

Number 701



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Vector microprocessors for cryptography

Jacques Jean-Alain Fournier

October 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<http://www.cl.cam.ac.uk/>

© 2007 Jacques Jean-Alain Fournier

This technical report is based on a dissertation submitted April 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

<http://www.cl.cam.ac.uk/techreports/>

ISSN 1476-2986

Abstract

Embedded security devices like ‘Trusted Platforms’ require both scalability (of power, performance and area) and flexibility (of software and countermeasures). This thesis illustrates how data parallel techniques can be used to implement scalable architectures for cryptography. Vector processing is used to provide high performance, power efficient and scalable processors. A programmable vector 4-stage pipelined co-processor, controlled by a scalar MIPS compatible processor, is described. The instruction set of the co-processor is defined for cryptographic algorithms like AES and Montgomery modular multiplication for RSA and ECC. The instructions are assessed using an instruction set simulator based on the ArchC tool. This instruction set simulator is used to see the impact of varying the vector register depth (p) and the number of vector processing units (r). Simulations indicate that for vector versions of AES, RSA and ECC the performance improves in $O(\log(r))$. A cycle-accurate synthesisable Verilog model of the system (VeMICry) is implemented in TSMC’s 90nm technology and used to show that the best area/power/performance trade-off is reached for $r = \frac{p}{4}$. Also, this highly scalable design allows area/power/performance trade-offs to be made for a panorama of applications ranging from smart-cards to servers. This thesis is, to my best knowledge, the first attempt to implement embedded cryptography using vector processing techniques.

Acknowledgements

This PhD has been a wonderful chapter of my life and I wouldn't have reached that point without the help and support of my family, friends and colleagues. I have to start with my family who has been so loving, caring and supportive: my parents Jacques & Rosemay, Claude-May, Joy, Suzy, Lindsay and my beloved Mée. There are the friends of course, who, even if I haven't been always there for them during those past 4 years, have always been willing to share with me both good and bad moments. It would take too long to mention all of them. . . I would also like to thank those who gave to me the opportunity to perform this research and work on such a challenging topic. I would like to start with *Simon Moore* who has so kindly welcomed me into his research group and helped me in doing this PhD in the best conditions. I am also grateful to my managers from Gemplus: *David