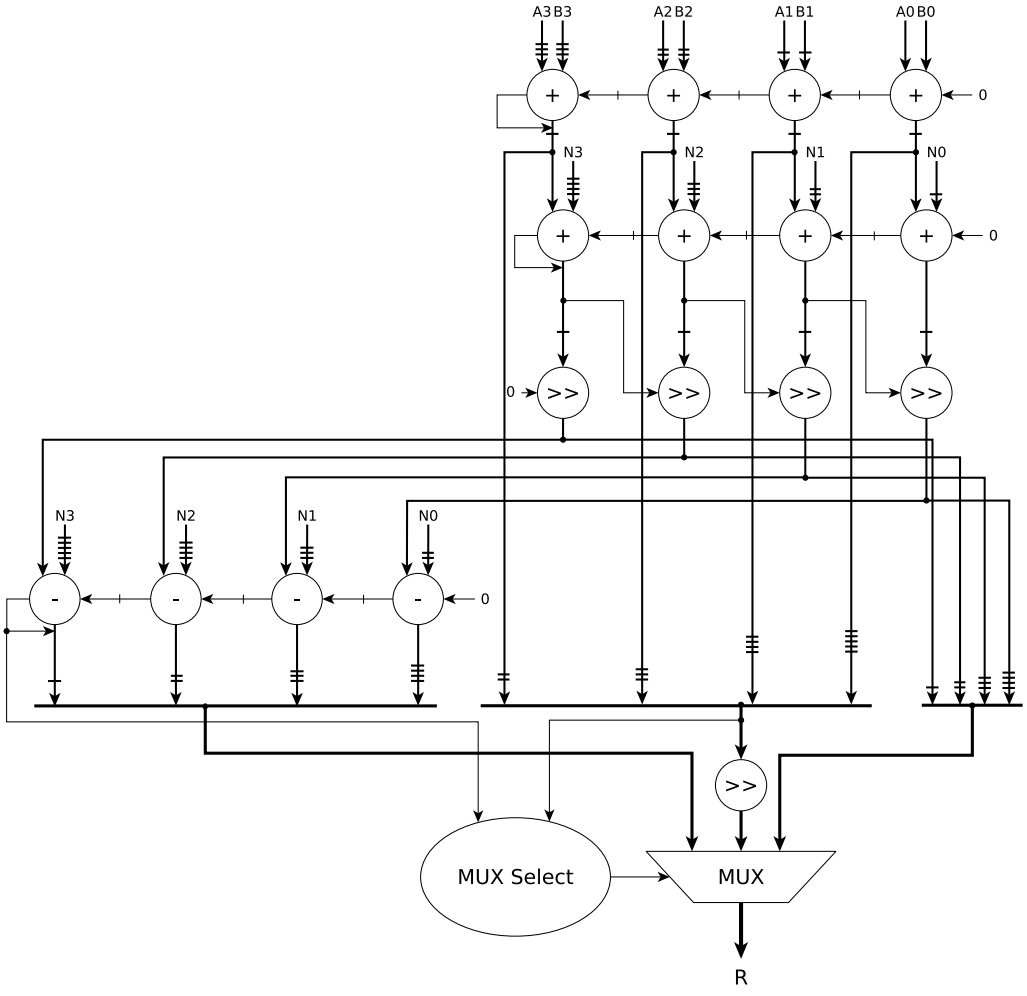


Figure 5.3: Hardware design for modular add-shift

Similarly, the algorithm performs between l and $3l$ modular add-shift operations. Algorithm 5.10 shows how the modular addition with right shift is implemented. Note that the values of the two operands A and B of this addition are always less than the modulo. Hence the modular reduction can be performed by a simple subtraction.

We develop a pipelined hardware design for the modular add-shift module. All the operations are computed by homogeneous blocks, the number of blocks being a parameter of the design. Our architecture is shown in Fig. 5.3. Short horizontal and vertical lines represent registers. The first row of adders computes $A + B$. Its output is given to the second row of adders which computes $A + B + N$ and to a shifter which computes $(A + B)/2$. The output of the second row of adders is shifted right and given to a row of subtractors. These subtractors compute $(A + B + N)/2 - N$. They are also used to determine if $(A + B + N)/2 \geq N$. The `MUX Select` generates the select signal of the multiplexer according to the borrow out of the row of subtractors and the value of the LSB of $A + B$. If m is the number of adders/subtractors in each row, after $m + 2$ cycles of latency our module can generate one result per clock cycle. This solves challenge 2.b.

Our Lucas calculator has four main parameters: the bit-width of the number under test, the number of pipeline stages of the Montgomery multiplier, the number of replicated carry-save adders in each multiplier's pipeline block and the pipeline depth of the modular add-shift module. By combining Montgomery multiplication and our pipelined modular add-shift architecture, we get rid of the need for direct modular reduction. A proper scheduler needs to be designed to make the most of our parametric modules.

5.4.3 Challenge 3: scheduling

To increase the speed of Algorithm 5.9 we reschedule the modular multiplications and the modular additions. In particular we want to minimise stalls in the Montgomery multiplier's pipeline. To keep the design simple and reduce its vulnerability to side-channel attacks we decide to execute the calculations of $(U_{temp} + V_{temp})/2 \bmod n$ and $(V_{temp} + temp2)/2 \bmod n$ even if $k_i = 0$. In that case, the result is disregarded. Hence $temp2$ always needs to be calculated and we always perform 5 Montgomery multiplications and 3 modular additions at each iteration. The true dependencies between the different operations are shown in the dependency graph of Fig. 5.4. Output and anti-dependencies have already been removed by register renaming.

The latency and the throughput of the Montgomery multiplier depend on the bit-width of the inputs, the number of pipeline stages and the number of replications. Using a pipeline

Instructions
1. $U_{temp} = U.V \mod n$
2. $V_{temp} = V^2 \mod n$
3. $temp1 = U^2 \mod n$
4. $temp1 = D.temp1 \mod n$
5. $V_{temp} = (V_{temp} + temp1)/2 \mod n$
6. $temp2 = D.U_{temp} \mod n$
7. $U = \begin{cases} U_{temp} & \text{if } k_i = 0 \\ (U_{temp} + V_{temp})/2 \mod n & \text{if } k_i = 1 \end{cases}$
8. $V = \begin{cases} V_{temp} & \text{if } k_i = 0 \\ (V_{temp} + temp2)/2 \mod n & \text{if } k_i = 1 \end{cases}$

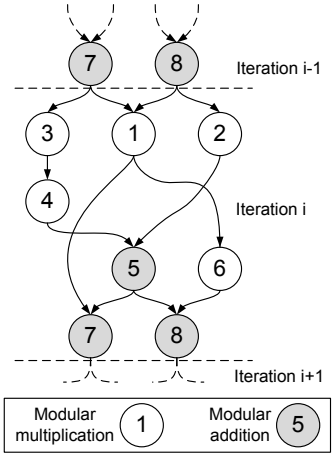


Figure 5.4: Analysis of the true dependencies in the Lucas sequence calculation algorithm

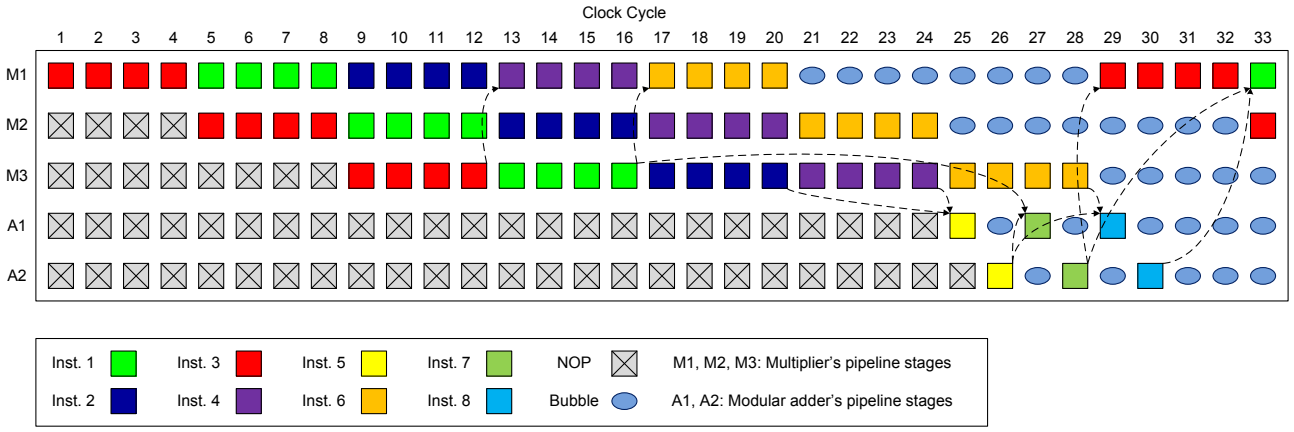


Figure 5.5: Lucas sequence calculator schedule for a 3-stage multiplier's pipeline

depth of more than 3 for the multiplier does not improve the speed of the Lucas sequence calculation given the dependencies shown in the dependency graph of Fig. 5.4. The latency of the modular add-shift module depends on the number of adders/subtractors blocks used in each row. Sticking to the instruction numbering introduced in the table of Fig. 5.4, we use the following schedule: Inst.3, Inst.1, Inst.2, Inst.4, Inst.6, Inst.5, Inst.7, Inst.8. Fig. 5.5 shows this schedule for a multiplier pipeline depth $p = 3$. To keep the example simple, we assume that each multiplier's pipeline stage takes 4 clock cycles to complete its operations. We also assume that the latency of the modular adder is 2 clock cycles. A1 and A2 represent the 2 pipeline stages of our modular adder. M1 to M3 represent the 3 pipeline stages of the multiplier. In the actual implementations, the latency of a multiplier's pipeline stage is much higher than the latency of the modular adder (42 and 10 respectively for our fastest implementation presented in section 5.5) and the number of pipeline stages of the module adder is chosen to maximise

the clock frequency of the implementation. However, this simplification does not change the interpretation of the system behaviour. From clock cycle 1 to clock cycle 20, the multiplier performs 5 multiplications without stalling. At clock cycle 21, no more multiplication can be given to the multiplier until Inst.7 terminates. Hence the multiplier stalls for 8 clock cycles. Once Inst.7 terminates the multiplier's pipeline can be filled again. This situation shows that the multiplier's pipeline cannot be run full all the time, which leads to some unavoidable performance loss.

Our particular schedule may not be optimal for all the values of the parameters but this schedule is a relevant compromise under the assumption that the addition time is shorter than the multiplication time.

The main components of our Lucas calculator module are the Montgomery multiplier, the modular add-shift module and several registers to store the values of U_{temp}, V_{temp}, $temp1$, $temp2$, D, U, V and k. We also need an adder to compute $k = n + 1$. A leading zeros detector is used to identify the most significant bit in the binary representation of k. Multiplexers select the inputs to the Montgomery multiplier and the modular add-shift module. FIFOs and dependency tables record the state of the Montgomery multiplier's pipeline and of the modular add-shift module. This enables the scheduler to prevent read-after-write dependencies and to feed the Montgomery multiplier and the modular add-shift pipelines correctly.

5.4.4 Lucas prime tester

The square test module, the Jacobi symbol calculator module and the Lucas calculator are integrated into our Lucas prime tester. The square test and the search for the value of D are performed in parallel. If the number tested is a perfect square or if a D such that $\left(\frac{D}{n}\right) = 0$ is found, the test finishes and returns 0 (the number is declared composite). Otherwise, the Lucas sequence calculator is started as soon as a correct value for D is found. Then a comparator tests if $U_{n+1}(a, b) = 0$. The test finishes and returns 1 if $U_{n+1}(a, b) = 0$ (the number is declared probably prime) or 0 if $U_{n+1}(a, b) \neq 0$ (the number is declared composite).

5.5 Speed/Area/Power trade-off of FPGA and ASIC implementations

5.5.1 FPGA implementation of our Miller-Rabin architecture

We place and route our designs with Xilinx ISE 14.1 for Xilinx Virtex-5 FPGAs in performance evaluation mode. The implementation parameters are given in appendix D.1.1. Table 5.1 compares the execution time of our Miller-Rabin prime tester with other implementations for $n = 1024$ bits. The pipeline depth of all ripple-carry adders and all subtractors is set to 8. The execution time of the prime tester depends on the number under test. Hence we choose a test set of 10 000 random numbers and use them for all our experiments. We report the mean and the standard deviation of the execution time. For $p = 1$ and $r = 1$, our prime tester is 7 times faster than Cheung's non scalable design and takes 1.8 times less area. It is 100 times faster than the fastest scalable design from [CBLC04] with only 20% area overhead. For $p = 2$ and $r = 8$, our design running at 73.7 MHz is 1.6 times faster than the GMP implementation on one core of an Intel Core 2 Duo running at 2.8 GHz. It is 49 times faster than Cheung's non scalable design and only takes 2 times more area. However, one should take into account that the Virtex-II FPGA used in Cheung's work is several generations behind our high-end Virtex-5 FPGA. In order to make the results independent of the speed of the devices, Table 5.2 compares the designs in terms of Time $\times$ Area product. These results show that our architecture is up to 2.9 times more efficient than Cheung's best design.

Our prime tester implementation is limited by data dependencies and cannot explore the full design space provided by the multiplier and the exponentiator presented in chapter 3. In fact, only one exponentiation is performed at a time. Unless we run several iterations of the tests speculatively[4], we cannot interleave exponentiations in the multiplier's pipeline. Hence using more than 2 pipeline stages for the multiplier does not improve the exponentiation speed. We can always increase the speed of the prime tester by increasing the number of replications r

[4]Even if interleaved exponentiation is supported by our exponentiator, this would require extra control and arbitration logic. As a matter of fact, each Miller-Rabin test does not only perform one exponentiation but also a number of multiplications depending on the number under test. Such a feature could be implemented as future work.

Table 5.1: Performance comparison of 1024 bit Miller-Rabin

Design	Device	Clock (MHz)	Area (LUTs)	Ex. Time mean/σ (ms)
Ours ($p = 2$, $r = 8$)	XC5VLX330T-2	73.7	74 779	2.64/0.97
Ours ($p = 2$, $r = 4$)	XC5VLX110T-3	93.4	46 071	3.51/1.30
GMP 4.2.4 [gmp] `mpz_millerrabin`	Intel Core 2 Duo E7400	2800	N/A	4.33/1.82
Ours ($p = 2$, $r = 2$)	XC5VLX110T-3	123.1	34 199	4.85/1.79
Ours ($p = 1$, $r = 4$)	XC5VLX110T-3	87.4	32 593	6.21/2.29
Ours ($p = 1$, $r = 1$)	XC5VLX110T-3	123.6	22 169	17.4/6.42
Cheung [CBLC04] (non-scalable)	XC2V6000	8.2	40 262	129.28/_
Cheung [CBLC04] (scalable 32 PE)	XC2V6000	32.2	18 666	1776.80/_
Cheung [CBLC04] (scalable 8 PE)	XC2V6000	30.7	5 744	5478.14/_

Table 5.2: Time × Area product normalised to our best design for 1024 bit Miller-Rabin test

Design	Area (LUTs)	Latency (clock cycles)	Time × Area
Ours ($p = 2$, $r = 8$)	74 779	194207	**1**
Ours ($p = 2$, $r = 4$)	46 071	328590	1.04
Ours ($p = 1$, $r = 4$)	32 593	542894	1.22
Ours ($p = 2$, $r = 2$)	34 199	597358	1.41
Cheung [CBLC04] (non-scalable)	40 262	1056727	2.93
Ours ($p = 1$, $r = 1$)	22 169	2151984	3.29
Cheung [CBLC04] (scalable 8 PE)	5 744	167938074	66.42
Cheung [CBLC04] (scalable 32 PE)	18 666	57260715	73.60

up to the point where the path through the carry-save adders becomes the critical path of the design. Moreover, as will be shown for the Lucas in the next section, a hardware implementation of the test is likely to be more energy efficient than its software counterpart.

5.5.2 FPGA and ASIC implementations of our Lucas architecture

FPGA results

We place and route our designs with Xilinx ISE 14.1 on a Virtex-5 XC5VLX330T-2 FPGA in performance evaluation mode. Our hardware designs are compared with an optimised software implementation of Algorithm 5.6 using dedicated functions of the GMP library 5.0.1 for all the operations and compiled with the Intel Compiler 12.0. The targeted architecture for our software implementation is an Intel Xeon W3505 @ 2.53 GHz with 3 GB of main memory. Only one core of the Xeon processor is used. We report the average execution time, the

dynamic power consumption and the average energy for both the hardware and the software implementations. For the average execution time, a random set of 10 000 1024-bit values is used as input. For the maximum execution time, we choose an input in our random set that goes through all the steps of Algorithm 5.6, requiring a full calculation of the Lucas sequence[5]. The input value is given in appendix D.3. The dynamic power and energy consumption are reported for this input. For the software version, the dynamic power is measured with a watt-metre at the output of the system. The dynamic power of the FPGA is estimated using XPower Analyzer 14.1. The toggle rates of the different nets are obtained by post place-and-route simulation.

The results are reported in Table 5.3. The parameters p and r represent respectively the number of pipeline stages of our Montgomery multiplier and the number of replicated carry-save adders in each pipeline block.

Table 5.3: FPGA implementation results of 1024 bit Lucas primality testers

Type	Area (LUTs)	Clock (MHz)	Average Ex. Time (ms)	Max. Ex. Time (ms)	Dynamic Power (mW)	Energy (mJ/op)
Hardware ($p = 1$, $r = 1$)	34 598	72.7	48.66	72.42	446.56	32.34
Hardware ($p = 1$, $r = 2$)	37 676	79.1	22.59	33.47	587.45	19.66
Hardware ($p = 2$, $r = 1$)	38 720	74.4	26.55	39.46	560.46	22.12
Hardware ($p = 2$, $r = 2$)	44 869	75.5	14.56	21.54	840.16	18.10
Hardware ($p = 3$, $r = 1$)	43 188	76.6	23.13	34.11	681.11	23.23
Hardware ($p = 3$, $r = 2$)	54 458	85.0	12.18	18.44	1191.61	21.97
Hardware ($p = 3$, $r = 4$)	75 455	92.8	7.65	11.17	1807.33	20.19
Hardware ($p = 3$, $r = 8$)	107 348	71.6	7.46	10.80	2357.36	25.46
Software	N/A	2530	2.40	3.30	28 000	92.40

As expected, the execution time decreases with the number of pipeline stages p and the

[5]Note that this input gives a number of clock cycles very close to the worst-case behaviour for the hardware implementations, for which the number of clock cycles to compute the Lucas sequence is fixed. The only variation is in the search for a correct value for D. This is negligible compared to the Lucas sequence calculation. However this might not be the worst-case input for the software version as the execution time of the Lucas sequence calculation in software depends on the numbers of ones in the binary representation of $n + 1$.

number of replications r while the dynamic power consumption increases with r and p as the area and switching activity increase. The area scaling is much better than linear with r and p as the area contribution of other fixed size modules such as the Jacobi symbol calculator or the modular add-shift modules cannot be neglected. For the same reason the achievable clock frequency is not directly linked to the number of replications r as the critical path is not always in the Montgomery multiplier. It is interesting to note that bigger designs sometimes achieve better clock frequencies than smaller ones. This behaviour is linked to the fact that we run the place-and-route tool in performance evaluation mode. Hence we might not reach the best possible timing closure for all the implementations.

Our fastest design is 3 times slower but 3 times more energy efficient than the software version. This design takes around 50% total area available in the Virtex-5 LX330T. Our most energy-efficient design is 5 times more energy-efficient than the software version. Most of the time is spent in Montgomery multiplications. Two main factors can explain the excellent speed of the software. First, unlike the hardware implementation, the software implementation does not always execute lines 13 and 14 of Algorithm 5.6. For random 1024-bit inputs, this saves 512 Montgomery multiplications and 1024 modular additions on average in the Lucas sequence calculation. Second, the particular structure of our Montgomery multiplier makes it much faster than a software implementation if inputs can be provided at a high throughput without stalling the pipeline. In this application, the data dependencies in the algorithm limit the maximum value for the number of pipeline stages p and therefore the achievable speed of our multiplier. Like the Miller-Rabin test, the results could be improved further by increasing the number of replications r if a bigger FPGA is available. Upcoming FPGAs such as Virtex 7 will be several times bigger than this Virtex-5 allowing much better performance while still having unused area. We also expect the energy efficiency of our FPGA designs will improve with advances in FPGA technology. It should be noted that the Virtex-5 FPGA (65 nm) is one generation behind the Xeon W3505 CPU (45 nm).

Server applications would benefit from augmenting the CPU with one of more FPGAs configured with this design. In these applications, power consumption can turn out to be as important as performance. As a matter of fact, improving the energy efficiency of the system

can lead to huge savings in fixed costs (cooling equipment) and variable costs (energy bill).

ASIC results

We synthesize our design for the TSMC 65 nm *tcbn65gplustc* library using Synopsys Design Compiler version G-2012.06-SP2. Our two fastest designs in 65 nm are also synthesized for the TSMC 45 nm *tcbn45gsbwptc* library to allow a fair comparison with the Xeon W3505 CPU which also adopts 45 nm technology. We suppose that all the inputs of our Lucas module are driven by a D flip-flop. We also assume that all the outputs of our module have the capacitive load of the same D flip-flop. We estimate the power of each synthesized architecture using Synopsys PrimeTime-PX. For our estimation, the same input as in the previous section is considered. The toggle rates of the different nets are obtained by post-synthesis simulation. In order to obtain a more accurate estimation of the power consumption, a virtual clock network is created for each architecture. We do not use any advanced power optimisation technique such as clock gating and dynamic voltage scaling. The constraints used for synthesis and power estimation as well as example scripts are given in appendix D.2. Our results are reported in Table 5.4. The area is given in 2-input NAND gates. One NAND gate corresponds to 4 transistors.

We see that for our 65 nm architectures running at 1 GHz, the variation of the execution time with r and p follows the expected trend. Namely the execution time decreases almost linearly with r whereas it is less impacted by an increase in p. This is due to the fact that the pipeline of the multiplier is not always full. However, for r greater than 4, the replicated carry-save adders of our Montgomery multiplier are on the critical path of the Lucas module. Hence the Lucas prime tester's maximum frequency decreases, which results in some performance lose.

Our fastest 65 nm ASIC implementation is 8 times faster than the FPGA implementation while only consuming 369 mW. It is at least 39 times more energy efficient. Our fastest 45 nm ASIC implementation is 4 times faster and 420 times more energy efficient than the software implementation. The cell count and the power consumption of our implementations make them suitable for integration as a co-processor into a general purpose CPU or an embedded system, different speed/area/energy trade-offs being available through parametrization.

Table 5.4: ASIC implementation results of 1024 bit Lucas primality testers

Type	Area (kGates)	Freq. (GHz)	Max Exec. Time (ms)	Dynamic Power (mW)	Energy (mJ/op)
65 nm ASIC ($p = 1$, $r = 1$)	426	1.00	5.26	325.1	1.73
65 nm ASIC ($p = 1$, $r = 2$)	441	1.00	2.65	338.5	0.91
65 nm ASIC ($p = 2$, $r = 1$)	485	1.00	2.94	394.1	1.17
65 nm ASIC ($p = 2$, $r = 2$)	516	1.00	1.63	412.9	0.68
65 nm ASIC ($p = 3$, $r = 1$)	546	1.00	2.61	453.4	1.20
65 nm ASIC ($p = 3$, $r = 2$)	595	1.00	1.57	472.9	0.75
65 nm ASIC ($p = 3$, $r = 4$)	658	0.75	1.38	368.9	0.52
65 nm ASIC ($p = 3$, $r = 8$)	623	0.50	1.68	238.9	0.41
45 nm ASIC ($p = 3$, $r = 2$)	644	1.66	0.94	334.3	0.32
45 nm ASIC ($p = 3$, $r = 4$)	714	1.25	0.83	264.8	0.22
Software	N/A	2.53	3.30	28 000	92.40

5.6 Summary

This chapter presents our parametric hardware architecture of the NIST approved Lucas probabilistic primality test and a novel architecture for the Miller-Rabin primality test. Primality testing is needed by the key generation process of many public-key crypto-systems. Our architectures reuse the scalable Montgomery multiplication and exponentiation designs presented in chapter 3.

The Miller-Rabin test is the most common primality test for cryptographic systems. Our implementations are up to 49 times faster than other hardware implementations while only twice as large. They are up to 2.9 times more efficient in terms of Time $\times$ Area product. Moreover our best implementation is 1.6 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz.

The use of a Lucas test as the last step of a probabilistic prime tester can greatly reduce

the probability of error of the overall test. To our knowledge, our work is the first hardware implementation of the Lucas test in the literature. Our architecture implemented on a Virtex-5 FPGA is 3 times slower but is up to 5 times more energy efficient than the software version running on a Intel Xeon W3505. A 65 nm ASIC implementation is more than 8 times faster and 39 times more energy efficient than the FPGA implementation. A 45 nm ASIC implementation is 4 times faster and 420 times more energy efficient than the optimised software implementation in comparable technology. Our design is scalable and the performance scaling of our architecture is much better than linear in area. The cell count and the power consumption of our ASIC implementations make them suitable for integration into a general purpose CPU or an embedded system whereas our FPGA implementation would more likely benefit server application.

Our prime tester implementations are limited by data dependencies and cannot explore the full design space provided by the multiplier and the exponentiator presented in chapter 3. Therefore they do not perform very well compared to software. However, our hardware implementation are significantly more energy efficient than software on FPGAs and on ASICs and mostly use hardware that is already present for encryption. Moreover, if we suppose that an attacker can obtain physical access to the crypto-system, implementing such cryptographic modules purely in hardware greatly reduces the vulnerability of the system when compared to a software implementation, which is usually run in a multi-tasking operating system. But without proper protections our cryptographic modules are still vulnerable to side-channel attacks. New countermeasures against power attacks, a typical example of side-channel attacks, are investigated in chapter 6.

Chapter 6

Power Attacks Countermeasures Involving On-Chip Monitoring

In this chapter we present a novel approach involving on-chip power monitoring to prevent power attacks on reconfigurable hardware. Our power monitor circuit is based on a network of evenly distributed ring-oscillators. Two power attack countermeasures are derived from this circuit. The first countermeasure is a general framework based on a closed-loop control system that keeps the power consumption of any FPGA implementation constant. The second countermeasure involves detecting the insertion of a shunt resistor-based power measurement circuit onto a device's power rail. After introducing the common shunt resistor-based power measurement setting, we present our on-chip monitor circuit. Then we describe and evaluate our two countermeasures.

6.1 Motivation

As shown in sections 2.2 and 2.3, encryption algorithms are designed to make brute-force attacks or exhaustive key search computationally infeasible and to resist cryptanalysis based on theoretical weaknesses. However, the physical implementation of an encryption algorithm can leak information and create security flaws. Attacks exploiting these physical flaws are called side-channel attacks.

Since their initial publication [KJJ99], a relevant type of side-channel attacks called power

attacks have been extensively studied. As explained in section 2.4, power attacks recover the key of the encryption algorithm by using one or multiple power traces, which are directly correlated to the output capacitance of the transistors in the device as well as their output switching frequency. They have been successfully demonstrated on many common encryption methods, including private key encryption such as DES [SOQP04] and AES [SMPQ06], finite field based public key encryption such as RSA [MDS99] and Diffie-Hellman, and elliptic curve based public key encryption [OOP03]. Theoretically, power attacks can be used to attack any crypto-system with a key-dependent power consumption.

Two major types of countermeasures exist in order to make an implementation resistant to power attacks. Masking countermeasures randomize the intermediate values processed by the cryptographic device. These countermeasures are application-dependent as they require the algorithm to be modified. Moreover, they are resource consuming. For instance random masking for AES can lead to 2 to 3 times area overhead [RWS11]. Hiding countermeasures remove the data dependency of the power consumption. The most common hiding countermeasures, namely differential logic and symmetrical routing, implement the complement of each logic operator and attempt to route it symmetrically to the initial operator. The idea is to always consume the same amount of power independently of the input data. These techniques are very efficient in protecting a hardware crypto-system against power attacks. However, they lead to 3 to 10 times area overhead and slow down the circuit [TV04, YS07].

Our work takes a different approach. We measure the power consumption of the device on-chip (section 6.3) and use this information to either prevent or detect power attacks. Our first countermeasure presented in section 6.4 is a hiding countermeasure that keeps the power consumption constant by using a closed-loop control system using our power measurement circuit as the sensor, a power consumer circuit as the actuator, and a PID controller. It has a relatively low area footprint, which is independent of the size of the crypto-system. However this countermeasure is not easily feasible on current commercial FPGAs due to the limited speed of the control loop. Our second countermeasure presented in section 6.5 detects the insertion of a shunt resistor-based power measurement circuit by comparing the power variations in the circuit at runtime with values obtained during calibration. We are not aware of previous work

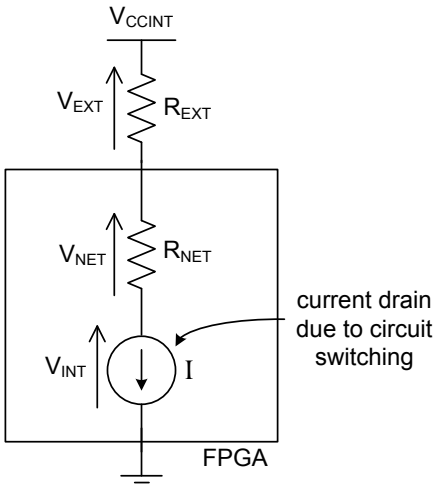


Figure 6.1: FPGA power measurement model

on detecting power attacks on reconfigurable hardware.

6.2 Power attack measurement settings

Fig. 6.1 shows a simplified model of the most common setting to measure the power consumption of an FPGA chip as presented in [MOP07, OOP03]. A shunt resistor R_{EXT} (typically 1 ohm to 50 ohms) is inserted into the core logic power supply line V_{CCINT} of the FPGA. R_{NET} represents the equivalent resistance of the FPGA power network. I is the current drain due to circuit switching. The power consumed by the FPGA is given by the following equations:

$$P = V_{INT} I = (V_{CCINT} - V_{TOT}) I \tag{6.1}$$

$$V_{TOT} = V_{EXT} + V_{NET} \tag{6.2}$$

$$R_{TOT} = R_{EXT} + R_{NET} \tag{6.3}$$

If the voltage drop due to the resistors V_{TOT} is small compared with V_{CCINT}, the power consumption of the device can be approximated by the following equations:

$$P \approx V_{CCINT} I \tag{6.4}$$

$$I = V_{EXT} / R_{EXT} \tag{6.5}$$

$$I = V_{TOT} / R_{TOT} = (V_{CCINT} - V_{INT}) / R_{TOT} \tag{6.6}$$

As shown in the equations, the power consumption of the FPGA is approximately proportional to the voltage drop across the resistors. An attacker with physical access to the voltage supply pin can obtain a power trace by inserting a shunt resistor R_{EXT} and measuring the voltage drop V_{EXT}. We can also monitor the FPGA's power consumption by measuring the internal voltage V_{INT} as shown in equation 6.6.

Ideally, in an attack scenario the power measurement circuit should not modify the electrical behaviour of the FPGA board. However, in order to obtain a clean power trace R_{EXT} cannot be too small. Even for 1 ohm, the voltage drop across the shunt resistor V_{EXT} is not completely negligible. Hence the FPGA supply voltage V_{INT} is smaller than V_{CCINT}. In practice, V_{CCINT} needs to be increased after programming the device to allow the FPGA to run close to normal operating voltage. Even with this compensation the FPGA supply voltage V_{INT} cannot be kept constant at all time. In fact, when the device is running, I and therefore V_{EXT} vary due to circuit switching. The variations in V_{EXT} are what enables the attacker to obtain power traces. However they also create variations in the FPGA supply voltage V_{INT} that are much larger than without the shunt resistor.

The power consumption of a device can also be measured using an electromagnetic (EM) probe or a contactless current probe. In this chapter, we only focus on the power measurement circuit of Fig. 6.1.

6.3 On-chip power monitoring with ring oscillators

As shown in section 6.2, R_{TOT} creates variations in the FPGA supply voltage V_{INT}. Ring oscillators (ROs) are perfect candidates to monitor these variations. Since the circuit switching speed of an FPGA is correlated with its supply voltage V_{INT}, the oscillation frequency of a RO is affected by the supply voltage [FBCP10]. A linear approximation can be used to model the relationship between the FPGA supply voltage V_{INT} and the oscillation frequency f_{R_i} of ring oscillator i:

$$f_{R_i} \approx k_i V_{INT} + f_i \tag{6.7}$$

where k_i and f_i are positive constants. Note that f_{R_i} also depends on the chip temperature. However, for a low number of inverters in the RO, we can neglect the variations with temperature compared to the variations with voltage. As a matter of fact the authors of [FBCP10] demonstrate experimentally that a 3-inverter RO implemented on a Virtex-5 is much less sensitive to temperature than to voltage variations and that the voltage sensitivity decreases with the number of inverters. The oscillation frequency of ROs can also be subject to crosstalk effects. The authors of [GWWT12] demonstrate a variation of ring oscillator frequencies of up to 5% on Virtex-5 and Virtex-6 due to capacitive coupling from long interconnects, whereas no significant effect is observed on other interconnect types. As none of our RO implementations uses long interconnects, this effect should not be observed in our power monitor.

The major challenge of using a RO to measure the FPGA's supply voltage is the trade-off between time resolution and response time. In order to obtain a sufficient resolution, we need to accumulate enough oscillations from the RO. This implies running the RO for a long period of time, which increases the measurement period T_s. However, increasing the measurement period decreases the number of measurements that can be taken per second and therefore reduces the sampling rate of the power monitor. A solution is to make the RO oscillate faster by decreasing the number of inverters. However, on FPGAs the inverters of ring oscillators are built from LUTs. Hence even a 1-inverter RO[1] does not oscillate very fast (around 350 MHz in our experiments on Spartan-6 presented in section 6.4.4).

Another improvement is to evenly distribute a network of ROs among the FPGA. The oscillations from each RO are accumulated locally during a fixed amount of time. All the accumulated values are then summed together and used as the power measurement. This is shown in Fig. 6.2. This solution provides a more consistent measurement because the effect of voltage variations within the FPGA is averaged. Moreover by averaging the values of uniformly distributed ROs, we reduce the influence of random wire delay variations on the power monitor reading.

[1]Such a RO oscillates on FPGAs due to the relatively long logic and wire delays.

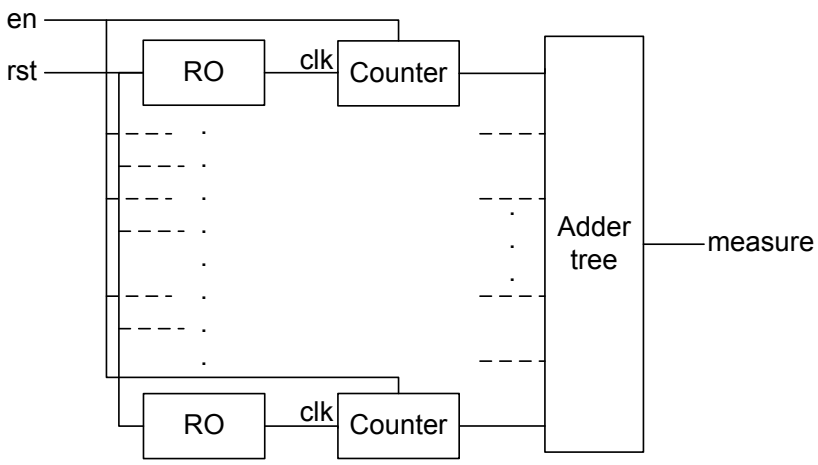


Figure 6.2: Power monitor

For m ROs, the power monitor output is as follows:

$$p = \frac{1}{f_s} \sum_{i=0}^{m-1} f_{R_i} \tag{6.8}$$

where $f_s = 1/T_s$ is the sampling frequency of the power monitor.

Some recent FPGAs such as the Xilinx Virtex-6 have system monitoring capabilities through ADCs that could also be used to monitor the supply voltage V_{INT} [Xil10b]. However the sampling rates of these ADCs are low (200 kHz for the Virtex 6 system monitor [Xil12c]). A RO-based power monitor has a much higher sampling rate (around 25 MHz in our implementation) that can be scaled according to the area available and the advances of fabrication technology. Moreover, ROs can be built using primitives that are available to all commercial FPGAs.

Our RO-based power monitor is the core element of the two countermeasures presented in sections 6.4 and 6.5.

6.4 Constant power reconfigurable computing

6.4.1 Principles

Our first countermeasure using on-chip power monitoring is summarised in Fig. 6.3. Our goal is to keep the power constant at a certain value (the setpoint) higher than the maximum power consumed by the user logic. This is illustrated in Fig. 6.4. The framework is based on the

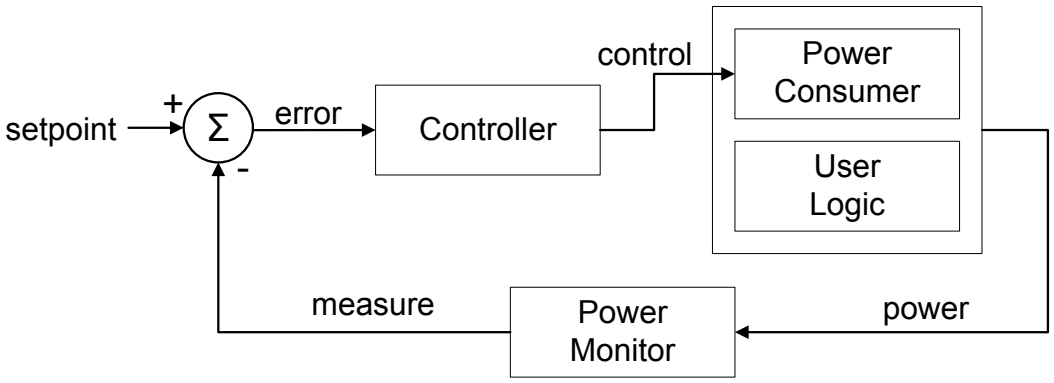


Figure 6.3: Constant-power framework

principles for a closed-loop control system.

The three main components of the control system together with their requirements are presented below:

Power monitor. The power monitor measures the on-chip power of the FPGA. Its input is a value correlated to the on-chip power consumption, such as the average voltage across the power network of the FPGA. Its output is a value proportional to the input that can be easily interpreted by the controller. The power monitor should provide precise and uniform power measurement across the chip. Its resolution should be high enough to detect any small variation of power that can be measured externally.

Power consumer. The power consumer is used to compensate for the on-chip power consumption. The power amplitude of the consumer should be higher than the power dynamic range of the user logic (as defined in Fig. 6.4) so that the power of the system can be kept constant. Its resolution should be high enough to compensate for the smallest variation of power measurable by the power monitor.

Controller. The controller manages the power consumer. Its goal is to make the measurement given by the power monitor match the setpoint. In order to quickly compensate for any power variation of the user logic, the controller needs to be chosen and tuned so that it has good regulation properties. In particular, the controller's response to a sudden power change should be fast enough to hide the power trace of the operations performed by the

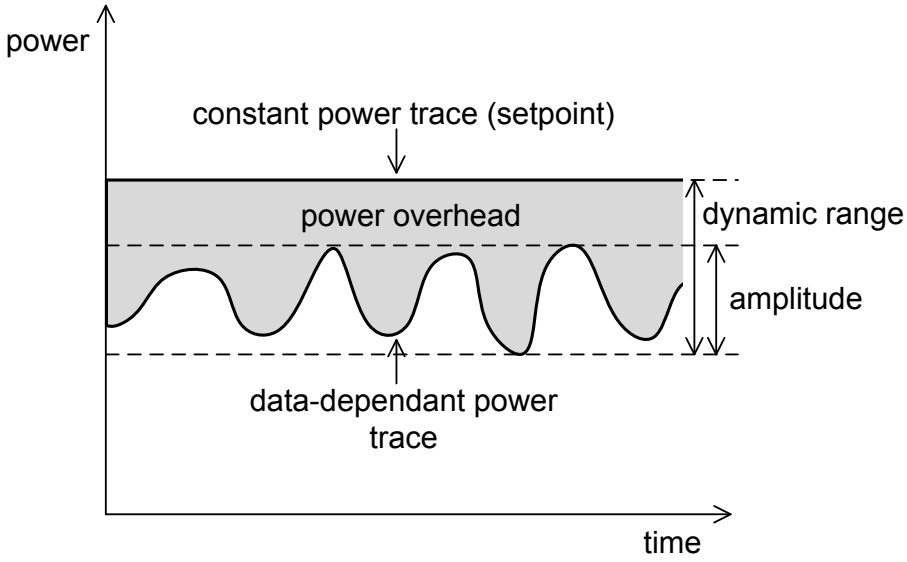


Figure 6.4: Power control principle

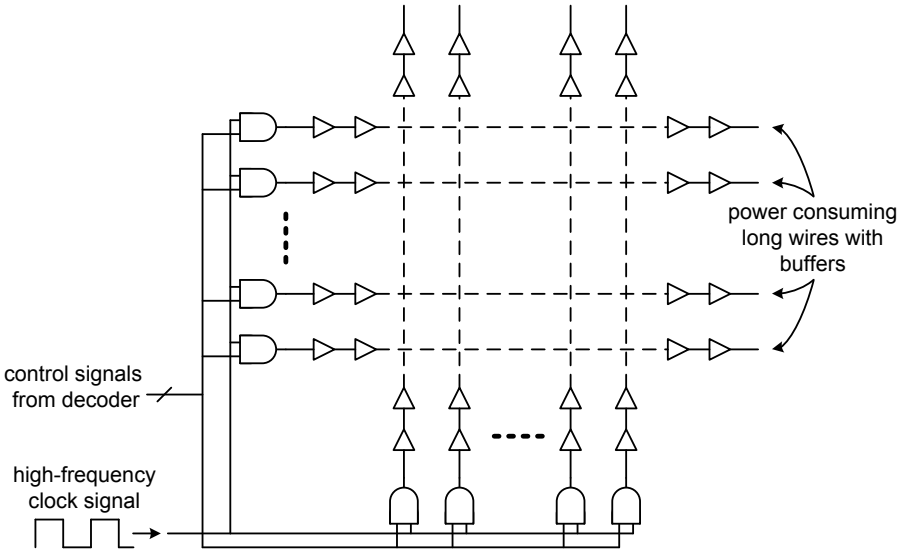


Figure 6.5: Power consumer

user logic.

The following sections describe our implementations of the power consumer and the controller. Like the power monitor, these two modules are self-contained into the FPGA fabric. This makes our framework resistant to attacks which would involve removing or replacing some of the on-board power modules in order to bypass the power regulation process.

6.4.2 On-chip power consumer

Fig. 6.5 shows the architecture of our power consumer. It consists of two major components: the power consuming wires and the control circuit. A power consuming wire is a routing interconnect that spans edge to edge vertically or horizontally across the FPGA. In modern commercial FPGAs all these long routing interconnects are buffered many times to reduce the logic delay. When a periodic switching signal such as a clock signal is fed into one of these long wires, current is drawn at each buffer in order to drive the parasitic capacitance along the wire. Thus significant power is drawn along the wire being activated. We distribute the power consuming wires evenly across the FPGA. We control the number of activated wires using a decoder and an array of AND gates. The hardware descriptions and constraints to guide the uniform placement of the wires are generated automatically by a script. A multiplexer is used at the clock input of the power consumer array to choose between several clock signals with different frequencies. The power consumed by the power consumer can be calculated as follows:

$$P_{consumer} = NCV_{INT}^2 f_{sw} \tag{6.9}$$

where N is the number of activated wires, C is the parasitic capacitance of each wire, V_{INT} is the supply voltage of the FPGA's core logic and f_{sw} is the frequency of the power consumer's switching signal.

6.4.3 Power controller

We use a proportional-integral-derivative (PID) controller to regulate our system. The PID controller is a commonly used feedback controller and, if well-tuned, has very good regulation and response properties. The controller module has two different modes:

Configuration mode. The following parameters of the system are determined and set up: the power setpoint, the optimal clock frequency of the power consumer, the optimal proportional, integral and derivative constants of the PID controller.

Regulation mode. The PID controller is regulating the power consumption of the FPGA.

When the FPGA is powered on, the controller begins with the configuration mode. The controller configurations sequence is shown in Fig. 6.6. First, the user logic is operated during a certain amount of time during which the minimum and maximum power values are obtained. This determines the user logic's power amplitude as shown in Fig. 6.4. The configuration run is not protected against power attacks. Hence for cryptographic applications, a key different from the secret key should be used. Then the setpoint is set to the maximum power value plus a given margin. This margin takes into account a possible increase in maximum power when using the actual secret key. The user logic's power dynamic range is calculated as the difference between the setpoint and the minimum power value. It is shown in Fig. 6.4 and corresponds to the maximum power that needs to be generated by the power consumer. Then the power consumer clock frequency is tuned so that the power consumer's amplitude is greater than but as close as possible to the user logic's power dynamic range. If the power consumer's amplitude is smaller than the user logic's power dynamic range, the control system might not be able to compensate for the user's logic power consumption. However, if the power consumer's amplitude is much higher than the user logic's power dynamic range, the control system is likely to experience quantization effects which would reduce its effectiveness. Finally, the PID controller parameters are determined using the relay feedback auto-tuning method.

The relay feedback auto-tuning method is commonly used to find the optimal parameters of a PID controller. We make the output of the system oscillate by alternating the control command between its maximum value and its minimum value. This corresponds to replacing the PID controller with a relay.

The principles of the PID auto-tuning method applied to our system are shown in Fig. 6.7. First the power consumer is set to half its maximum control value $C_{max}/2$ in order to determine the nominal bias value for the power monitor around which the system would oscillate. Then the control command is set to 0. In order to maintain the oscillations, the power consumer control value is set to C_{max} when the power monitor measurement value becomes greater than the nominal bias value. It is set back to 0 when the power monitor measurement value becomes less than the nominal bias value. We wait for the oscillations to stabilise and obtain the amplitude A and the period T of the oscillations. We stop the PID auto-tuning procedure after

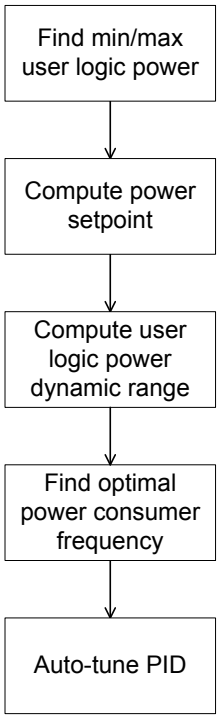


Figure 6.6: Controller configuration sequence

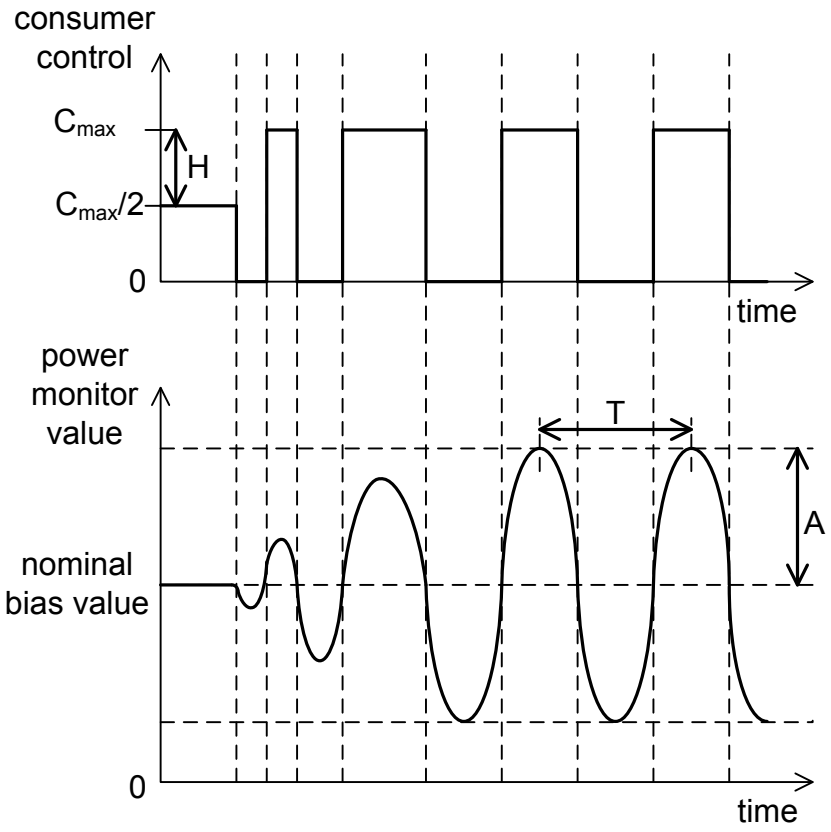


Figure 6.7: PID auto-tuning method

a few hundred oscillations are recorded.

The ultimate gain can be computed as follows:

$$K_u = \frac{4H}{\pi A} \tag{6.10}$$

where H is the control action (amplitude of the command) and A is the amplitude of the response. The ultimate period T_u is equal to the oscillation period T. Using the notations of Fig. 6.7, we deduce:

$$K_u = \frac{2C_{max}}{\pi A} \tag{6.11}$$

$$T_u = T \tag{6.12}$$

In order to reduce the sensitivity to high-frequency noise, the derivative term of the PID controller is filtered by a first-order system with the time constant T_d/N. The controller includes an optional set-point weighting mechanism to prevent impulses in the control signal when the reference value changes. Moreover an anti-windup mechanism based on back-calculation is used to prevent large transients when the power consumer saturates. The PID control equation and these three advanced techniques are detailed in appendix A. The proportional, integral, derivative and tracking constants of the PID controller are:

$$K_p = K \qquad K_i = \frac{Kh}{T_i} \tag{6.13}$$

$$K_{d0} = \frac{T_d}{T_d + Nh} \qquad K_{d1} = \frac{KT_dN}{T_d + Nh} \tag{6.14}$$

$$K_t = \frac{h}{T_t} \tag{6.15}$$

where h is the sampling period of the control loop and according to the ZNFD method for PID controllers [AM08]:

$$K = 0.6K_u \qquad T_i = 0.5T_u \qquad T_d = 0.125T_u \tag{6.16}$$

and:

$$T_t = \sqrt{T_i T_d} \tag{6.17}$$

After the configuration of the system is finished, the controller switches to regulation mode. The PID control algorithm is shown in Algorithm 6.1. This algorithm is based on the discretization of the PID controller equation. Each time a new value from the power monitor is received, the PID control command is computed and the power consumer control value is updated accordingly.

Algorithm 6.1: PID control algorithm

Input: C_{max}: max power consumer control command
$setpoint$: setpoint
K_p, K_{d0}, K_{d1}, K_i, K_t: PID controller constants

1 $P = D = I = 0$
2 $y = y_{prev} = v = v_{prev} = u = u_{prev} = 0$
3 **while** $PIDControllerRunning()$ **do**
4 $y = GetPowerMonitorValue()$ /* Get process variable */
5 $P = K_p(setpoint - y)$ /* Proportional action */
6 $D = K_{d0}D - K_{d1}(y - y_{prev})$ /* Derivative action */
7 $I = I + K_i(setpoint - y_{prev}) + K_t(u_{prev} - v_{prev})$ /* Integral action */
8 $u = v = P + I + D$ /* Total control command */
9 **if** $v > C_{max}$ **then** $u = C_{max}$ /* Saturated control command (high) */
10 **if** $v < 0$ **then** $u = 0$ /* Saturated control command (low) */
11 $SetPowerConsumerControl(u)$ /* Update power consumer command */
12 $y_{prev} = y$, $v_{prev} = v$, $u_{prev} = u$ /* Store current values */
13 **end**

The constant-power framework control rate is given by:

$$f_c = \frac{f_l}{l_c} \tag{6.18}$$

f_l is the running frequency of the controller, power monitor and power consumer logic. l_c is the control latency given by:

$$l_c = l_s + l_{add} + 4 \tag{6.19}$$

where l_s is the power monitor sampling latency and l_{add} is the power monitor's adder tree latency (in clock cycles). The constant factor consists of 1 clock cycle to register the power monitor reading, 1 clock cycle to register the adder tree output and 2 clock cycles to compute the PID control signal using DSPs. A new power consumer control command is issued every $h = 1/f_c$ seconds.

6.4.4 Evaluation

Experimental setting

We use an ExpressCard Spartan-6 LX150T FPGA board manufactured by BlueRISC. The board is modified with a power measurement circuit. The power measurement circuit consists of a 0.1 ohm shunt resistor inserted in the 1.2V power line after the regulator output capacitors. The 3.3V and 1.2V switching regulators are replaced with more stable low dropout regulators. The 1.2V can be adjusted via a variable resistor. We use a Tektronix MSO 2024 oscilloscope with a 200 MHz bandwidth and a 500 MHz sampling rate for all our measurements. An SMA connector is soldered across the shunt resistor. The oscilloscope is connected to the board via an SMA-to-BNC cable. The cable has a characteristic impedance of 50 Ω and is rated for frequencies up to 12.4 GHz. The low-pass filter of the oscilloscope is set to 150 MHz to eliminate noise in our measurement circuit.

Case study

We consider a hardware implementation of 512-bit binary left-to-right modular exponentiation using the square-and-multiply algorithm as shown in Algorithm 6.2. The square and multiply operations are both performed by the Montgomery multiplier, which makes them hardly distinguishable. To get a better reading of the power consumption, we implement four 512-bit modular exponentiation cores on our Spartan-6 LX150T FPGA and set the clock frequency to 5 MHz. All cores are given the same set of inputs in parallel[2]. The three components of our framework are implemented alongside the four exponentiation cores. For this experiment, the

[2]This scenario is rather unrealistic in practice but enables us to obtain a relatively clear signal with our medium quality measuring equipment (oscilloscope and probes).

power monitor uses a grid of 576 1-inverter ROs. The ring oscillators are oscillating at around 350 MHz. The power consumer consists of 231 vertical and horizontal interconnects. The elements of our framework are all running at 100 MHz. The power monitor sampling latency is set to $l_s = 2$ clock cycles and the adder tree latency is $l_{add} = 2$ clock cycles. Hence the constant-power framework control rate is $f_c = 12.5$ MHz. Our implementation of the framework takes 11 882 extra LUTs, that is 13% of the area available on the LX150T. To reduce the effects of noise, we perform each measurement 30 times and report the average power trace.

Algorithm 6.2: Left-to-right binary exponentiation

Input: X, N, $E = (e_{n-1}, ..., e_1, e_0)$
Output: $R = X^E \bmod N$

```
1 R = 1
2 for i = n − 1 downto 0 do
3     R = R^2 mod N                          /* Square */
4     if e_i = 1 then
5         R = R.X mod N                      /* Multiply */
6 end
7 return R
```

Chosen-message attack

To quantify the effectiveness of our framework, we mount a chosen-message doubling attack [FV03] against our 512-bit modular exponentiation module. The attack works by detecting collisions between squaring operations for related inputs X and X^2. A collision between a squaring operation at cycle $i + 1$ in the power trace of X and a squaring operation at cycle i in the power trace of X^2 occurs only if the key bit e_i is 0.

The power traces of the two operations are compared using the Phase Only Correlation (POC) waveform matching technique presented in [HMA+10, HNI+06]. For two waveforms $f(n)$ and $g(n)$ where $n = -M, ..., M$, the cross-phase spectrum $R_{FG}(k)$ is defined as [HNI+06]:

$$R_{FG}(k) = \frac{F(k)\overline{G(k)}}{\left|F(k)\overline{G(k)}\right|} \tag{6.20}$$

where F and G are the discrete Fourier transforms of f and g. The POC function $r_{fg}(n)$ is

defined as the inverse discrete Fourier transform of R_{FG}:

$$r_{fg}(n) = \frac{1}{N} \sum_{k=-M}^{M} R_{FG}(k) e^{j\frac{2\pi kn}{N}} \tag{6.21}$$

If the two waveforms are similar minus a given displacement, the POC function presents a sharp correlation peak at the location of this displacement. The sharpness of the correlation peak is evaluated using the Peak-to-Sidelobe Ratio $PSR = (peak - mean)/std$. In our case $PSR = (max - mean)/std$, where max is the maximum, $mean$ is the mean and std is the standard deviation of the POC function. We note PSR_i, the PSR obtained for the collision corresponding to bit e_i. More detail about the doubling attack and the POC waveform matching techniques are given in appendix B.

In a standard attack we would set a threshold for the PSR after performing calibration attacks with known keys. For each key bit e_i, if the PSR is greater than the threshold, we guess $e_i = 0$ (a collision occurred), otherwise we guess $e_i = 1$. Note that in order to determine the correct positions of the traces to compare for bit e_i, we need to know the correct values of bits $e_n, ..., e_{i+1}$. Hence the attack fails if one bit is guessed incorrectly.

In our experiment, we only perform a few iterations of an attack, knowing the key E we want to recover. We obtain the 24 PSR values for bits e_{n-8} down to e_{n-31}. The waveforms for e_n to e_{n-7} are slightly harder to compare due to capacitive effects in the measurement circuit, which are visible on the waveforms of the first few square/multiply operations. We split the PSR obtained in two sets:

$$S_0 = \{PSR_i : e_i = 0\} \tag{6.22}$$

$$S_1 = \{PSR_i : e_i = 1\} \tag{6.23}$$

We define the resolution of the attack as follows:

$$\rho = \frac{min(S_0)}{max(S_1)} \tag{6.24}$$

ρ is a measure of how easily we can detect a collision. If $\rho \leq 1$, we cannot distinguish PSR_i

for a key bit $e_i = 0$ from PSR_j for a key bit $e_j = 1$ and the attack fails.

Fig. 6.8 shows the resolution of the attack against the attack bandwidth. A bandwidth of 150 MHz corresponds to the full oscilloscope signal. Bandwidths from 1 MHz to 100 MHz are obtained by digitally filtering the oscilloscope signal with a FIR filter of order 1000. For bandwidths less than 10 MHz, the use of the framework prevents the chosen-message attacks. However for bandwidths greater than 10 MHz, including the full oscilloscope bandwidth the attack is made harder but cannot be prevented. This is due to the relatively low 12.5 MHz control rate f_c of the framework. Intuitively, our constant power framework filters out all frequencies less than the Nyquist frequency $f_c/2 = 6.25$ MHz. The fundamental frequency of the 5 MHz clock is filtered out, but all its harmonics remain. Enough information is present in the harmonics to still be able to perform the attack.

Hence, several obstacles have to be overcome to make the constant power framework a hiding power attack countermeasure. Looking at equations 6.18 and 6.19, we can increase f_c by increasing f_l and decreasing l_{add} and l_s. This would be possible using future FPGA technologies provided that the clock frequency of the crypto-system remains much slower than $f_c/2$. However, in order to get enough resolution on the power monitor reading, we would need faster ring oscillators (which would also be available on future FPGA technologies) and/or a larger number of on chip ROs.

A possible improvement to our constant framework would involve adding noise to the system. For example we could vary the power consumer command randomly in a range defined by the current control command calculated by the PID controller. This random change could occur faster than the control rate and therefore improve the bandwidth in which the countermeasure is effective.

Another, maybe more realistic approach, would be to implement our framework as a hard IP block on an FPGA dedicated to cryptography. The controller and the ROs could be made much faster, at a lower power cost. A fast on-chip ADC could also be considered to replace the RO-based power monitor.

From this section, we can conclude that designing hiding power attack countermeasures based on our on-chip power monitoring technique is not easily feasible on current commercial

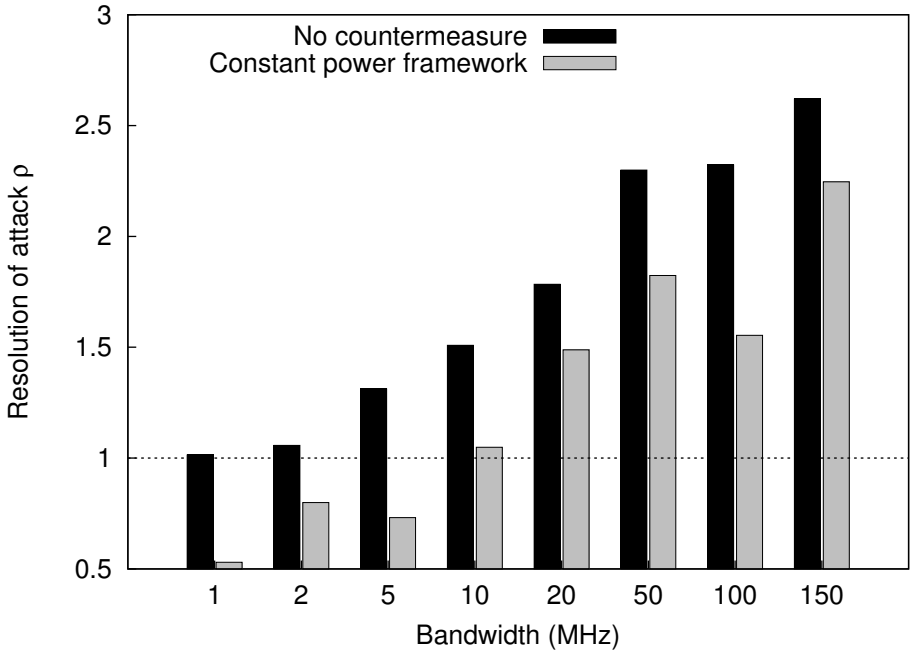


Figure 6.8: Resolution of the attack against bandwidth

FPGAs. However, on-chip power monitoring can be used to detect shunt resistor-based power attacks effectively, as shown in the following section.

6.5 Detecting Power Attacks

6.5.1 Principles

Our second countermeasure involves using on-chip power monitoring in order to detect power attacks. We still use the measurement method presented in section 6.2. The integration of our power attack detection framework into a typical crypto-system is shown in Fig. 6.9. A typical System-on-Chip (SoC) for cryptography consists of several hardware cores communicating through a system bus. Some cores implement critical cryptographic functions such as RSA, AES, or random number generation. Other non-critical cores are used to handle generic tasks such as communication (UART, Ethernet cores, ...), clock generation, etc. Usually a simple processor core controls the system. Our attack detection logic consists of two main modules: the power monitor and the attack detector.

The attack detector receives information about the state of each hardware module in the system. Using the information given by the power monitor and the knowledge of which hardware

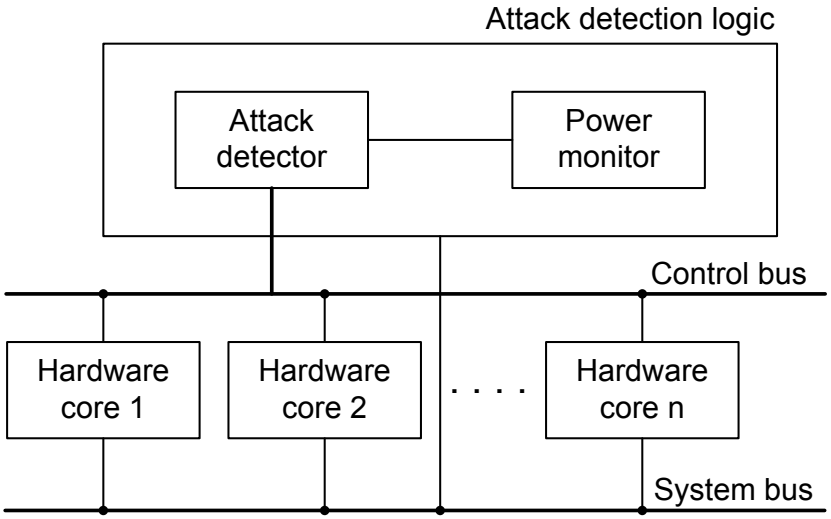


Figure 6.9: Power attack detection framework

modules are running by reading the control bus, the attack detector checks whether the power consumption of the device stays within a pre-defined range.

6.5.2 Attack detection model

The attack detector has two operating modes: calibration mode and monitoring mode. In calibration mode, the power characteristics of each hardware core are determined. These power characteristics depend on the FPGA board on which the system is running, in particular on the design of the power supply circuit of the board. Hence calibration is FPGA-dependent and has to be performed for each instance of the hardware cores and for each type of boards on which the crypto-system is implemented. The different calibration and monitoring stages are shown in Fig. 6.10.

First p_{ref}, the reference power monitor value when the system is idle is determined. In the idle state, the switching activity of the transistors and therefore the current I and the voltage drop V_{TOT} are minimum. Here no measurement circuit is present, therefore the voltage drop only depends on the equivalent resistance of the power rail R_{NET}, and V_{TOT} is equal to V_{NET}. A low voltage drop leads to a high power monitor value.

Then we determine the minimum reference power monitor value $p_{min,i}$ of each core i, corresponding to the highest voltage drop. In this step we provide each core with a sample of inputs representative of the operating conditions of the core. The reference power monitor amplitude

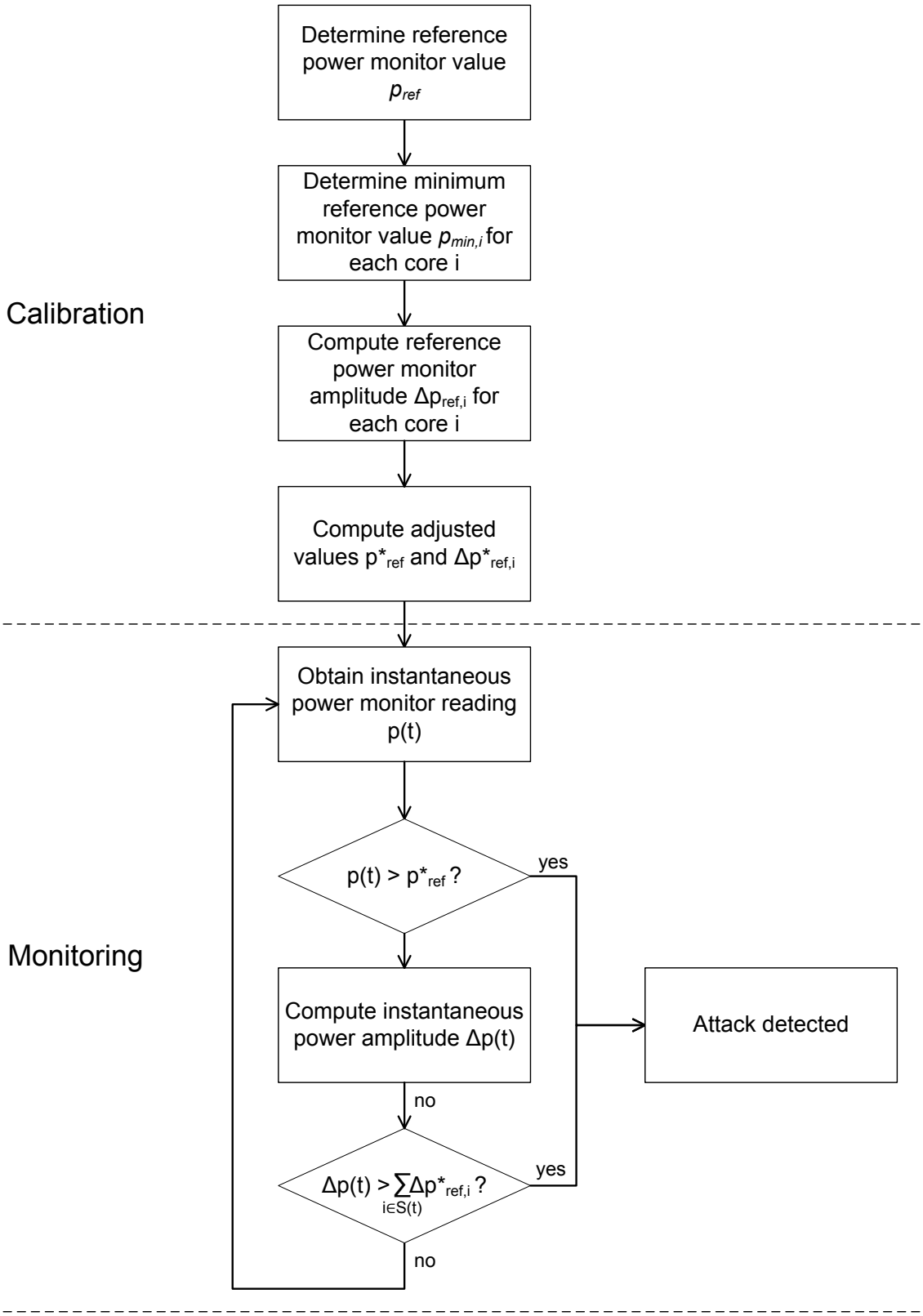


Figure 6.10: Calibration and monitoring stages

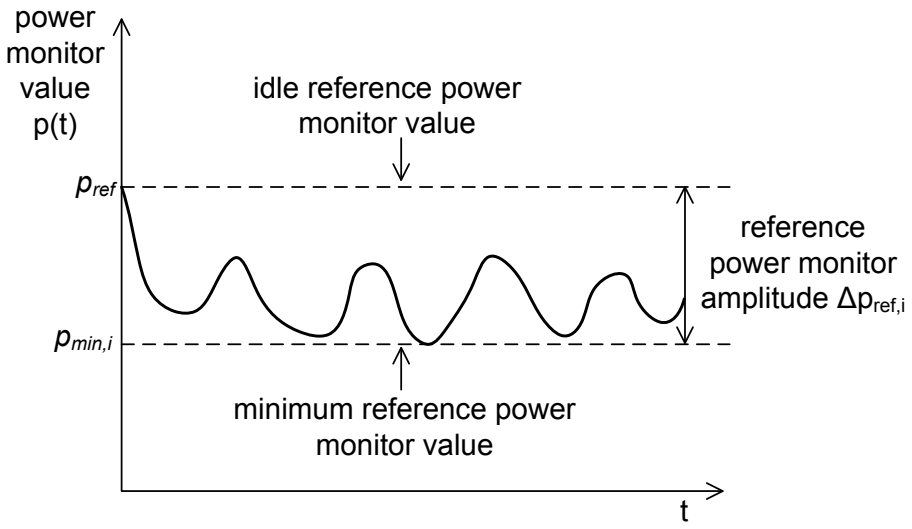


Figure 6.11: Calibration of a hardware core

of core i is:

$$\Delta p_{ref,i} = p_{ref} - p_{min,i} \tag{6.25}$$

p_{ref}, $p_{min,i}$ and $\Delta p_{ref,i}$ are shown in Fig. 6.11. In most cases we can only find approximations for these values as all combinations of input values of the cores cannot be tested. Hence, we allow margins on p_{ref} and $\Delta p_{ref,i}$, respectively m_{ref} and $m_{ref,i}$. We define:

$$p^*_{ref} = p_{ref}(1 + m_{ref}) \tag{6.26}$$

$$\Delta p^*_{ref,i} = (p^*_{ref} - p_{min,i})(1 + m_{ref,i}) \tag{6.27}$$

In monitoring mode, the attack detector is given p^*_{ref} and $\Delta p^*_{ref,i}$ for each core i. It monitors the instantaneous power monitor reading $p(t)$ and the instantaneous power amplitude:

$$\Delta p(t) = p^*_{ref} - p(t) \tag{6.28}$$

The attack detector records which hardware modules are currently running by reading their `start` and `done` signals on the control bus.

We assume that at time t, a subset $S(t)$ of the n hardware cores are running. An attack

flag is raised if:

$$p(t) > p^*_{ref} \quad \text{or} \tag{6.29}$$

$$\Delta p(t) > \sum_{i \in S(t)} \Delta p^*_{ref,i} \tag{6.30}$$

Equation 6.30 uses the fact that several cores running in parallel should not reach an amplitude higher than the sum of their reference amplitudes. This is an upper bound limit. One could also obtain reference values for every possible combination of cores running in parallel. However this may not be possible for a crypto-system with a large number of cores.

On one hand, the power amplitude of a core Δp is a relative value. Δp is proportional to the total resistance R_{TOT} on the reference FPGA power rail. As a matter of fact, let us assume that only core i is running and its switching during a fixed time interval Δt leads to a variation of current:

$$\Delta I_i = I_{i,max} - I_{i,min} \tag{6.31}$$

If V_{CCINT} is fixed, the maximum and minimum FPGA supply voltages are respectively:

$$V_{INT,max} = V_{CCINT} - R_{TOT} I_{i,min} \tag{6.32}$$

$$V_{INT,min} = V_{CCINT} - R_{TOT} I_{i,max} \tag{6.33}$$

Hence the maximum variation of supply voltage during the operation of core i is:

$$\Delta V_{INT} = R_{TOT} \Delta I_i \tag{6.34}$$

Using equation 6.7 the variation in oscillation frequency of ring oscillator j is:

$$\Delta f_{R_j} \approx k_j \Delta V_{INT} = k_j R_{TOT} \Delta I_i \tag{6.35}$$

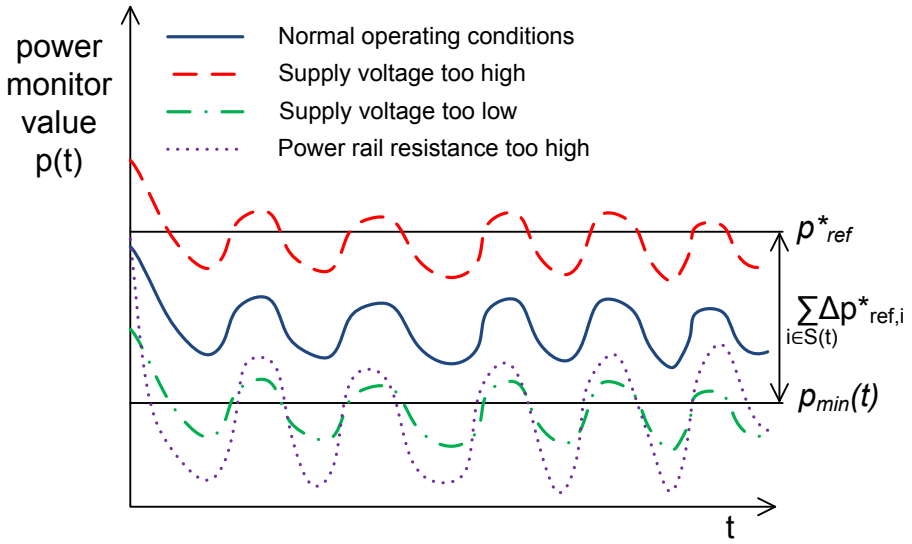


Figure 6.12: Operating conditions monitored by the attack detector

Hence, if the power monitor consists of m ROs and $m_{ref} = 0$, using equation 6.8 we have:

$$\Delta' p = p_{max} - p_{min} = \frac{1}{f_s} \sum_{j=0}^{m-1} \Delta f_{R_j} \approx \frac{1}{f_s} \sum_{j=0}^{m-1} k_j R_{TOT} \Delta I_i \tag{6.36}$$

Note that if we ignore the margin m_{ref}, $\Delta' p$ is equal to the maximum variations of Δp during Δt. Hence a change in R_{TOT} leads to a proportional change in Δp. This result also applies to $\Delta p_{ref,i}$. In particular, adding a shunt resistor R_{EXT} to the reference device in order to perform a power attack will lead to Δp being higher than $\Delta p_{ref,i}$ at a certain time t_d, as $\Delta p_{ref,i}$ have been obtained on the reference device without shunt resistor. This will be detected by equation 6.30.

On the other hand, the idle reference power monitor value p_{ref} and the instantaneous power monitor reading $p(t)$ are absolute values. The combination of equation 6.29 and equation 6.30 fixes $p(t)$ in a given range. This makes sure that an attacker is not tampering with the supply voltage of the FPGA.

Fig. 6.12 shows the different operating conditions detected by the attack detector. In the time frame of this graph, we assume that the number of hardware cores running in parallel is kept constant. Let us define:

$$p_{min}(t) = p^*_{ref} - \sum_{i \in S(t)} \Delta p^*_{ref,i} \tag{6.37}$$

In normal operating conditions, at time t the power trace $p(t)$ always stays between p^*_{ref} and $p_{min}(t)$. If the supply voltage is too high, p raises over p^*_{ref} and the attack flag is raised according to equation 6.29. If either the supply voltage is too low or the supply voltage is at a normal level but the power rail resistance is too high, p falls below p_{min} at a time t_d. Hence $\Delta p(t_d)$ is higher than $\sum_{i \in S(t_d)} \Delta p^*_{ref,i}$ and the attack flag is raised according to equation 6.30.

Depending on the application, different strategies can be adopted when an attack is detected. The critical cores can be reset to prevent the current power acquisition from being carried on, the corrupted keys can be revoked and new keys generated, or the keys can be erased from the cryptographic device, making the device unusable.

The attack detector sampling rate is given by:

$$f_d = \frac{f_l}{l_d} \tag{6.38}$$

f_l is the running frequency of the attack detector and power monitor logic. l_d is the attack detector latency given by:

$$l_d = l_s + l_{add} \tag{6.39}$$

Recall that l_s is the power monitor sampling latency and l_{add} is the power monitor's adder tree latency.

6.5.3 Evaluation of the model

Our experimental setting is based on two Pico E-101 FPGA boards. The Pico E-101 has a Spartan-6 LX45 FPGA. We modify one of the boards to include a power measurement circuit as shown in Fig. 6.13. The modification is similar to the one described in section 6.4.4. We use a Tektronix MSO 2024 oscilloscope with a 200 MHz bandwidth and a 500 MHz sampling rate for all our measurements.

The power monitor now consists of 128 3-inverter ring oscillators which can be activated independently. The attack detection logic is running at $f_l = 100$ MHz. The adder tree latency l_{add} is set to 2 clock cycles to meet timing requirements. The entire attack detection logic

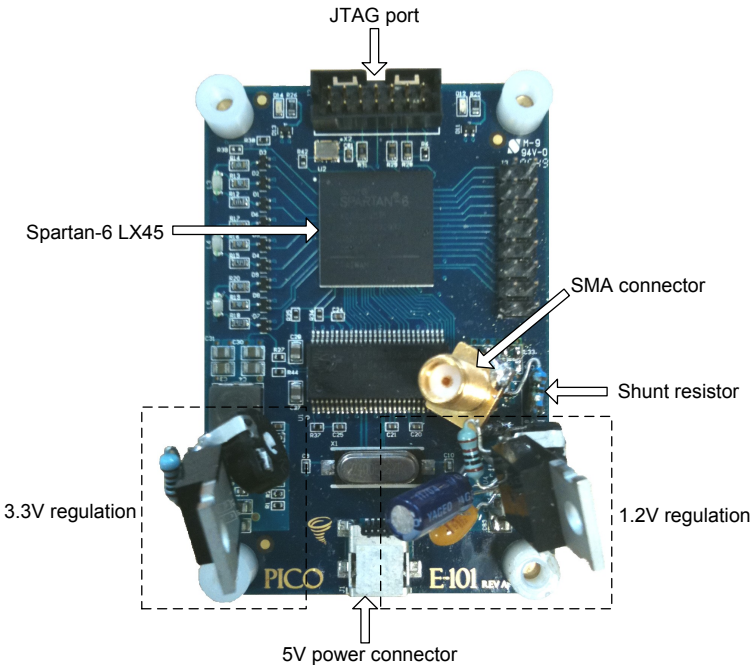


Figure 6.13: Modified Pico E-101 FPGA board

(attack detector and power monitor) takes 4365 LUTs, that is 16% of the area available on the LX45. For comparison, standard countermeasures such as masking and WDDL have area overheads of respectively 20% to 300% [RWS11] and 3 to 10 times [TV04, YS07].

Fig. 6.14 shows the oscillation frequency f_{R_i} of each RO i against the FPGA voltage V_{INT}. R_{EXT} is set to 0 and we vary V_{CCINT}. Note that we actually measure $(V_{NET} + V_{INT})$. However, assuming R_{NET} is small, the effect of V_{NET} can be neglected. We select a long power monitor sampling latency l_s of 180 clock cycles. We perform a linear regression on all of the 128 curves and obtain an average coefficient of determination R^2 of 0.997 with a standard deviation of 0.08%. This result shows that equation 6.7 is a relevant approximation of the RO behaviour. It is also interesting to observe that each RO, even if identically placed and routed, behaves slightly differently with the supply voltage. As a matter of fact, we measure a standard deviation of respectively 2.05% and 2.71% on the slopes k_i and intersects f_i given by the linear regressions. This deviation is due to both correlated and random variability in the chip manufacturing process. Note that random variability also happens in between similar ROs implemented on two different FPGA chips. However, it is reduced by averaging several RO outputs and should not affect our detection mechanism.

Fig. 6.15 shows the power traces of a 512-bit modular exponentiation core running at 20 MHz

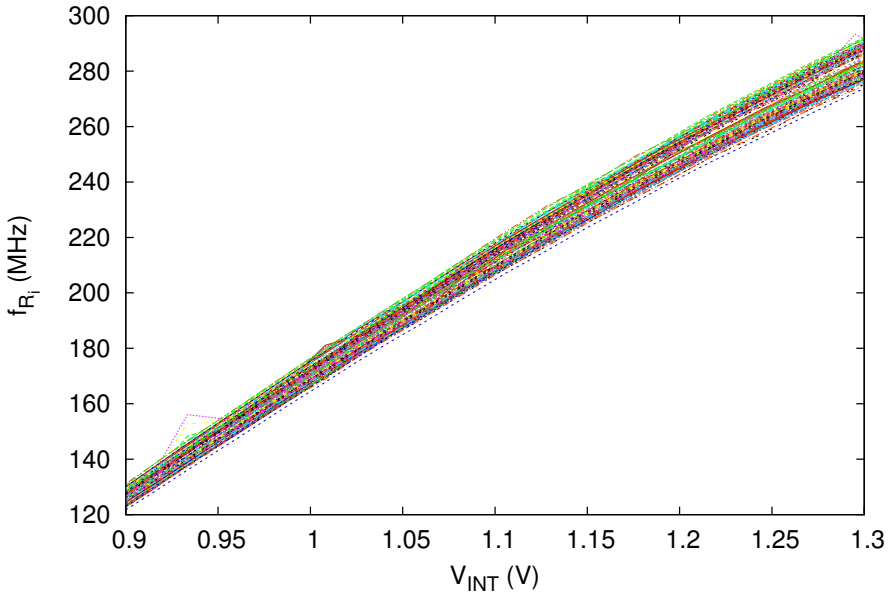


Figure 6.14: Frequency against supply voltage for each of the 128 ROs

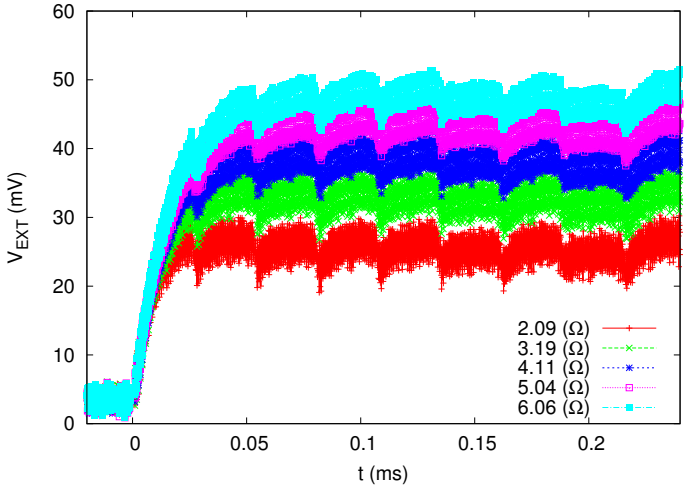


Figure 6.15: Oscilloscope power traces of modular exponentiation for different shunt resistor values R_{EXT}

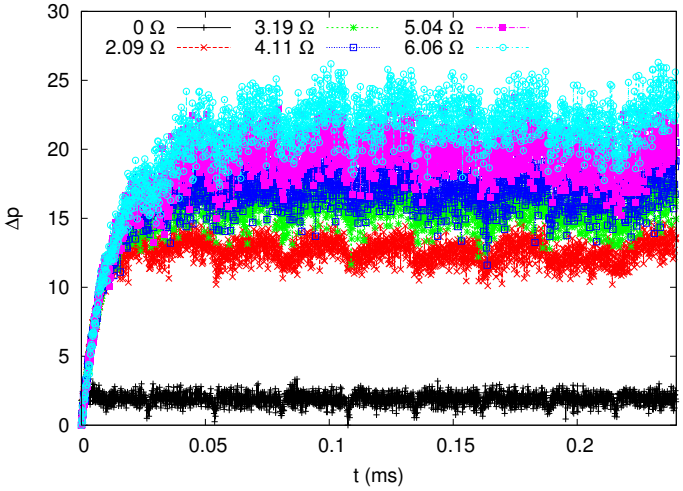


Figure 6.16: 16-RO power monitor traces of modular exponentiation for different shunt resistor values R_{EXT}

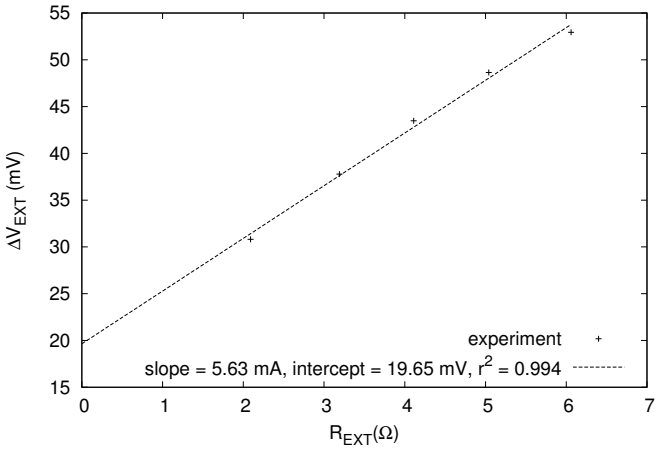


Figure 6.17: Amplitude of the shunt resistor voltage against the resistor value

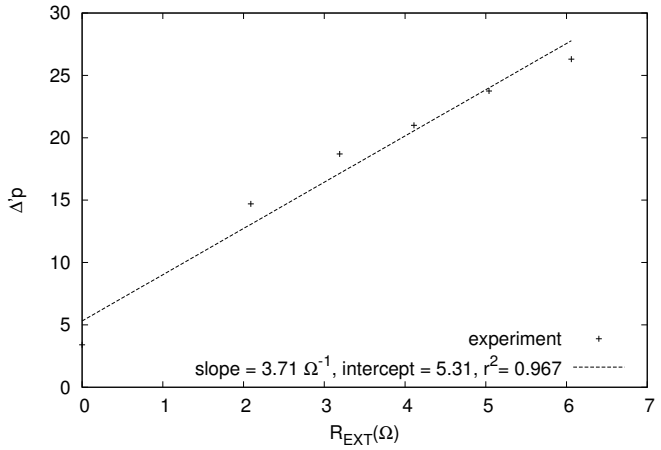


Figure 6.18: Amplitude of the power monitor reading against the resistor value

for different shunt resistor values[3]. It is obtained with the oscilloscope. The values reported for R_{EXT} in this section are the actual resistance values measured before starting the experiment. We cannot obtain actual shunt resistance values less than 2 Ω due to the resistance of the wires and solder in our setting. Before each run, V_{CCINT} is adjusted so that the FPGA is supplied with the voltage $V_{INT} + V_{NET} = 1.2\ V$ when idle. Fig. 6.16 shows the same power traces obtained with the on-chip power monitor for 16 activated ROs. For this experiment, the power monitor sampling latency l_s is set to 10 clock cycles. Hence, the attack detector sampling rate is $f_d = 8.3$ MHz. To reduce noise, both the power monitor and oscilloscope traces are averaged over 20 runs. In both cases the successive multiplication patterns are easily recognisable. The power traces obtained on-chip are obviously less detailed than the oscilloscope traces, which are sampled at a 60 times higher frequency. As expected, the amplitude of the power traces increases with the shunt resistor value. Fig. 6.17 shows the amplitude of the shunt resistor voltage V_{EXT} against the shunt resistor value R_{EXT}. According to our model, we should have:

$$\Delta V_{EXT} = R_{EXT} \Delta I \tag{6.40}$$

However, the linear regression does not fit this equation as it shows a non-negligible positive intersect of 19.65 mV. Several phenomena not taken into account by our model could explain

[3]As in section 6.4.4, we use a relatively low clock frequency to obtain a clear signal with our measuring equipment. However, we expect our attack detection framework to work similarly for cores running at higher frequencies, which would consume more dynamic power.

this behaviour. First, when R_{EXT} increases, the relative drop in the FPGA voltage V_{INT} during computation increases. Therefore, the FPGA's transistors are supplied with a lower voltage on average, which could cause $\Delta I = I_{max} - I_{min}$ to drop, violating our hypothesis that ΔI is constant. Second, internal capacitance of the circuit, which clearly affects the variations of current, are not taken into account by our model.

Fig. 6.16 also shows the power trace obtained with the power monitor when the shunt resistor is bypassed ($R_{EXT} = 0$). The successive multiplications patterns can still be distinguished because of the effect of R_{NET} and the resistance of our bypass circuit. Fig. 6.18 depicts the amplitude of the power monitor reading against the shunt resistor value. We see that $\Delta' p$ is also affected by the variations of ΔI, which limits the validity of equation 6.36.

The fact that $\Delta' p$ and therefore Δp increase with R_{EXT}, even if not proportionally, is enough to make our attack detector effective as shown in the next section.

6.5.4 Detection capabilities

In this section, we evaluate the detection capabilities of our framework. We consider a cryptosystem with five main cores running at 20 MHz: the power attack detection logic, a 512-bit RSA encryption core (core number 0), a 128-bit AES core (core number 1) [aes], a Microblaze processor controlling the different cores and a UART for communication. The UART is only used to get the RSA and AES inputs from a PC and to send the results back after the computation is finished. Hence it is never running in parallel with the RSA or the AES core. The processor only waits for an interrupt during an RSA or AES computation and we can therefore neglect its power consumption during this phase. When the power attack detection logic starts, we wait for a few thousand clock cycles before starting recording the power variations of the system. This prevents the attack detector from recording the relatively large power variations caused by the power monitor start-up. A 128-bit AES encryption finishes in 8 clock cycles whereas an RSA encryption finishes in between 275 968 and 551 936 clock cycles depending on the number of ones in the 512-bit key.

We compare the original Pico E-101 board against the modified board with shunt resistors ranging from 2 Ω to 8 Ω. The original board has no shunt resistor. Moreover the filtering

capacitors, which are removed in the modified board, limit the power supply variations. The attack detector sampling rate is still $f_d = 8.3$ MHz. Fig. 6.19 shows stacked histograms of the distribution of $\Delta' p = p_{max} - p_{min}$ for 1000 random 512-bit RSA input pairs, using a different seed for the original and modified board. Not all resistors values can be tested for all ring oscillator numbers in our current settings. As a matter of fact, for a high shunt resistor value and a large number of ring oscillators, our regulation circuit cannot compensate for the large voltage drop when the circuit is idle. The $\Delta' p$ distributions of the original board only intersect with the one of the modified board for a 2-RO detector (not clearly visible in Fig. 6.19). Hence it should be possible to detect any attack with a shunt resistor above 2 Ω for all other tested RO number. However, the attack detector only discriminates between the $\Delta' p$ distributions of the different shunt resistor configurations from around 16 ROs. Below this value, the distributions are entangled due to the low power monitor resolution.

Table 6.1 show the percentage of detected attacks for three different configurations:

1. the original board (nominal $V_{INT} + V_{NET} = 1.2\ V$)
2. the modified board with no shunt resistor but $V_{INT} + V_{NET}$ set above its nominal value
3. the modified board with a shunt resistor $R_{EXT} = 2.09\ \Omega$

For each configuration and each RO number, we perform 10 000 RSA exponentiations with random inputs. $p^*_{ref,0}$ is set to the maximum power monitor value obtained during the calibration of the original board, that is we do not set any margin m_{ref}. $\Delta p^*_{ref,0}$ is set at equal distance from the maximum $\Delta' p$ obtained on the original board and the minimum $\Delta' p$ obtained for $R_{EXT} = 2.09\ \Omega$. This is shown in Fig. 6.19. In configuration 3, the voltage $V_{INT} + V_{NET}$ is manually set when the system is idle so that $p(t)$ is as close as possible to p^*_{ref}. This requires access to the value of the power monitor, which an attacker would not acquire easily. In practice an attacker would simply set $V_{INT} + V_{NET}$ to its nominal value when the system is idle. Hence configuration 3 is a best case scenario in an attacker's point of view.

We see that configuration 2 is properly detected as high voltage for all numbers of RO tested with a false-negative rate of 0%. For RSA in configuration 1, we get a maximum false-positive rate of 3.71%. False-positives are either created by an amplitude of voltage higher

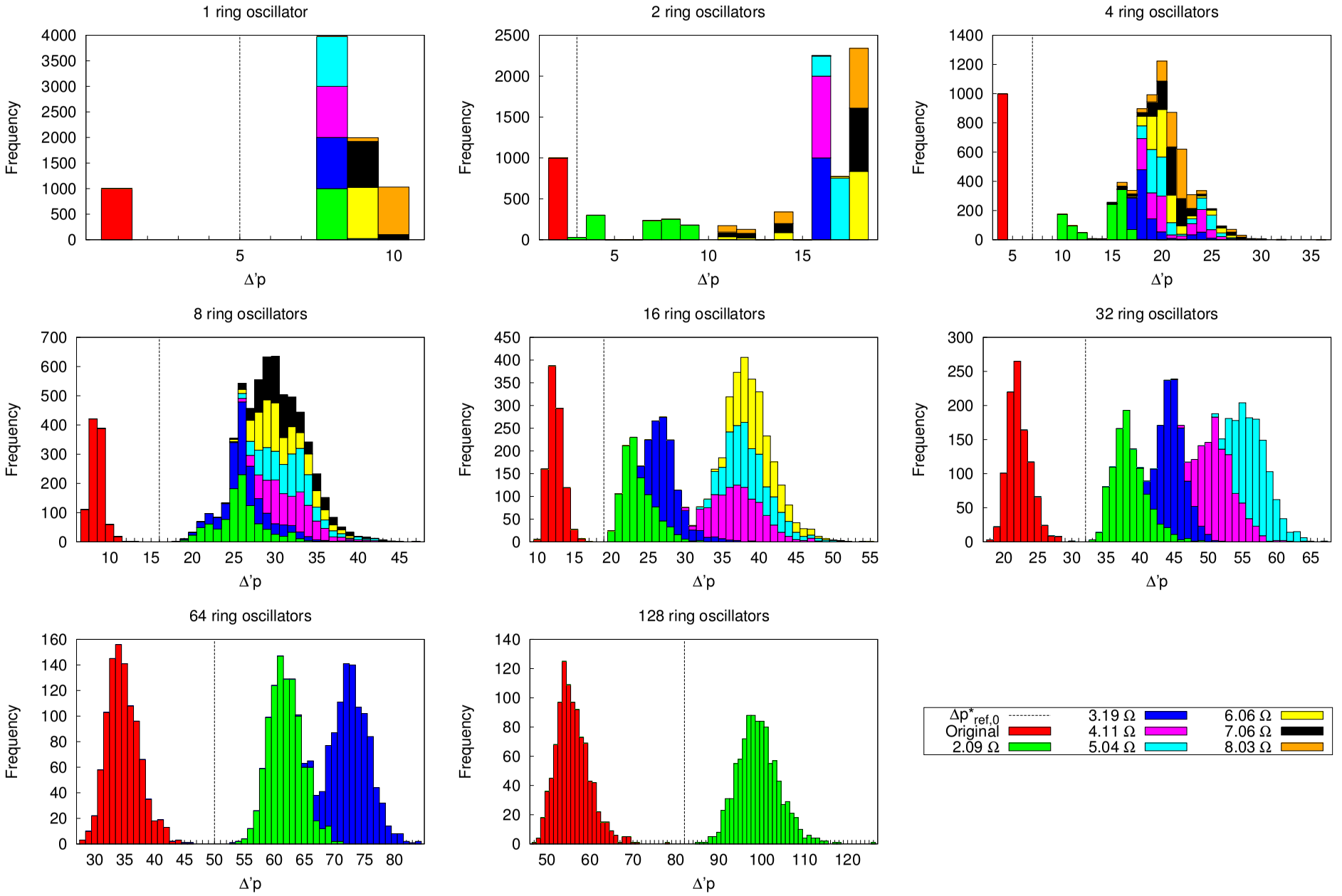


Figure 6.19: $\Delta' p$ distribution of 512-bit RSA for different number of ring oscillators and shunt resistor values

Table 6.1: Detected attacks (% of total runs - of which % of high voltage detections)

		1 RO	2 ROs	4 ROs	8 ROs	16 ROs	32 ROs	64 ROs	128 ROs
RSA	Original	0 - NA	0 - NA	0 - NA	0 - NA	3.71 - 0	0.03 - 100	0.23 - 96	0 - NA
	$V_{INT} + V_{NET} = 1.25\ V$	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100
	$R_{EXT} = 2.09\ \Omega$	100 - 100	100 - 100	100 - 76.8	94.26 - 100	100 - 0.01	100 - 0	100 - 0	100 - 0
AES	Original	0 - NA	0 - NA	0 - NA	0 - NA	0.04 - 0	0.03 - NA	0.4 - 22.5	0.02 - 0
	$V_{INT} + V_{NET} = 1.25\ V$	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100
	$R_{EXT} = 2.09\ \Omega$	0.01 - 100	100 - 100	16.5 - 100	0.07 - 100	0.03 - 0	12.94 - 0	0.27- 0	0.04 - 0
AES+	Original	0 - NA	0 - NA	0 - NA	0 - NA	0.01 - 0	0.21 - 4.8	0.25 - 12	0.32 - 12.5
	$V_{INT} + V_{NET} = 1.25\ V$	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100	100 - 100
	$R_{EXT} = 2.09\ \Omega$	3.27 - 100	100 - 100	3.73 - 100	100 - 100	99.56 - 0	100 - 0.01	100 - 0	100 - 0

Table 6.2: Attack detector power overhead for 512-bit RSA

	No detector	1 RO	2 ROs	4 ROs	8 ROs	16 ROs	32 ROs	64 ROs	128 ROs
Power (mW)	69	88	90	92	96	105	118	139	167
Overhead (%)	0	28	30	33	39	52	71	101	142

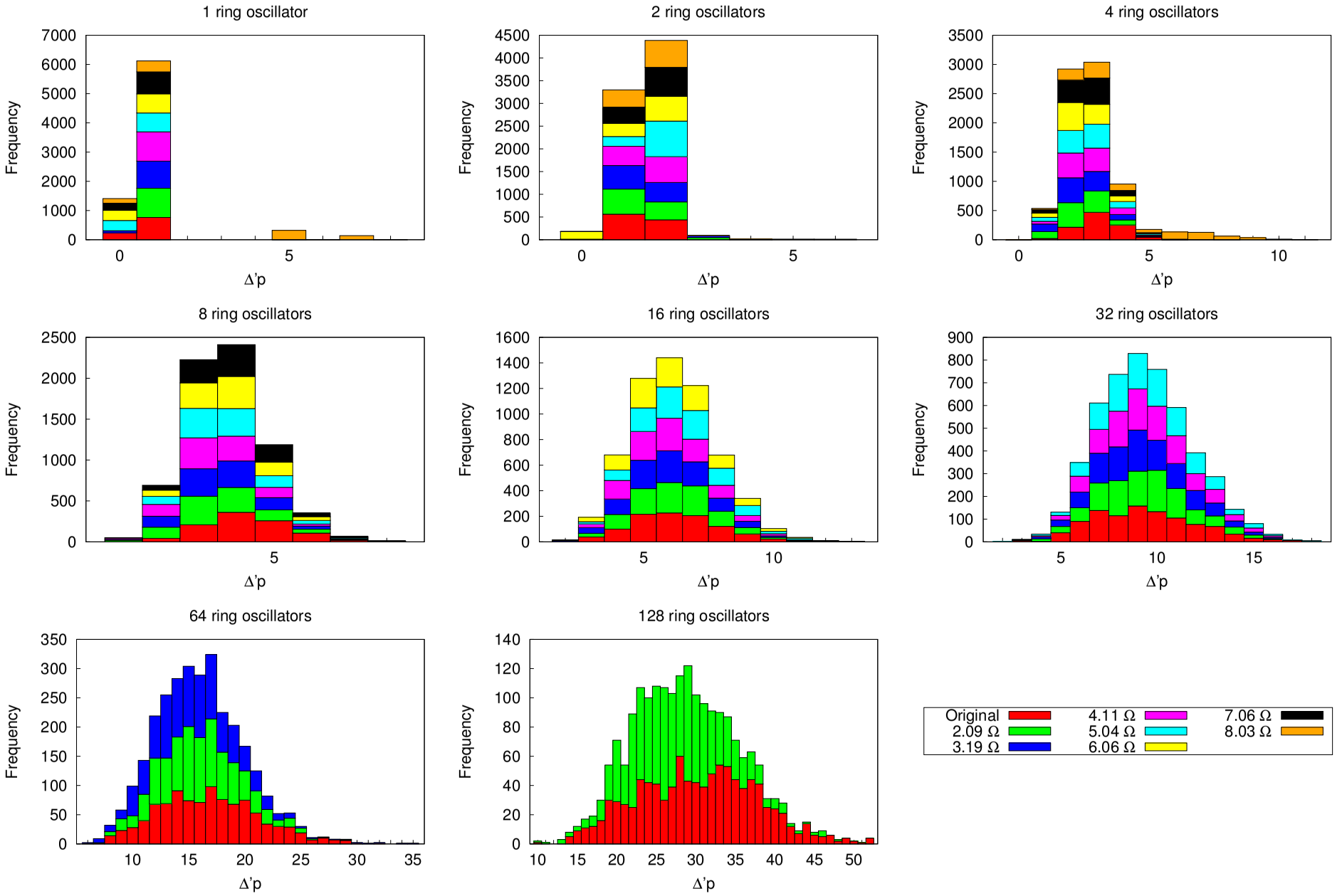


Figure 6.20: $\Delta' p$ distribution of 128-bit AES for different number of ring oscillators and shunt resistor values

than expected or by a slight temporary increase in the supply voltage. In configuration 3, we detect all the attacks for all RO values but 8, for which we have a false-negative rate of 5.7%. However, the attack is most of the time wrongly detected as high voltage for less than 16 ROs. This phenomenon is due to a lack of resolution of the power monitor for low RO values, which results in a noisy power monitor reading. Random noise coming from several sources (electronic noise, timing errors due to voltage drops) is reduced by increasing the number of ROs the power monitor reading is accumulated from. In pratice to ensure proper operation of our detection system in this crypto-system, we would use more than 16 ROs.

Table 6.2 shows the average total power consumption of the system performing the same 512-bit RSA encoding operation for different number of ROs. V_{CCINT} is fixed at 1.39V, $R_{EXT} = 2.02\ \Omega$ and the power consumption is computed as:

$$P_{FPGA} = (V_{CCINT} - V_{EXT})\frac{V_{EXT}}{R_{EXT}} \tag{6.41}$$

As expected, the power consumption increases with the number of ROs activated. A circuit with 16 RO and 32 RO attack detectors, which both have very good detection properties, consume respectively 52% and 71% more power than an unprotected circuit.

Hence, we can conclude that for a power monitor with more than 16 ROs, our attack detector can detect power attacks on RSA using relatively low shunt resistor values (superior to 2 Ω) with a false-positive rate less than 4%, a false-negative rate of 0% and a power overhead as low as 52%. Note that depending on the requirements of the crypto-system, the false-positive rate could be reduced by increasing $\Delta p^*_{ref,0}$ if we are willing to accept a possibly higher false-negative rate.

The RSA core has a power pattern that can be easily recognised with our power monitor as shown in Fig. 6.16. An RSA encryption consumes a relatively large amount of power and takes a few hundred thousand clock cycles. This is not the case for an AES encryption which only takes 8 clock cycles and consumes around 5 mW. For a 2.09 Ω shunt resistor, the power monitor cannot distinguish the AES power patter from the background noise and an attack on the AES core cannot be effectively detected. This is shown in Fig. 6.20 where the $\Delta' p$ distributions on the modified board cannot be distinguished from the $\Delta' p$ distribution on the original board.

The "AES" entry of Table 6.1 confirms this observation as we obtain false-negative rates as high as 99.97%. The next section presents an alternative detection strategy to address this issue.

6.5.5 Alternative detection strategy

Our alternative detection strategy is based on two properties:

1. If the power supply of the original board is properly filtered, $\Delta p_{ref,i}$ does not depend much on ΔI_i.

2. The attack detection logic consumes power that can be measured by the power monitor.

Let us first explain property 1. The measurement model of Fig. 6.1 does not take into account the capacitance of the supply voltage network. In an attack scenario most filtering capacitors located after R_{EXT} would be removed in order to obtain a clean power trace. However, the crypto-system is likely to be calibrated on a board with a properly filtered power supply. A properly filtered power supply has a combination of large, medium and small capacitors between V_{CCINT} and GND to filter out the noise on V_{INT}. These capacitors also help compensate for the voltage drops caused by large current transients due to the switching activity of the chip. Hence for a properly filtered power supply, equations 6.32 to 6.36 do not hold and we can assume that $\Delta p_{ref,i}$ does not depend on ΔI_i, that is $\Delta p_{ref,i}$ is constant for all i. Note that this does not change the principles behind our attack detector illustrated by Fig. 6.12. As a matter of fact, equations 6.32 to 6.36 do hold on the modified board where most filtering capacitors are removed (with the limitations discussed in section 6.5.3).

Property 2 is due to the fact that even if we remove all the filtering capacitors from the attack board, the board and the power measurement circuit still have an internal capacitance, which combined with R_{EXT} acts as a RC circuit. If the attack detector sampling period is several times shorter than the time constant of the equivalent RC circuit, the voltage drop created while powering it on can be measured. Basically, the power monitor measures its own power consumption, which adds up to the power consumption of the other cores. For the experiments of the previous section, this effect was ignored by waiting a few clock cycles before

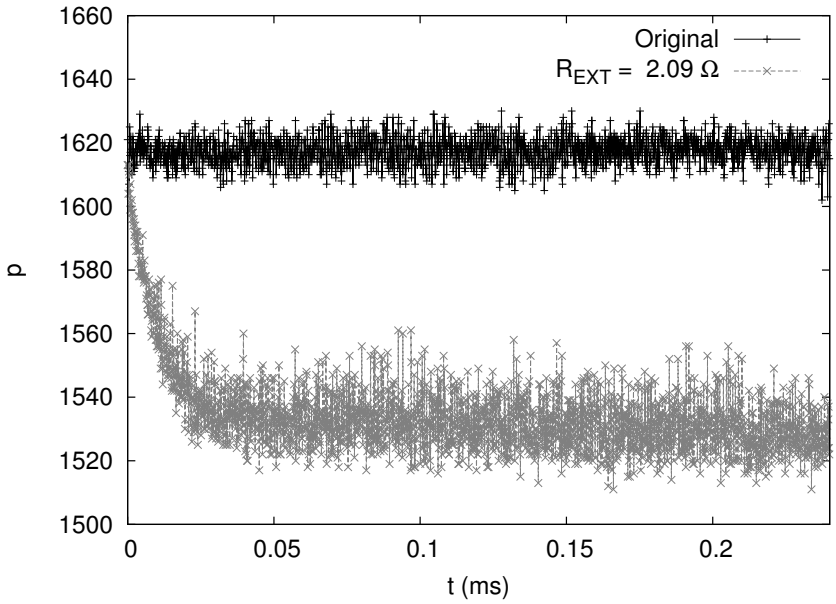


Figure 6.21: Power drop due to the attack detector for 64 ROs

starting recording the power variations of the system. In this section, we start recording the power variations of the system as soon as the attack detection starts.

To illustrate these two properties, Fig 6.21 shows the power monitor reading just after the attack detector logic starts on the original board and on the modified board with $R_{EXT} = 2.09\ \Omega$. We see that no voltage drop is observed on the original board whereas a large voltage drop is observed for the modified board. This observation confirms properties 1 and 2.

The application of the alternative detection strategy to AES is shown in the "AES+" entry of Table 6.1. For less than 16 ROs the number of ROs is not large enough to create a significant start-up drop, hence the attack cannot be reliability detected. For more than 32 ROs, this strategy enables the attack detector to detect the attacks using a 2.09 Ω shunt resistor with no false-negatives. Moreover, the false-positive rate on the original board is below 0.4%.

It is interesting to notice that under the assumption that the power supply of the original board is properly filtered, the power consumption of the attack detector itself can make the detection possible. Hence a slight variation on this idea would decouple the attack detection mechanism from the cryptographic cores. This would be achieved by starting and stopping the attack detector at proper times and only considering the start-up power of the detection logic. This strategy would reduce the energy overhead of the countermeasure. However, the intervals between each run of the detector should be short enough so that the attacker cannot

gain much information before being detected. It should also be non-predictable, so that the attacker cannot build a mechanism to bypass the measurement circuit when the detector is running. The intervals could be randomly chosen below a certain threshold.

6.6 Summary

In this chapter, we present a novel approach involving on-chip power monitoring to prevent power attacks on reconfigurable hardware. Our power monitor circuit is based on a network of evenly distributed ring-oscillators. Two power attack countermeasures are derived from this circuit. The first countermeasure is a general framework based on a closed-loop control system that keeps the power consumption of any FPGA implementation constant. Our implementation of the framework on a Spartan-6 LX150T takes 11 882 LUTs, that is 13% of the area available. Using a doubling-attack against 512-bit modular exponentiation as a case study, we demonstrate experimentally that the framework can prevent attacks performed at a bandwidth lower than half of its control rate. However using the framework as a hiding countermeasure is not feasible on current commercial FPGAs due to the limited speed of the control loop. A more realistic approach would be to implement our framework as a hard IP block on an FPGA dedicated to cryptography. The second countermeasure involves detecting the insertion of a shunt resistor-based power measurement circuit onto a device's power rail. Our implementation of the countermeasure on a Spartan-6 LX45 fits in 4 365 LUTs, that is 16% of the area available. For 512-bit RSA, with less than 71% power overhead we can detect a shunt resistor as low as 2 Ω with a maximum false-positive rate of 0.2% and no false-negatives. Our alternative detection strategy also enables the detection of power attacks on cores whose power variations are too small to be directly measured by the power monitor circuit.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we develop parametric designs for cryptographic applications. Our designs are especially useful for key generation and encryption in public-key cryptography. The design space is explored in terms of speed, area, power consumption and security. A technology-independent parametric model is developed to allow rapid optimisation under speed and area constraints of a general class of algorithms, which include some of our cryptographic applications. The security aspects are covered by two new power attack countermeasures based on on-chip power monitoring.

In chapter 3 we introduce new architectures for Montgomery multiplication and Montgomery exponentiation. The loop-carried dependency in the Montgomery algorithm prevents a parallel hardware implementation. In order to explore a large design space in terms of speed-area trade-off, our Montgomery multiplier architecture uses variable pipeline stages and variable serial replications. We create a modular exponentiation circuit based on our Montgomery multiplier architecture. Our circuit supports interleaving several exponentiations in the multiplier's pipeline in order to overcome the data dependencies in the square-and-multiply exponentiation algorithm. Our scalable Montgomery multiplier implemented on a Virtex-5 FPGA can achieve a throughput 4 times higher than the best existing hardware implementation and 13 times higher than an optimised software implementation. Our multiplier is also more efficient than existing designs in terms of Latency $\times$ Area and Area / Throughput. By interleaving multiple

exponentiations our exponentiator can run more than 5 times faster than optimised software and 2 times faster than the best modular exponentiation hardware reported in the literature. Only designs based on high-radix multipliers are more efficient than our exponentiator in terms of Time $\times$ Area product. Our Montgomery multiplier and exponentiation architectures scale well with the number of pipeline stages and replications. Hence better performance will be achieved on upcoming devices such as the Virtex-7, which will be faster and larger.

Chapter 4 generalises the idea used to scale our Montgomery multiplier. We present a coarse-grained method to automatically map a general class of algorithms consisting of a loop with loop dependencies carried from one iteration to the next to a parametric hardware design with pipelining and replication features. A technology-independent parametric model of the proposed design is developed to capture the variations of area and throughput with the number of pipeline stages and replications. Our model allows rapid optimisation of design parameters by a few pre-synthesis operations. We present an optimisation method based on the model and we evaluate our method on three different applications implemented on a Xilinx Spartan 6 XC6SLX45T FPGA: the Montgomery multiplier, the modular exponentiatior and a integer square root module. Our model facilitates design space exploration by quickly predicting the area taken by the designs with less than 5% of error and their maximum frequencies and throughputs with less than 22% of error. Our optimisation method is up to 96 times faster than a full search through the design space.

In chapter 5 we present the first hardware architecture for the NIST approved Lucas primality test and a novel architecture for the Miller-Rabin primality test. This chapter shows how our Montgomery multiplication and exponentiation circuit, which are needed for public-key encryption, can be reused for key generation. Our prime tester implementations are limited by data dependencies and cannot explore the full design space provided by our pipelining and replication approaches. However, our primality test architectures are still significantly more energy efficient than software both on FPGAs and ASIC. In fact the FPGA implementation of the Lucas test is up to 5 times more energy efficient than the software version. The 45 nm ASIC implementation is 420 times more energy efficient than the software version. The cell count and the power consumption of our ASIC implementations make them suitable for integration into a

general purpose CPU or an embedded system whereas our FPGA implementation would more likely benefit server application. Moreover, if we suppose that an attacker can obtain physical access to the crypto-system, implementing such cryptographic modules purely in hardware greatly reduces the vulnerability of a crypto-system when compared to a software implementation, which is usually run in a multi-tasking operating system. However, without proper protections cryptographic modules are still likely to be vulnerable to side-channel attacks.

Chapter 6 tackles the security dimension by presenting two novel countermeasures against power attacks on reconfigurable hardware. Our approach is based on on-chip power monitoring with a network of ring-oscillators. The first countermeasure is a general framework based on a closed-loop control system that keeps the power consumption of any FPGA implementation constant. Our implementation of the framework on a Spartan-6 LX150T takes 11 882 LUTs, that is 13% of the area available. Using a doubling-attack against 512-bit modular exponentiation as a case study, we demonstrate experimentally that the framework can prevent attacks performed at a bandwidth lower than half of its control rate. However using the framework as a hiding countermeasure is not directly feasible on current commercial FPGAs due to the limited speed of the control loop. The second countermeasure involves detecting the insertion of a shunt resistor-based power measurement circuit onto a device's power rail. To our knowledge, this is the first attempt at detecting power attacks targeting reconfigurable hardware. Our implementation of the countermeasure on a Spartan-6 LX45 fits in 4 365 LUTs, that is 16% of the area available. This countermeasure is lightweight and has a relatively low power overhead compared to existing masking and hiding countermeasures. For 512-bit RSA, with less than 71% power overhead we can detect a shunt resistor as low as 2 Ω with a maximum false-positive rate of 0.2% and no false-negatives. Our alternative detection strategy also enables the detection of power attacks on cores whose power variations are too small to be directly measured by the power monitor circuit.

The key results of this thesis are summarised in Table 7.1.

Table 7.1: Summary of the key results

Research area	Results
Parametric designs for public-key encryption (Chapter 3)	• Montgomery multiplier architecture using variable pipelining and serial replications: – 13 times faster on Virtex-5 than optimised software implementation on Intel Core 2 Duo running at 2.8GHz. – 4 times faster than the best existing hardware implementation. – more efficient than existing implementations in terms of Latency × Area and Area/Throughput • Modular exponentiation circuit based on the multiplier: – 5 times faster on Virtex-5 than the optimised software implementation on Intel Core 2 Duo running at 2.8GHz. – 2 times faster then the best existing hardware implementation. – more efficient than all the radix-2 implementations in terms of Time × Area product.
Rapid optimisation of parametric designs (Chapter 4)	• Coarse-grained method to automatically map a general class of algorithms to a parametric hardware design. • Model of the proposed design that captures the variations of area and throughput with the number of pipeline stages and replications: – area predicted with less than 5% of error. – frequency and throughput predicted with less than 22% of error. – up to 96 times faster than a full search through the design space.
Primality testing architectures (Chapter 5)	• Novel architecture for the Miller-Rabin primality test: – 1.6 times faster than optimised software on Intel Core 2 Duo running at 2.8GHz. – 49 times faster than other existing hardware implementations. – 2.9 times more efficient than existing implementations in terms of Time × Area product. • First hardware architecture for the NIST approved Lucas primality test: – FPGA implementation up to 5 times more energy efficient than optimised software on Intel Xeon W3505 at 2.53 GHz. – 65 nm ASIC implementation more than 8 times faster than the FPGA implementation and 39 times more energy efficient. – 45 nm ASIC implementation 420 times more energy efficient and 4 times faster than optimised software.
Power attack prevention and detection (Chapter 6)	• General framework based on a closed-loop control system that keeps the power consumption of any FPGA implementation constant: – 11 882 LUTs, 13% of area available on Spartan-6 LX150T – prevents attacks performed at a bandwidth lower than half of the control rate. • Circuit to detect the insertion of a shunt resistor-based power measurement circuit onto a device's power rail: – first attempt at detecting power attacks targeting reconfigurable hardware. – 4 365 LUTs, 16% of area available on Spartan-6 LX45. – less than 71% power overhead on 512-bit RSA. – shunt resistor as low as 2 Ω detected with a maximum false-positive rate of 0.2% and 0.4% for 512-bit RSA and 128-bit AES respectively, and no false-negatives.

7.2 Future work

The following sections highlight some limitations and present possible future extensions to the work presented in this thesis. These are also summarised in Table 7.2.

7.2.1 Very low area end of the design space

Our architectures presented in chapters 3 and 5 explore a large design space in terms of speed and area. However they are not very suitable for small low-end FPGAs as the current architecture of the Montgomery multiplier cannot reach the very low area end of the design space. As a matter of fact, a basic cell of our Montgomery multiplier already takes more than 11 000 LUTs as shown in section 3.5. This is due to our choice of a bitwise algorithm for Montgomery multiplication (algorithm 3.5) which works on the full lengths of B, N, D, S and C at each iteration, requiring a large amount of logic. To reduce the logic area, we would have to switch to one of the multiple-precision blockwise algorithm presented in [KABSK96] and used for instance in [FL05, OS08, Suz07, CELL10]. In these algorithms the Montgomery product is computed by processing the inputs word by word in two nested loops, usually leading to smaller logic. Our optimisation method presented in chapter 4 could also be extended to support blockwise processing.

7.2.2 Power consumption model

Our coarse-grained optimisation method presented in chapter 4 does not take into account power consumption. Our model could be extended to take into account power consumption by using an approach similar to the work presented in [BLC10]. In the `Pre-synthesis` phase (Fig. 4.3), we could estimate or measure the power of our basic cell and feed it to the model. The model would then be extended to estimate the power of the entire design based on the power consumption of a cell. One possibility for FPGAs is to assume that the static power is constant for a given device and to scale the dynamic power according to the maximum frequency f. Then we could estimate the dynamic power of the whole design as being proportional to the total number of basic cells in the design. The throughput optimisation process (equation 4.7) would

be extended to take into account a given power budget. Another interesting optimisation would be to minimise the area and the power consumption of the design given a minimum throughput goal. The static power for FPGAs is high and depends on the size of the chip. Hence such an optimisation would be equivalent to determining the smallest possible device with enough area to achieve a given throughput.

7.2.3 Speculative execution of Miller-Rabin tests

The Miller-Rabin architecture could be extended to support running several iterations of different tests simultaneously until a prime is found. This would allow interleaving modular multiplications from different tests in the Montgomery multiplier pipeline and would improve the throughput of the design.

7.2.4 Lucas instruction schedule

In chapter 5, the schedule of the Lucas sequence calculator instructions is fixed and may not be optimal for all the values of the parameters. In future work, we could explore different schedules and develop models and tools to find the optimum schedule given the bitwidth of the number under test, the number of replications of the Montgomery multiplier and the pipeline depths of the different sub-modules.

7.2.5 Other on-chip measurement methods

In chapter 6 we use a uniformly distributed network of ring oscillators to measure on-chip power. We could investigate local measurement methods by putting the ring oscillators close to the cryptographic cores. We would also like to study other on-chip measurement methods. In particular, we expect new generations of FPGAs to have better on-chip power measurement capabilities. For instance the Virtex-7 FPGA has an 12-bit ADC sampling at 1 MHz with on chip supply voltages sensors [Xil11a]. Such an ADC could be sufficient to make our attack detection framework work efficiently.

7.2.6 Faster constant power control loop

As stated in chapter 6, using the constant power framework as a hiding countermeasure is not feasible on current commercial FPGAs due to the limited speed of the control loop. A possible improvement to our constant framework would involve adding noise to the system. We could vary the power consumer command randomly in a range defined by the current control command calculated by the PID controller. This random change could occur faster than the control rate and therefore improve the bandwidth in which the countermeasure is effective. Future work could also investigate further the possibility of implementing such a framework as a hard IP block on an FPGA dedicated to cryptography.

7.2.7 Improved attack detection

Our attack detection framework is very efficient at detecting power attacks using a shunt-resistor based measurement circuit and would also benefit from a better on-chip power monitor. Using our alternative detection strategy we could investigate further the idea of decoupling the attack detection mechanism from the cryptographic cores by simultaneously using the power monitor as a power measurement and a power consumer circuit. Moreover we would also like to test our framework with shunt resistors smaller than 2 Ω.

However, our attack detection framework cannot detect electromagnetic attacks. Such attacks obtain a power profile of the chip by measuring the leaked electromagnetic radiations using contactless methods. Possible future work for the attack detection framework could investigate ways of detecting electromagnetic attacks on-chip. A possible direction is to study the suitability of on-chip antennas designed with unused FPGA interconnects [TCA11].

Table 7.2: Summary of limitations of this thesis and possible future work

Research area	Limitations and future work
Parametric designs for public-key encryption (Chapter 3)	• The Montgomery multiplier cannot reach the very low area end of the design space: – investigate blockwise algorithms.
Rapid optimisation of parametric designs (Chapter 4)	• The model does not take into account power consumption: – extend the model by using an approach similar to [BLC10].
Primality testing architectures (Chapter 5)	• The speed of the Miller-Rabin test is limited by data dependencies: – add hardware support to execute several iterations of different tests simultaneously in the multiplier's pipeline in order to increase throughput. • The Lucas calculator instruction schedule is fixed: – investigate other schedules. – create methods and tools to find the optimum schedule.
Power attack prevention and detection (Chapter 6)	• The global on-chip network of ring-oscillators may not be the best way of measuring on-chip power in all the cases: – investigate local measurement methods by putting the ring oscillators close to the cryptographic cores. – investigate on-chip ADC of new generations of FPGAs, especially for the attack detection framework. • The constant power framework is not efficient as a hiding countermeasure on current commercial FPGAs due to the limited speed of the control loop: – add noise to the system by varying the power consumer command randomly in a given range. – study the possibility of implementing such a framework as a hard IP block in an FPGA dedicated to cryptography. • An attacker might use shunt resistors smaller than 2 Ω: – test our attack detection framework with smaller shunt resistors. • The attack detection framework does not detect electromagnetic attacks: – investigate on-chip electromagnetic attack detection. – using a network of on-chip antennas designed with unused FPGA interconnects might be a starting point [TCA11].

Bibliography

[aes] Aoki laboratory webpage. `http://www.aoki.ecei.tohoku.ac.jp/crypto/`.

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in p. *Annals of Mathematics*, 2:781–793, 2002.

[Alt12] Altera. Logic array blocks and adaptive logic modules in Stratix V devices (sv51002-1.4), Jun 2012.

[AM08] Karl J. Astrom and Richard M Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. 2008. Princeton University Press, pp. 293-313.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. Cryptology ePrint Archive, Report 2009/317, 2009.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. Cryptology ePrint Archive, Report 2011/449, 2011.

[BLC09] Tobias Becker, Wayne Luk, and Peter Y. Cheung. Parametric design for reconfigurable software-defined radio. In *5th International Workshop on Reconfigurable Computing (ARC '09)*, pages 15–26, 2009.

[BLC10] Tobias Becker, Wayne Luk, and Peter Y. K. Cheung. Energy-aware optimisation for run-time reconfiguration. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 55–62, 2010.

[BP98] Kiran Bondalapati and Viktor K. Prasanna. Mapping loops onto reconfigurable architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, 1998.

[BP99] Thomas Blum and Christof Paar. Montgomery modular exponentiation on reconfigurable hardware. *IEEE Symposium on Computer Arithmetic*, pages 70–77, 1999.

[BP01] Thomas Blum and Christof Paar. High-radix Montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[BST02] V. Bunimov, M. Schimmler, and B. Tolg. A complexity-effective version of Montgomery's algorithm. In *Workshop on Complexity Effective Designs*, 2002.

[BW80] R. Baillie and S. S. Wagstaff, Jr. Lucas pseudoprimes. *Mathematics of Computation*, 35:1391–1417, 1980.

[BZ07] Karthik Baddam and Mark Zwolinski. Evaluation of dynamic voltage and frequency scaling as a differential power analysis countermeasure. In *VLSID '07*, pages 854 –862, 2007.

[CBLC04] R.C.C. Cheung, A. Brown, W. Luk, and P.Y.K. Cheung. A scalable hardware architecture for prime number validation. In *IEEE International Conference on Field-Programmable Technology*, pages 177–184, 2004.

[CELL10] Gary C. T. Chow, Ken Eguro, Wayne Luk, and Philip Leong. A Karatsuba-based Montgomery multiplier. In *2010 International Conference on Field Programmable Logic and Applications (FPL '10)*, pages 434–437, 2010.

[Cha08] Mark L. Chang. Device architecture. *Reconfigurable Computing: The theory and practice of FPGA-based computing*, pages 3–27, 2008.

[Cre] Jack W. Crenshaw. Integer square roots. `http://www.embedded.com/electronics-blogs/programmer-s-toolbox/4219659/Integer-Square-Roots`.

[DH76a] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, January 1976.

[DH76b] Whitfield Diffie and Martin E. Hellman. Multiuser cryptographic techniques. In *AFIPS National Computer Conference and Exposition*, pages 109–112, 1976.

[DH10] N. Dulay and M. Huth. The advanced encryption standard, 2010. Network Security Lecture Notes.

[DLP93] Ivan Damgard, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of Computation*, 61:177–194, 1993.

[DM02] Alan Daly and William Marnane. Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In *ACM Symposium on FPGAs*, pages 40–49, 2002.

[DMOPV07] E. De Mulder, S. B. Örs, B. Preneel, and I. Verbauwhede. Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems. *Computer & Electrical Engineering*, 33(5-6):367–382, 2007.

[EJCT00] Rolf Enzler, Tobias Jeger, Didier Cottet, and Gerhard Tröster. High-level area and performance estimation of hardware building blocks on FPGAs. In *FPL '00: 10th International Conference on Field-Programmable Logic and Applications*, pages 525–534, LNCS 1896, 2000.

[ES98] Shawna Meyer Eikenberry and Jonathan P. Sorenson. Efficient algorithms for computing the Jacobi symbol. *Journal of Symbolic Computation*, 26(4):509–523, 1998.

[FBCP10] J.J.L. Franco, E. Boemo, E. Castillo, and L. Parrilla. Ring oscillators as thermal sensors in FPGAs: Experiments in low voltage. In *Southern Programmable Logic Conference (SPL '10)*, pages 133 –137, 2010.

[FL05] John Fry and Martin Langhammer. RSA & Public key cryptography in FPGAs, 2005. Altera Technical Report.

[Fou98] Electronic Frontier Foundation. *Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design*. 1998.

[FSK10] Neils Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wyley Publishing, Inc., 2010.

[FV03] Pierre-Alain Fouque and Frédéric Valette. The doubling attack - why upwards is better than downwards. In *Cryptographic Hardware and Embedded Systems (CHES '03)*, pages 269–280, 2003.

[GM11] Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In *Cryptographic Hardware and Embedded Systems (CHES '11)*, pages 33–48, 2011.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems (CHES '01)*, pages 251–261. 2001.

[gmp] Gmp library manual. `http://gmplib.org/manual/`.

[Gor98] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[Gra10] Mentor Graphics. DK design suite datasheet, 2010.

[Gua03] D. J. Guan. Montgomery algorithm for modular multiplication, August 25, 2003. Lecture note.

[GWWT12] M. Gag, T. Wegner, A. Waschki, and D. Timmermann. Temperature and on-chip crosstalk measurement using ring oscillators in FPGA. In *15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 201 –204, 2012.

[HMA+10] N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Samir. Comparative power analysis of modular exponentiation algorithms. *IEEE Transactions on Computers*, 59(6):795–807, 2010.

[HNI+06] Naofumi Homma, Sei Nagashima, Yuichi Imai, Takafumi Aoki, and Akashi Satoh. High-resolution side-channel attack using phase-based waveform matching. In *Cryptographic Hardware and Embedded Systems (CHES '06)*, pages 187–200, 2006.

[HVG+07] M.C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *Computer*, 40(3):50 –57, march 2007.

[ica] Icarus Verilog website. `http://iverilog.icarus.com/`.

[ITLN09] Information Technology Laboratory (NIST). Digital Signature Standard, 2009. FIPS PUB 186-3.

[JQ96] M. Joye and J.-J. Quisquater. Efficient computation of full Lucas sequences. *Electronics Letters*, 32(6):537–538, Mar 1996.

[KABSK96] Cetin Kaya Koc, Tolga Acar, and Jr. Burton S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16:26–33, 1996.

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, 9:5–83,161–191, 1883.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)*, pages 789–789. 1999.

[KNRK06] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Compile-time area estimation for LUT-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 11:104–122, January 2006.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*, pages 104–113, 1996.

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA – a cost-optimized parallel code breaker. In *Cryptographic Hardware and Embedded Systems (CHES '06)*, pages 101–118, 2006.

[Lab04] RSA Labs. RSA Labs article on RSA security, 2004. `http://www.rsa.com/rsalabs/node.asp?id=2004`.

[LAD+98] W. Luk, P. Andreou, A. Derbyshire, F. Dupont-De-Dinechin, J. Rice, N. Shirazi, and D. Siganos. A reconfigurable engine for real-time video processing. In *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm*, pages 169–178. 1998.

[Liu11] Yvonne Liu. Using FPGAs to solve challenges in industrial applications, xilinx White Paper WP410 v1.0, November 2011.

[LMLEC10] Adrien Le Masle, Wayne Luk, Jared Eldredge, and Kris Carver. Parametric encryption hardware design. In *6th International Workshop on Applied Reconfigurable Computing (ARC '10)*, pages 68–79, 2010.

[Mas09] Adrien Le Masle. Parametric encryption hardware design, 2009. Master thesis, Imperial College London, Department of Computing.

[MD00] Zachary S. McGregor-Dorsey. Methods of primality testing. *MIT Undergraduate Journal of Mathematics*, 1, 2000.

[MDS99] Thomas Messerges, Ezzy Dabbish, and Robert Sloan. Power analysis attacks of modular exponentiation in smartcards. In *Cryptographic Hardware and Embedded Systems (CHES '99)*, pages 724–724. 1999.

[MMM03] C. Mclvor, M. McLoone, and J.V. McCanny. Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In *37th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 379–384, 2003.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power analysis attacks: Revealing the secrets of smart cards, 2007. Springer.

[MQF91] J.M. Mulder, N.T. Quach, and M.J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2):98 –106, Feb. 1991.

[myh] MyHDL website. `http://www.myhdl.org/`.

[NAPP+05] D. Narh Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler. Efficient hardware architectures for modular multiplication on FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 539–542, 2005.

[NdMM06] Nadia Nedjah and Luiza de Macedo Mourelle. A review of modular multiplication methods and respective hardware implementation. *Informatica*, 30(1):111–129, 2006.

[NIS] NIST. NIST webpage on approved block cipher mode. `http://csrc.nist.gov/groups/ST/toolkit/BCM/current_modes.html`.

[NIS99] NIST. Data Encryption Standard (DES), 1999. FIPS PUB 46-3.

[NIS01] NIST. Advanced Encryption Standard (AES), 2001. FIPS 197.

[NIS12] NIST. Recommendation for key management - part 1: General (revision 3), 2012. NIST Special Publication 800-57.

[OGOP04] S.B. Ors, F. Gurkaynak, E. Oswald, and B. Preneel. Power-analysis attack on an ASIC AES implementation. In *International Conference on Information Technology: Coding and Computing (ITCC '04)*, volume 2, pages 546–552, april 2004.

[OOP03] Siddika Ors, Elisabeth Oswald, and Bart Preneel. Power-analysis attacks on an FPGA: First experimental results. In *Cryptographic Hardware and Embedded Systems (CHES '03)*, pages 35–50. 2003.

[OS08] Ersin Oksuzoglu and Erkay Savas. Parametric, secure and compact implementation of RSA on FPGA. *International Conference on Reconfigurable Computing and FPGAs*, pages 391–396, 2008.

[OST06] DagArne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006*, volume 3860, pages 1–20. 2006.

[Par00] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. 2000. First Edition.

[PPV06] George Purdy, Carla Purdy, and Kiran Vedantam. Two binary algorithms for calculating the Jacobi symbol and a fast systolic implementation in hardware. In *49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '06)*, volume 1, pages 428 –432, Aug. 2006.

[PSJ80] C. Pomerance, J. L. Selfridge, and Samuel S. Wagstaff Jr. The pseudoprimes to 25e9. *Mathematics of Computation*, 35:1003–1026, 1980.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. 2001.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128 – 138, 1980.

[RM08] Chester Rebeiro and Debdeep Mukhpodhyay. Power attack resistant efficient FPGA architecture for Karatsuba multiplier. In *VLSID '08*, pages 706–711, 2008.

[Roh10] Pankaj Rohatgi. Protecting FPGAs from power analysis, 2010. `http://www.eetimes.com/design/programmable-logic/4199399/Protecting-FPGAs-from-power-analysis`.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[RWS11] Francesco Regazzoni, Yi Wang, and Francois-Xavier Standaert. FPGA implementations of the AES masked against power analysis attacks. In *International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2011)*, pages 56–66, 2011.

[SHD02] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. *ACM SIGPLAN Notices*, 37:165–176, May 2002.

[SJ08] P. Schumacher and P. Jha. Fast and accurate resource estimation of RTL-based designs targeting FPGAs. In *FPL '08: 18th International Conference on Field-Programmable Logic and Applications*, pages 59 –64, Sept. 2008.

[SMPQ06] F. Standaert, F. Mace, E. Peeters, and J. Quisquater. Updates on the security of FPGAs against power analysis attacks. In *2nd International Workshop on Applied Reconfigurable Computing (ARC '06)*, pages 335–346. 2006.

[SOQP04] Francois-Xavier Standaert, Siddika Berna Ors, Jean-Jacques Quisquater, and Bart Preneel. Power analysis attacks against FPGA implementations of the DES. In *2004 International Conference on Field Programmable Logic and Applications (FPL '04)*, pages 84–94. 2004.

[SP03] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In *2003 International Conference on Field Programmable Logic and Applications (FPL '03)*, pages 880–889. 2003.

[SS93] Jeffrey Shallit and Jonathan Sorenson. A binary algorithm for the Jacobi symbol. *ACM SIGSAM Bulletin*, 27(1):4–11, 1993.

[Sta11] William Stallings. *Cryptography and Network Security: Principles and Practice, Fifth Edition*. Pearson Education, 2011.

[STL04] H. Styles, D.B. Thomas, and W. Luk. Pipelining designs with loop-carried dependencies. In *International Conference on Field-Programmable Technology*, pages 255 – 262, 2004.

[Suz07] Daisuke Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Cryptographic Hardware and Embedded Systems (CHES '07)*, pages 272–288, 2007.

[TCA11] Abhay Tavaragiri, Jacob Couch, and Peter Athanas. Exploration of FPGA interconnect for the design of unconventional antennas. In *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pages 219–226, 2011.

[TCW+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *IEEE Computers and Digital Techniques*, 152(2):193–207, 2005.

[Tec11] Maxeler Technologies. Maxcompiler white paper, February 2011.

[TTL03] S.H. Tang, K.S. Tsui, and P.H.W. Leong. Modular exponentiation using parallel multipliers. In *IEEE International Conference on Field-Programmable Technology (FPT '03)*, pages 52–59, 2003.

[TV04] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Design, Automation, and Test in Europe Conference (DATE '04)*, 2004.

[Ved06] Kiran Vedantam. Hardware implementations for systolic computation of the Jacobi symbol, 2006. Master thesis, University of Cincinnati, 2006.

[ver] Verilator website. `http://www.veripool.org/wiki/verilator/Intro`.

[Wei97] M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Reconfigurable Architectures Workshop (RAW'97)*, April 1997.

[Xil10a] Xilinx. Spartan-6 FPGA configurable logic block, User Guide UG384 v1.1, February 2010.

[Xil10b] Xilinx. Virtex-6 FPGA System Monitor, User Guide UG370 (v1.1), June 2010.

[Xil11a] Xilinx. 7 series FPGAs XADC dual 12-bit 1MSPS analog-to-digital converter, User Guide UG480 v1.1, March 2011.

[Xil11b] Xilinx. Spartan-6 family overview, User Guide DS160 v2.0, October 2011.

[Xil12a] Xilinx. Vivado Design Suite Product Brief, 2012.

[Xil12b] Xilinx. Command line tools User Guide, UG628 v14.1, April 2012.

[Xil12c] Xilinx. Virtex-6 FPGA Data Sheet: DC and Switching Characteristics, User Guide DS152 v3.4, January 2012.

[Xil12d] Xilinx. 7 series FPGAs overview, User Guide DS180 v1.11, May 2012.

[YLS+08] S. Yusuf, W. Luk, M. Sloman, N. Dulay, E.C. Lupu, and G. Brown. Reconfigurable architecture for network flow analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):57–65, Jan. 2008.

[YS07] Pengyuan Yu and Patrick Schaumont. Secure FPGA circuits using controlled placement and routing. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pages 45–50, 2007.

[YWV+04] Shengqi Yang, W. Wolf, N. Vijaykrishnan, D.N. Serpanos, and Yuan Xie. Power attack resistant cryptosystem design: a dynamic voltage and frequency switching approach. In *Design, Automation, and Test in Europe Conference (DATE '04)*, pages 64–69, 2004.

Appendix A

Proportional-Integral-Derivative (PID) Control

This appendix describes in more detail the PID controller principles and the techniques used to come up with the power controller presented in section 6.4.3. The following sections are mostly based on [AM08].

A.1 Algorithm

PID control is heavily used in process control. Its simple version is described by the following equation:

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{A.1}$$

$e = r - y$ is the control error, where y is the measured process variable and r is the reference variable or setpoint. u is the control signal. The control signal is the sum of three terms:

- $Ke(t)$: the proportional (P) term
- $\frac{K}{T_i}\int_0^t e(\tau)d\tau$: the integral (I) term
- $KT_d\frac{de(t)}{dt}$: the derivative (D) term

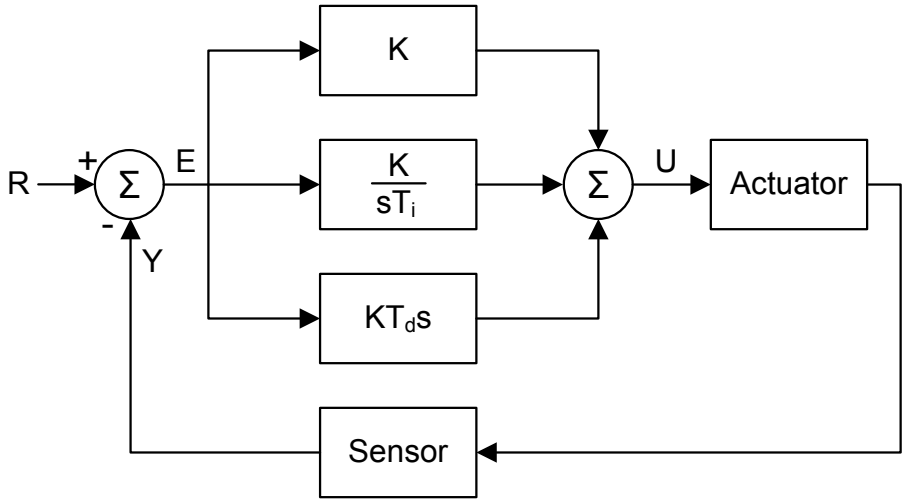


Figure A.1: Simple PID control loop

The controller parameters are the proportional gain K, the integral time T_i and the derivative time T_d. The integral, proportional and derivative terms are often interpreted as control actions based on the past, the present and the future.

In the Laplace domain, the PID control equation is written as:

$$\frac{U(s)}{E(s)} = K\left(1 + \frac{1}{sT_i} + sT_d\right) \tag{A.2}$$

The diagram of the PID control loop is shown in Fig. A.1.

In proportional control (P term only) there is always a steady state error, that is the process variable does not reach the setpoint in the steady state:

$$\lim_{t\to\infty} e(t) = E > 0 \tag{A.3}$$

The error decreases when the gain K increases but if the gain is too high the system can become unstable and oscillate.

Adding the integral term accelerates the movement of the system towards the setpoint and eliminate the steady state error. The strength of integral action increases with decreasing integral time T_i. The tendency to oscillations increases when T_i decreases. The integral term is proportional to accumulated errors from the past and can cause the process variable to overshoot the setpoint.

The derivative term slows down the response of the controller and is used to reduce the overshot due to the integral term. Damping initially increases with T_d but decreases again when T_d becomes too large. The derivative term is highly sensitive to noise and can cause the system to become unstable.

Several techniques are used to improve the simple PID control equation and obtain a good PID controller. In the following section, we present the three techniques we use in our PID controller implementation: setpoint weighting, filtering and anti-windup.

A.2 Setpoint weighting

According to equation A.1, a step change in the setpoint r creates an impulse in the control signal $u(t)$. To avoid large transients in the control signal, derivative action is not applied to the reference signal. We can also reduce the overshot by letting the proportional action act only on a fraction of the reference signal. The PID equation therefore becomes:

$$u(t) = K\left(br(t) - y(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau - Td\frac{dy(t)}{dt}\right) \tag{A.4}$$

b is an additional parameter used to control the overshot created by setpoint changes.

A.3 Filtering

Filtering aims at reducing the sensitivity of the derivative term to noise by limiting the high frequency gain of the derivative term. We can implement the derivative term as:

$$D(s) = -\frac{sKT_d}{1 + sT_d/N}Y \tag{A.5}$$

This amounts to filtering the ideal derivative sT_d by a first-order system with time constant T_d/N. The gain is now limited to KN. N is usually chosen between 8 and 20 depending on the application.

A.4 Anti-windup

In a control system, the actuators have limitations. For example, our power consumer presented in section 6.4.2 can only consume a finite amount of power which is basically proportional to the number of horizontal/vertical power consuming wire pairs, also called C_{max} in section 6.4.3. When the control signal reaches the actuator limit, the control saturates and the systems starts running in open-loop. If an integral action is used, the error will keep on being integrated, leading to a very large integral term. This phenomenon is called windup and may lead to large transients in the control system.

A solution to this problem is to use back-calculation. The idea is to create an extra feedback path when the output saturates in order to dynamically reset the integrator with a constant T_t. This is shown in Fig. A.2 which integrates all the three technique used to improve the PID controller. A model of the actuator saturation is used to form an error signal e_s between the output of the controller v and the actuator output u. This error signal goes to the input of the integrator with a gain $1/T_t$. When there is no saturation, e_s is equal to zero and the extra feedback loop has no effect. When the actuator saturates, the feedback path going through the sensor is broken because the output of the actuator remains constant. However the feedback path around the integrator prevents it from winding up. In fact, e_s compensates for the error e and v quickly settles outside the saturation limits.

The tracking time T_t should lie in between T_d and T_i. In our controller we set $T_t = \sqrt{T_i T_d}$ as suggested in [AM08].

A.5 Discretisation of the PID control algorithm

The PID control algorithm needs to be discretised in order to be implemented on hardware. We note h the sampling period. k represents a sampling instant. The proportional term becomes:

$$P[k] = K(br[k] - y[k]) \tag{A.6}$$

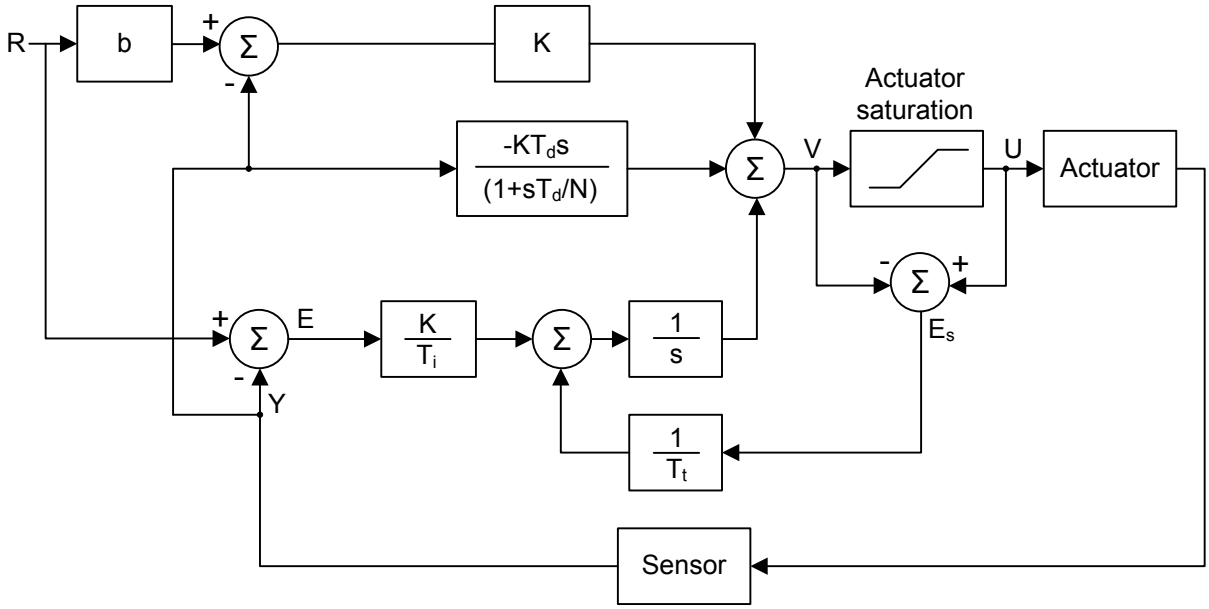


Figure A.2: PID control loop with setpoint weighting, filtering and anti-windup

From the integral term with anti-windup:

$$I(t) = K\frac{1}{T_i}\int_0^t e(\tau)d\tau + \frac{1}{T_t}\int_0^t e_s(\tau)d\tau \tag{A.7}$$

we derive:

$$\frac{dI}{dt} = \frac{K}{T_i}e + \frac{1}{T_t}e_s \tag{A.8}$$

which can be approximated as:

$$\frac{I[k+1] - I[k]}{h} = \frac{K}{T_i}e[k] + \frac{1}{T_t}e_s[k] \tag{A.9}$$

Hence:

$$I[k+1] = I[k] + \frac{Kh}{T_i}e[k] + \frac{h}{T_t}e_s[k] \tag{A.10}$$

Converting equation A.5 back to the time domain, we have the following relation for the

derivative term:

$$\frac{T_d}{N}\frac{dD}{dt} + D = -KT_d\frac{dy}{dt} \tag{A.11}$$

which can be approximated as:

$$\frac{T_d}{N}\frac{D[k] - D[k-1]}{h} + D[k] = -KT_d\frac{y[k] - y[k-1]}{h} \tag{A.12}$$

Hence:

$$D[k] = \frac{T_d}{T_d + Nh}D[k-1] - \frac{KT_dN}{T_d + Nh}(y[k] - y[k-1]) \tag{A.13}$$

To summarise, our discrete PID controller is defined by the following set of equations:

$$\begin{aligned}
P[k] &= K_p(br[k] - y[k]) \\
I[k] &= I[k-1] + K_i(r[k-1] - y[k-1]) + K_t(u[k-1] - v[k-1]) \\
D[k] &= K_{d0}D[k-1] - K_{d1}(y[k] - y[k-1]) \\
v[k] &= P[k] + I[k] + D[k] \\
u[k] &= sat(v[k], 0, C_{max})
\end{aligned} \tag{A.14}$$

where:

$$K_p = K \qquad K_i = \frac{Kh}{T_i} \tag{A.15}$$

$$K_{d0} = \frac{T_d}{T_d + Nh} \qquad K_{d1} = \frac{KT_dN}{T_d + Nh} \tag{A.16}$$

$$K_t = \frac{h}{T_t} \tag{A.17}$$

This leads to Algorithm 6.1 presented in section 6.4.3. Note that in our constant power framework, we do not change the setpoint when the PID controller is running. Hence $r[k] = setpoint$ is constant and we set $b = 1$.

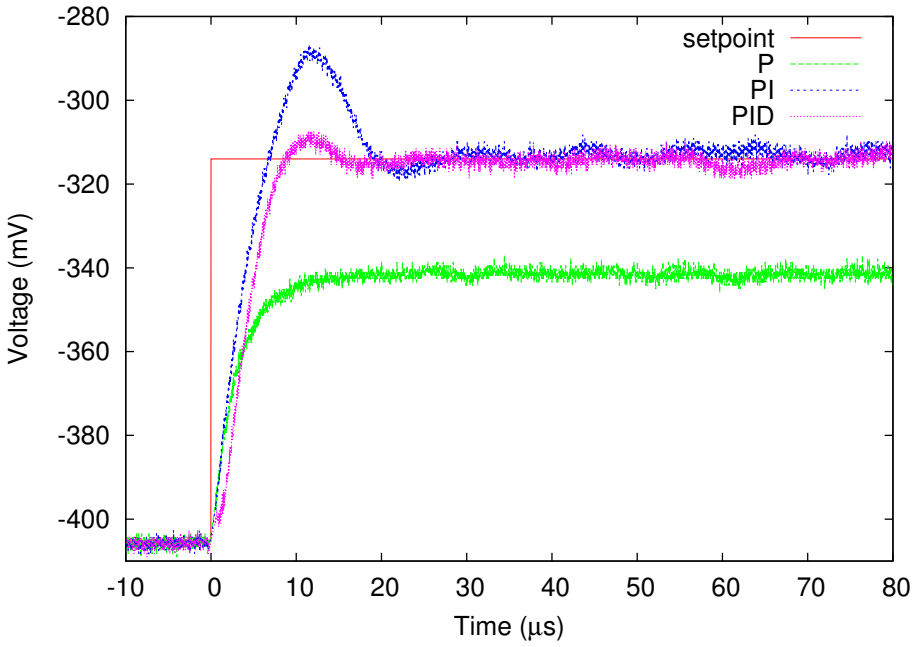


Figure A.3: Comparison of P, PI and PID controllers in our constant power framework

A.6 PID control in our constant framework

Fig. A.3 shows the step response of properly tuned P, PI and PID controllers in our constant power framework. The PID controller uses set-point weighting, filtering and anti-windup. The traces are obtained using an implementation of the constant power framework with a power monitor consisting of 128 3-inverter ROs on the Pico E-101 board described in section 6.5.3. This is a cut down version of the power monitor presented in section 6.4.4, which has 576 1-inverter ROs.

The P controller does not reach the setpoint due to the steady state error. Adding the integral action eliminates the steady state error but the PI controller shows a pronounced overshot. The overshot is reduced by adding the derivative action. However the PID controller is slightly slower to react to the setpoint change. These behaviours are conformed with the equations described in the previous section.

Fig. A.4 to A.7 show the beginning of the power trace of a 512-bit RSA computation for different controller configurations. Without control (Fig. A.4), the start of the computation as well as the different multiplication patterns are clearly visible. The P controller (Fig. A.5) reduces the voltage drop due to the RSA computation but the multiplication patterns are still visible. Using a PI controller (Fig. A.6) makes the start of the computation as well as

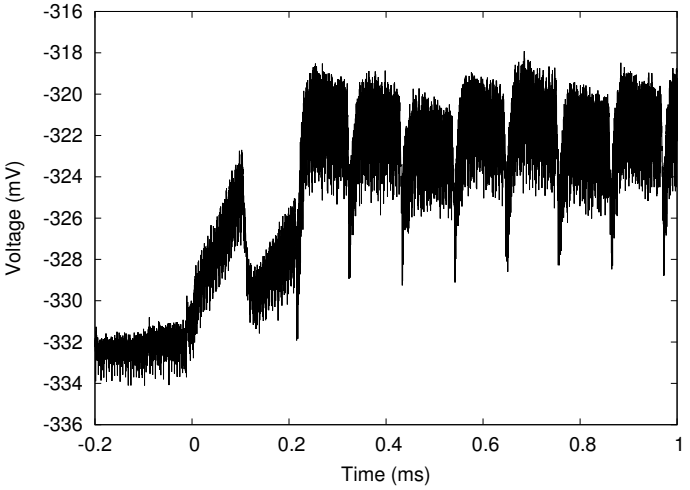


Figure A.4: RSA power trace without control

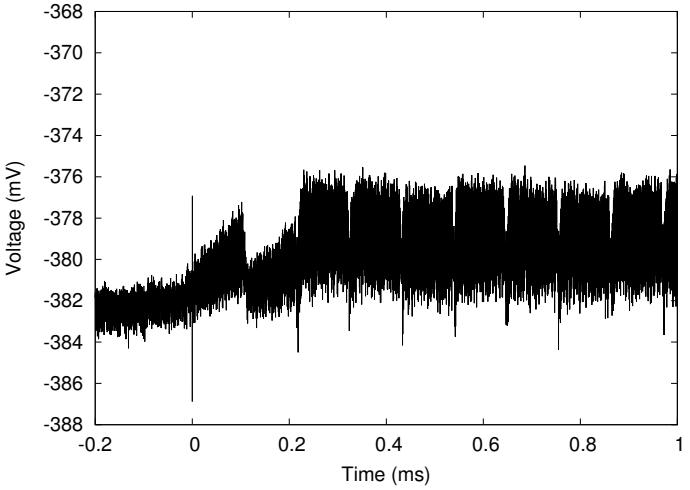


Figure A.5: RSA power trace with P controller

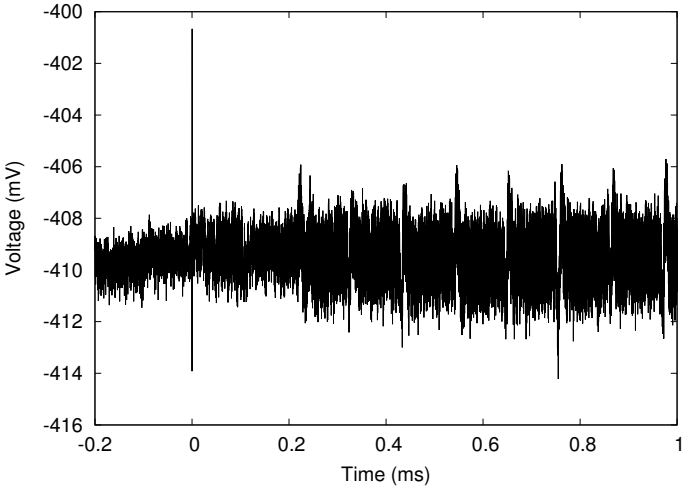


Figure A.6: RSA power trace with PI controller

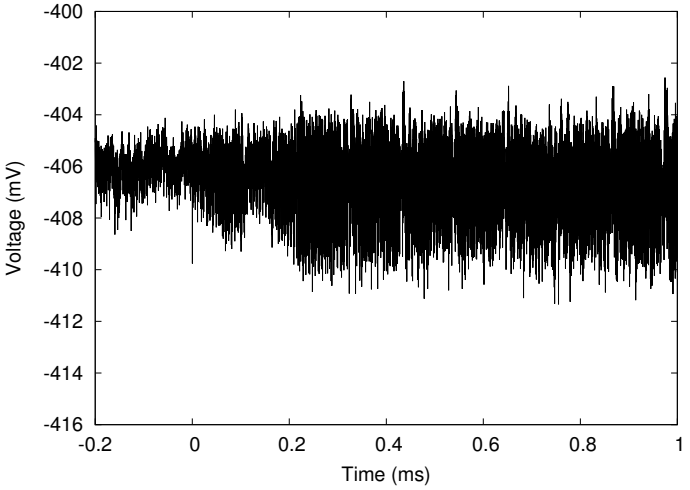


Figure A.7: RSA power trace with PID controller

the multiplication patterns harder to distinguish. Note that the spike at $t = 0$ is due to the trigger from the FPGA that is set specifically for our measurements. In a real-world attack environment, this trigger would not be present. Adding the derivative action (Fig. A.7) does not seem to improve on the PI version at first glance. In fact, the noise is slightly amplified compared to the PI controller. This noise amplification phenomenon could help the attacker realise that an operation has just started on the device. However, it also makes a prospective attack harder as more power traces would be needed to reduce noise, so that enough information can be extracted from this run. This justifies using a PID controller rather than a PI controller in our constant power framework.

Appendix B

Power Attacks on Reconfigurable Hardware

This appendix details two different power attacks that we perform on FPGAs. The first attack is a doubling attack against RSA which is used to evaluate our constant power framework in section 6.4.4. The second attack is a differential power attack against DES.

B.1 Comparative power attack against RSA

B.1.1 Principles

Comparative power attacks are a mixture between simple and differential power attacks. The main idea is to compare two segments of a single or two power traces in order to determine whether some pairs of intermediate values are the same. This terminology was introduced in [HMA+10]. In this section we describe our implementation of the doubling attack against RSA, which is a particular comparative power attack first described in [FV03]. This attack uses two inputs X and X^2 in order to create collisions between adjacent squaring operations in the left-to-right binary exponentiation algorithm shown in Algorithm 6.2.

According to Algorithm 6.2, at iteration i if the secret exponent bit e_i is zero only a squaring operation is performed. If the secret exponent bit e_i is one, a squaring operation followed by a multiplication operation are performed. We compare the two power traces obtained respectively

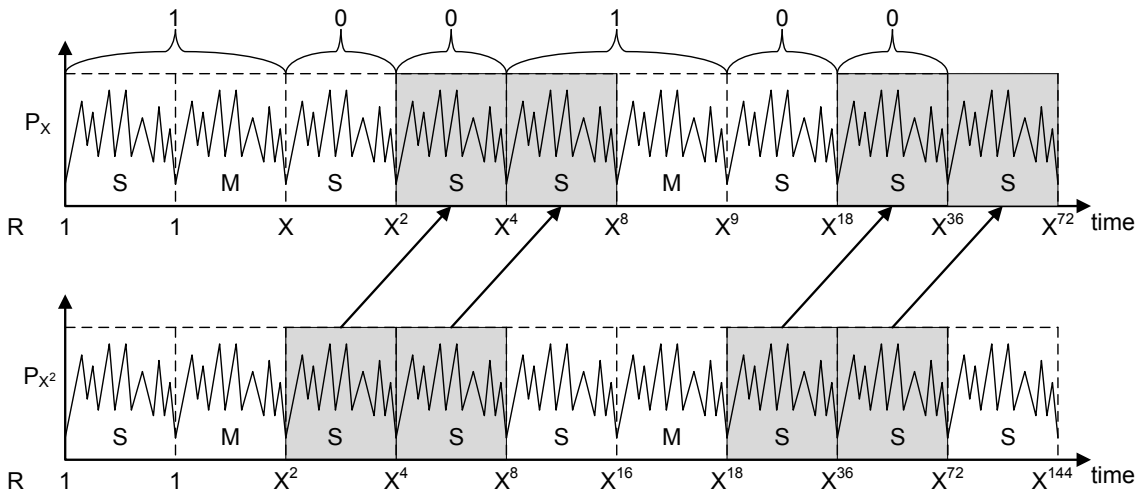


Figure B.1: Doubling attack

for inputs X (trace P_X) and X^2 (trace P_{X^2}). If the power trace P_X at iteration $i+1$ is identical to the power trace P_{X^2} at iteration i, the operation performed at iteration i is a squaring and key bit $e_i = 0$. Otherwise the operation is a multiplication and $e_i = 1$. Fig. B.1 shows an example of doubling attack for a secret exponent:

$$E = e_n e_{n-1} e_{n-2} e_{n-3} e_{n-4} e_{n-5} ... = 100100...$$

We observe collisions at iterations $(n-1)$, $(n-2)$, $(n-4)$ and $(n-5)$ corresponding to the squaring operations.

B.1.2 Phase Only Correlation (POC) waveform matching

Phase Only Correlation (POC) waveform matching is an effective method to compare two waveforms [HNI+06]. It exhibits much higher discrimination than other matching techniques based on the correlation function and can evaluate the displacement between two waveforms created by trigger delays or deliberately introduced random delays.

Let $f(n)$ and $g(n)$ be two waveforms of length $2M+1$ where $-M \leq n \leq M$. $F(k)$ and

$G(k)$ are the discrete Fourier transforms of $f(n)$ and $g(n)$:

$$F(k) = \sum_{n=-M}^{M} f(n) e^{-j\frac{2\pi kn}{N}} \tag{B.1}$$

$$G(k) = \sum_{n=-M}^{M} g(n) e^{-j\frac{2\pi kn}{N}} \tag{B.2}$$

The cross-phase spectrum $R_{FG}(k)$ is defined as:

$$R_{FG}(k) = \frac{F(k)\overline{G(k)}}{\left|F(k)\overline{G(k)}\right|} \tag{B.3}$$

$\overline{G(k)}$ is the complex conjugate of $G(k)$. The POC function $r_{fg}(n)$ is the inverse discrete Fourier transform of R_{FG}:

$$r_{fg}(n) = \frac{1}{N} \sum_{k=-M}^{M} R_{FG}(k) e^{j\frac{2\pi kn}{N}} \tag{B.4}$$

If the two waveforms are similar minus a given displacement, the POC function gives a distinct sharp peak located at the displacement δ between the two waveforms.

The sharpness of the correlation peak is evaluated using the Peak-to-Sidelobe Ratio $PSR = (peak - mean)/std$. In our case $PSR = (max - mean)/std$, where max is the maximum, $mean$ is the mean and std is the standard deviation of the POC function.

B.1.3 Implementation

We implement the attack on an unprotected 512-bit left-to-right binary exponentiation core. A Python script controls the trigger of the FPGA board and our Tektronix MSO 2024 oscilloscope. On both our Spartan-6 LX150T and LX45 FPGA boards, the full 512-bit key can be recovered in less than two days. Most of the time is spent acquiring the power traces. As a matter of fact, for each acquisition we average 30 power traces in order to reduce the measurement noise. Each bit requires two power trace subsets: one from P_X and one from P_{X^2}. Note that our oscilloscope does not enable us to extract the subsets from a single acquisition of P_X and P_{X^2} with enough resolution to perform the POC matching. Hence to recover the full 512-bit key,

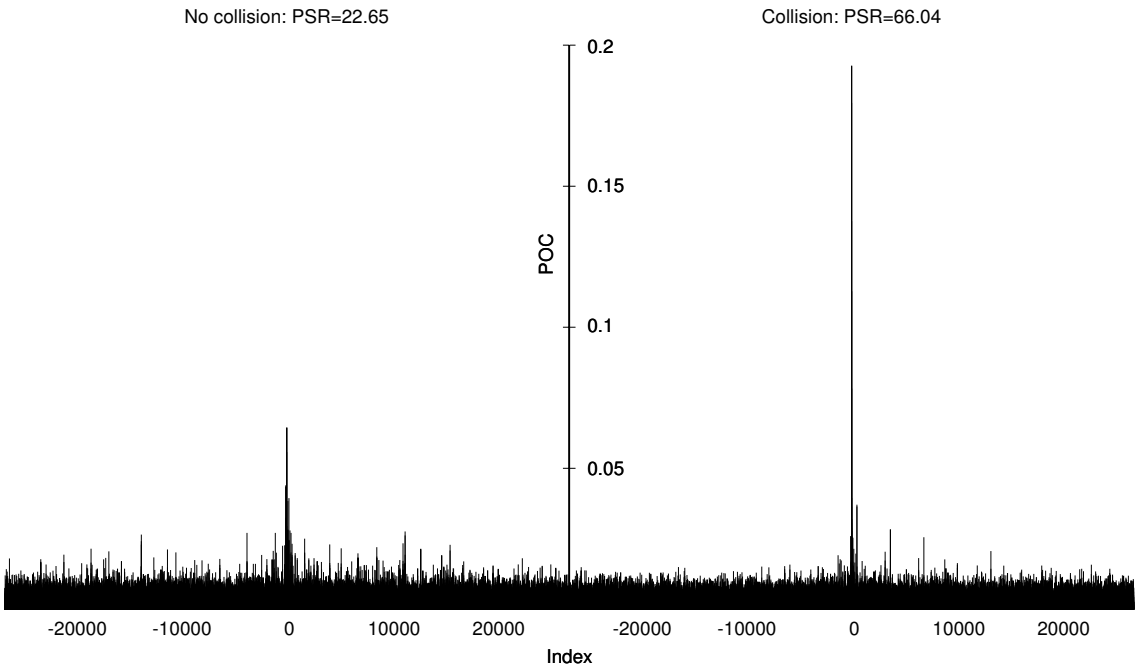


Figure B.2: Comparison of the POC functions

we actually need two acquisitions per bit, that is 30720 traces.

Fig B.2 compares the POC functions for a key bit of 1, where no collision occurs and a key bit of 0, where there is a collision between the corresponding square operations in P_X and P_{X^2}. When no collision occurs, there is a small peak at index 0 and the PSR is 22.65. As a matter of fact, even if the intermediate inputs are different, the traces still present some similarities as they both correspond to a multiplication operation. In the case of a collision, the peak is much more pronounced with a PSR 3 times larger than when no collision occurs. Hence the collisions can be easily detected using the POC function.

B.2 Differential power attack against DES

In this section, we describe the DPA we mount against DES in order to evaluate our modified FPGA boards. The attack is based on [SOQP04]. The DES algorithm is described in detail in section 2.2.3.

B.2.1 Principles

The attack is a ciphertext-only correlation power analysis attack against DES encryption using the Hamming distance between L_{15} and L_{16} (see left part of Fig. 2.7) as the weight model. For a ciphertext C_m, we have:

$$\{R_{16}, L_{16}\} = IP(C_m) \tag{B.5}$$

where IP is the initial permutation matrix.

We aim at recovering the last 48-bit round key K_{16}. The full 56-bit master key can then be recovered by inverting the key schedule and performing exhaustive search on the missing 8 bits. $K_{16} = k_1...k_8$ is divided into 8 words of 6 bits. Word k_l corresponds to the inputs of S-box S_l. According to Fig. 2.7 and Fig. 2.9, the output of S_l in the last round can be recovered as a function of R_{16} and L_{15}:

$$Sout_l = P^{-1}(R_{16} \oplus L_{15})[4(l-1) : 4l-1] \tag{B.6}$$

$$= P^{-1}(R_{16})[4(l-1) : 4l-1] \oplus P^{-1}(L_{15})[4(l-1) : 4l-1] \tag{B.7}$$

Here the numbers are represented LSB first and $A[j : k]$ denotes the slice of A from bit number j to bit number k. $Sout_l$ can also be written as a function of K_{16} and $R_{15} = L_{16}$:

$$Sout_l = S_l(E(L_{16})[6(l-1) : 6l-1] \oplus K_{16}[6(l-1) : 6l-1]) \tag{B.8}$$

$$= S_l(E(L_{16})[6(l-1) : 6l-1] \oplus k_l) \tag{B.9}$$

Hence we have:

$$P^{-1}(L_{15})[4(l-1) : 4l-1] = P^{-1}(R_{16})[4(l-1) : 4l-1] \oplus S_i(E(L_{16})[6(l-1) : 6l-1] \oplus k_l) \tag{B.10}$$

For subkey k_l, we define the selection function:

$$D_l(k_l, C_m) = H(P^{-1}(L_{15})[4(l-1):4l-1] \oplus P^{-1}(L_{16})[4(l-1):4l-1]) \tag{B.11}$$

This corresponds to the number of bit flips due to the subkey k_l in the register holding L. Note that the selection function depends on k_l and C_m through equations B.10 and B.5.

The subkey k_l can take $2^6 = 64$ values called key hypotheses. We note $h_{m,i}$ the estimated power consumption for ciphertext C_m and key hypothesis i. $h_{m,i}$ is therefore:

$$h_{m,i} = D_l(i, C_m) \tag{B.12}$$

The dependence of the estimated power consumption in l is not reflected in our notation $h_{m,i}$ to simplify the readability of the equations.

We capture M different ciphertexts together with the corresponding DES power traces of length T. We note p_m, j the recorder value of the power trace for message m at time j. We compute the $64 \times T$ correlation matrix R with its elements $r_{i,j}$ defined as:

$$r_{i,j} = \frac{\sum_{m=1}^{M}(h_{m,i} - \overline{h}_i).(p_{m,j} - \overline{p}_j)}{\sqrt{\sum_{m=1}^{M}(h_{m,i} - \overline{h}_i)^2 . \sum_{m=1}^{M}(p_{m,j} - \overline{p}_j)^2}} \tag{B.13}$$

$\overline{h}_i$ is the mean value of the vector $\mathbf{h}_i$ of estimated power consumptions for all messages under key hypothesis i. $\overline{p}_j$ is the mean value of the vector $\mathbf{p}_j$ of recorded power consumptions for all messages at time j. The most likely value for the secret subkey k_l is simply determined by identifying the row that contains the extreme value of matrix R.

B.2.2 Implementation

We implement the attack on the modified Spartan-6 Pico E-101 board. To simplify the attack we use 16 DES cores processing the same inputs in parallel. We perform $M = 4096$ encryptions with different plaintexts and we store the corresponding ciphertexts. Note that if we wanted to run an attack on a single DES core, we would require many more ciphertexts for the attack to be successful. For each encryption, the power trace is averaged over 10 runs in order to

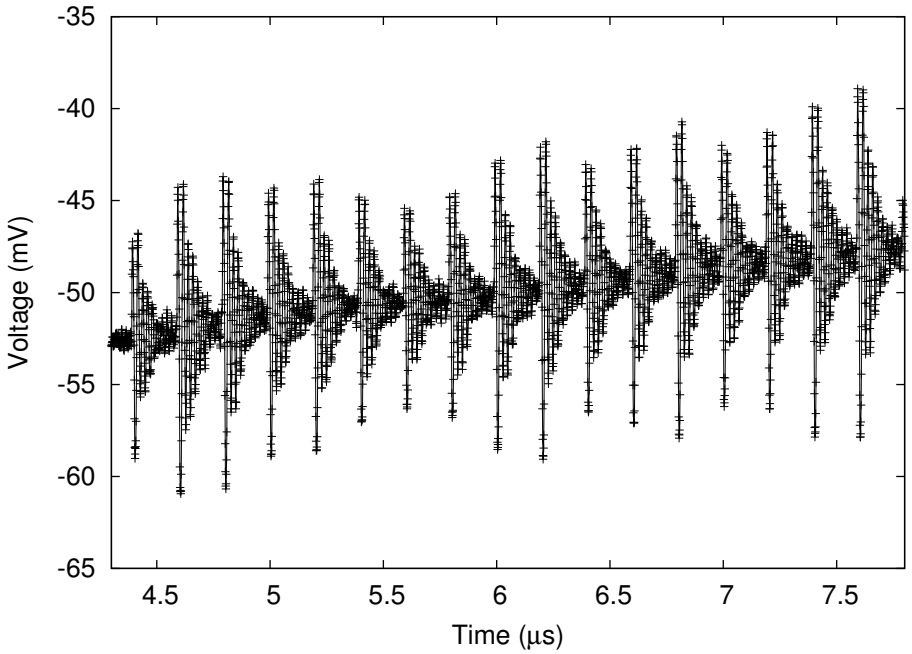


Figure B.3: Power trace of a DES encryption

reduce noise. The power traces have length $T = 4000$. Fig. B.3 shows the power trace of a DES encryption. The first spike in the trace corresponds to the initialisation of the different registers and the 16 following spikes correspond to the 16 DES rounds.

Fig. B.4 shows the correlation for the 64 key hypotheses on each of the subkeys k_1 to k_8. For each subkey, the correlation spike for the correct key hypothesis (in red) can clearly be distinguished from the correlations for the other key hypotheses (in green). A full 64-bit key can be recovered in less than 2 days. Like the attack against RSA, most of the time is spent acquiring waveforms.

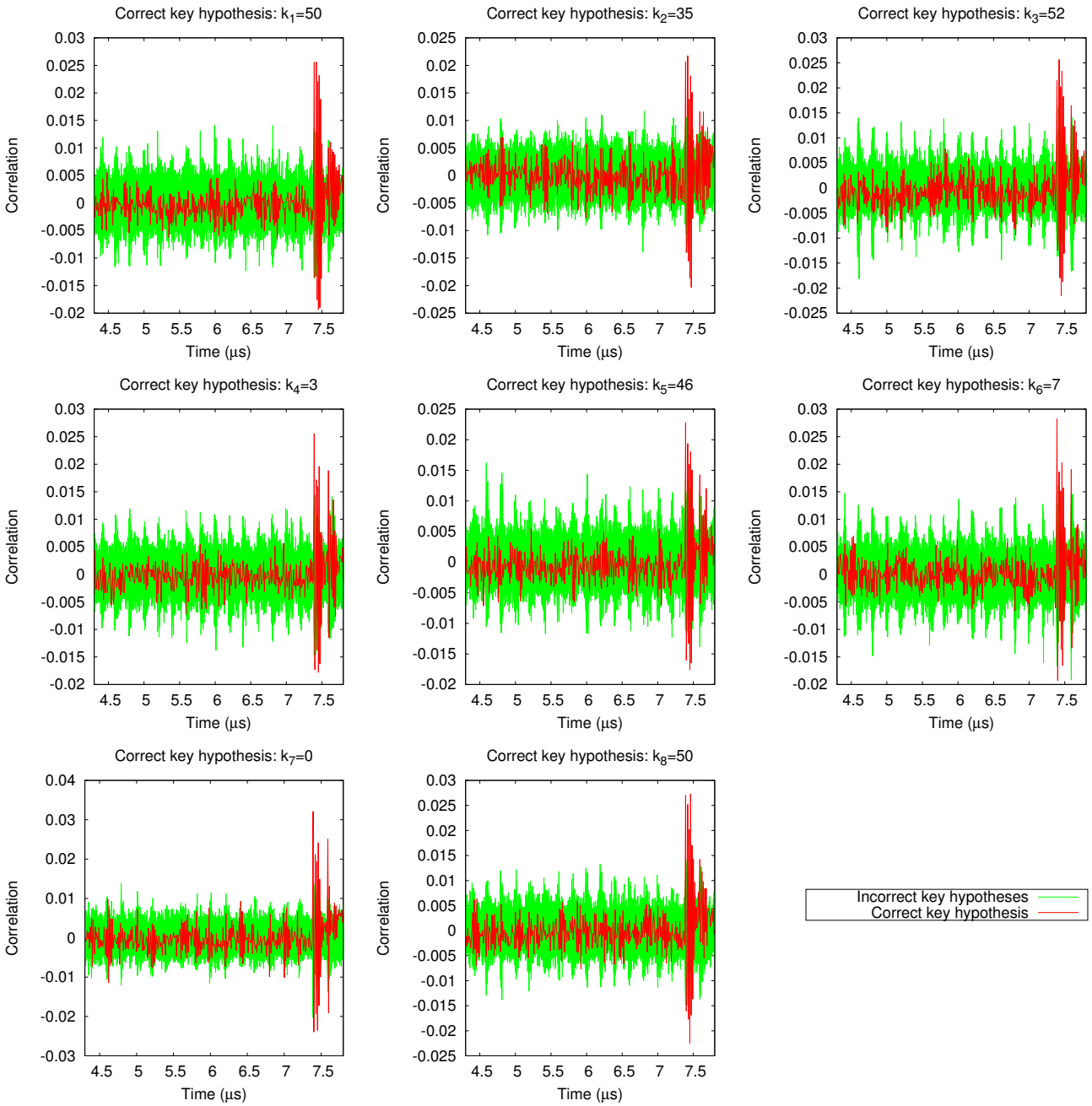


Figure B.4: Result of the DPA attack against DES

Appendix C

Design Testing

This appendix briefly describes the methodology and tools used to test our Verilog designs.

C.1 Unit testing

All our Verilog modules are individually tested. Our test vectors consist of a series of random inputs together with the extremes of the input domain (boundary conditions). We write our testbenches in Python using the MyHDL package [myh]. The Verilog modules are compiled using Icarus Verilog [ica]. The testbenches and the Verilog modules are connected using the co-simulation feature of MyHDL.

As an example, Fig. C.1 shows the Verilog code for the carry-save adder used in the Montgomery multiplier. The Verilog code in Fig. C.2 connects the CSA module to the MyHDL testbench. The testbench can then be written in Python using the MyHDL package as shown in Fig. C.3. Note that the imported libraries are not shown in this listing. Moreover this code only tests the CSA module with random input vectors.

C.2 Integration testing

Integration tests on high-level modules follow the same methodology as unit tests. For large modules, when simulations using Icarus Verilog are too slow, we use Verilator [ver] to convert our Verilog code to C++ and we write testbenches in C++.

```
/* csa.v */
module csa (

  // Inputs
  input ['WIDTH-1:0] data1_i,
  input ['WIDTH-1:0] data2_i,
  input ['WIDTH-1:0] data3_i,

  // Outputs
  output reg ['WIDTH-1:0] result_o,
  output reg ['WIDTH:0] carry_o

);

// Combinational logic
  always @(data1_i or data2_i or data3_i) begin
    result_o = data1_i ^ data2_i ^ data3_i;
    carry_o = {((data1_i & (data2_i | data3_i)) | (data2_i & data3_i)),1'b0};
  end

endmodule
```

Figure C.1: Verilog code for the CSA

```
/* test_csa.v */
module test_csa;

  //Inputs
  reg [`WIDTH-1:0] data1_i;
  reg [`WIDTH-1:0] data2_i;
  reg [`WIDTH-1:0] data3_i;

  //Outputs
  wire [`WIDTH-1:0] result_o;
  wire [`WIDTH:0] carry_o;

  // Pipes to and from MyHDL
  initial begin
      $from_myhdl(data1_i,data2_i,data3_i);
      $to_myhdl(result_o,carry_o);
  end

  // Instantiate the CSA
  csa #(
    .WIDTH(`WIDTH)
  ) csa1 (
    .data1_i(data1_i),
    .data2_i(data2_i),
    .data3_i(data3_i),
    .result_o(result_o),
    .carry_o(carry_o)
  );

endmodule
```

Figure C.2: Wrapper for testing the CSA with MyHDL

```
class test_CSA(unittest.TestCase):
  '''Tests of the CSA module'''

  def csa(self,data1_i,data2_i,data3_i,result_o,carry_o,width):
    '''Design under test'''

    # Compile module
    os.system("iverilog -o csa -DWIDTH=%s csa.v test_csa.v" % width)

    # Return co-simulation object
    return Cosimulation("vvp -M. -mmyhdl csa,data1_i=data1_i,data2_i=data2_i,\
    data3_i=data3_i,result_o=result_o,carry_o=carry_o)

  def test_ResultCarry(self):
    '''Tests that the module CSA gives the expected results'''

    # Testbench
    def stimulus(width):
      for i in range(test_csa_nb):
        # Set inputs
        data1_i.next = intbv(random.randrange(1,2**width-1,1))[width:]
        data2_i.next = intbv(random.randrange(1,2**width-1,1))[width:]
        data3_i.next = intbv(random.randrange(1,2**width-1,1))[width:]

        # Wait for 1 ns
        yield delay(1)

        # Compute the expected results
        r,c = c_add(data1_i, data2_i, data3_i,width)

        # Compare to the results of the Verilog module
        self.assertEqual(result_o,r)
        self.assertEqual(carry_o,c)

        print "Test %i: OK"%(i+1)

    # Define signals
    data1_i = Signal(intbv(0)[width:])
    data2_i = Signal(intbv(0)[width:])
    data3_i = Signal(intbv(0)[width:])
    result_o = Signal(intbv(0)[width:])
    carry_o = Signal(intbv(0)[width+1:])

    # Instantiate DUT
    dut = self.csa(data1_i,data2_i,data3_i,result_o,carry_o,width)

    # Instantiate testbench
    check = stimulus(width)

    # Create simulation
    sim = Simulation(dut,check)

    # Run simulation
    sim.run(quiet=1)
```

Figure C.3: MyHDL testbench for the CSA

Appendix D

Experimental Parameters

This appendix details the different parameters used for synthesis and place-and-route of our designs on FPGA and ASIC.

D.1 Xilinx tools options

D.1.1 Virtex-5

Our Virtex-5 FPGA designs are all implemented using Xilinx ISE 14.1. In chapters 3 and 5, the following synthesis (`xst`) options are used (parameters not specified below are set to their default values):

```
-iuc No
-keep_hierarchy No
-netlist_hierarchy As_Optimized
-rtlview Yes
-glob_opt AllClockNets
-read_cores Yes
-write_timing_constraints No
-cross_clock_analysis No
-hierarchy_separator /
-bus_delimiter <>
-case Maintain
-slice_utilization_ratio 100
-bram_utilization_ratio 100
-dsp_utilization_ratio 100
-rom_style Auto
-auto_bram_packing No
-mux_extract Yes
-resource_sharing Yes
-async_to_sync No
-use_dsp48 Auto
-iobuf Yes
-max_fanout 100000
-bufg 32
-register_duplication Yes
-register_balancing No
-slice_packing Yes
-iob Auto
-equivalent_register_removal Yes
```

```
-opt_mode Speed
-opt_level 1
-power No
-lc Off
-fsm_style LUT
-ram_extract Yes
-ram_style Auto
-rom_extract Yes
-reduce_control_sets Off
-verilog2001 Yes
-fsm_extract Yes -fsm_encoding Auto
-safe_implementation No
-shreg_extract Yes
-shift_extract Yes
-xor_collapse Yes
-mux_style Auto
-decoder_extract Yes
-priority_extract Yes
-slice_utilization_ratio_maxmargin 5
-optimize_primitives No
-use_clock_enable Auto
-use_sync_set Auto
-use_sync_reset Auto
```

We use the following map options for Virtex-5 FPGAs (the other options are set to their default values):

```
-w
-logic_opt off
-ol high
-register_duplication off
-global_opt off
-mt off
-cm area
-ir off
-pr off
-x
-lc off
-power off
```

and the following place-and-route (`par`) options:

```
-ol high
-x
```

In particular the `-x` option instructs the `map` and `par` tools to run in performance evaluation mode. Instead of using external timing constraints, the tool automatically generates timing constraints for each internal clock separately and adjusts the constraints during execution [Xil12b, p. 123].

D.1.2 Spartan-6

In chapter 4, for synthesis on Spartan-6 XC6SLX45T, we use XST 13.2 with the following parameters:

```
-opt_mode area
-opt_level 1
-use_dsp48 no
-resource_sharing no
-netlist_hierarchy rebuilt
-equivalent_register_removal no
-iobuf NO
```

For the Spartan-6 implementations of chapter 6, we use the default parameters provided by Xilinx Platform Studio 14.1.

D.2 Synopsys tools options

D.2.1 65 nm

For 65 nm, we use Synopsys Design Compiler version G-2012.06-SP2 with the following constraints (this example is for a 1 GHz clock):

```
set CLK_PORT [get_ports clock_i]
set CLK_PERIOD 1.00
set CLK_SKEW 0.1
set RST_PORT [get_ports reset_i]
set MAX_OUTPUT_LOAD [load_of tcbn65gplustc/DFD1/D]
set MAX_AREA 20000000
create_clock -period $CLK_PERIOD -name my_clock $CLK_PORT
set_dont_touch_network my_clock
set_dont_touch $RST_PORT
set_ideal_network $RST_PORT
set_driving_cell -library tcbn65gplustc -lib_cell DFD1 -pin Q\
[remove_from_collection [all_inputs] $CLK_PORT]
set_max_area $MAX_AREA
set_load  $MAX_OUTPUT_LOAD [all_outputs]
```

Setting an area constraint of 20000000 is equivalent to setting no constraint on the area available. We assume that the inputs and outputs of our designs are connected to D flip-flops. An example Design Compiler script is as follows:

```
analyze -format verilog {VERILOG MODULES HERE}
elaborate lucas -architecture verilog -param "PARAMETERS HERE"\
-library WORK -update
uplevel #0 source top-level.tcl
compile -exact_map
report_timing
report_area
write -hierarchy -format verilog -output lucas.v
exit
```

A corresponding PrimeTime-PX G-2012.06-SP2 script for power analysis is as follows:

```
set power_enable_analysis TRUE
set target_library "tcbn65gplus_200a/tcbn65gplustc.db"
```

```
set link_library "tcbn65gplus_200a/tcbn65gplustc.db *"
read_db $target_library
read_verilog lucas.v
current_design lucas
link
read_vcd dmp_lucas.vcd -strip_path lucas_tb/lucas -pipe_exec "./simv"
create_clock -name CLK -period 1.00 clock_i
estimate_clock_network_power tcbn65gplustc/BUFFD1
report_power -verbose -include_estimated_clock_network
exit
```

The VCD simulation file obtained with Synopys VCS G-2012.09 is piped to PrimeTime-PX to prevent the creation of a multi-gigabyte VCD file on disk. A virtual clock network is created to improve the power consumption estimations.

D.2.2 45 nm

For 45 nm, we use the following Design Compiler constraints (this example is for a 1.66 GHz clock) in the `top-level.tcl` script:

```
set CLK_PORT [get_ports clock_i]
set CLK_PERIOD 0.60
set CLK_SKEW 0.1
set RST_PORT [get_ports reset_i]
set MAX_OUTPUT_LOAD [load_of tcbn45gsbwptc/DFD1BWP/D]
set MAX_AREA 20000000
create_clock -period $CLK_PERIOD -name my_clock $CLK_PORT
set_dont_touch_network my_clock
set_dont_touch $RST_PORT
set_ideal_network $RST_PORT
set_driving_cell -library tcbn45gsbwptc -lib_cell DFD1BWP -pin Q\
[remove_from_collection [all_inputs] $CLK_PORT]
set_max_area $MAX_AREA
set_load  $MAX_OUTPUT_LOAD [all_outputs]
```

An example Design Compiler script is as follows:

```
analyze -format verilog {VERILOG MODULES HERE}
elaborate lucas -architecture verilog -param "PARAMETERS HERE"\
-library WORK -update
uplevel #0 source top-level.tcl
compile -exact_map
```

```
report_timing
report_area
write -hierarchy -format verilog -output lucas.v
exit
```

A corresponding PrimeTime-PX script for power analysis is as follows:

```
set power_enable_analysis TRUE
set target_library "tcbn45gsbwp_120a/tcbn45gsbwptc.db"
set link_library "tcbn45gsbwp_120a/tcbn45gsbwptc.db *"
read_db $target_library
read_verilog lucas.v
current_design lucas
link
read_vcd dmp_lucas.vcd -strip_path lucas_tb/lucas -pipe_exec "./simv"
create_clock -name CLK -period 0.60 clock_i
estimate_clock_network_power tcbn45gsbwptc/BUFFD0BWP
report_power -verbose -include_estimated_clock_network
exit
```

D.3 Lucas test input for maximum execution time

The 1024-bit input used for evaluating the maximum execution time of our Lucas prime tester in chapter 5 is (in base 10):

```
668647161145670418181901000971516076762123750811390196208253026225533690229139571815455241406124952270728781603175574855517910585526761579985264395395360040448636127782595837522244568144171382131502104839848463557748841668455543963929080898584291027115270995547960636450319749790819497207914905194043439437 83
```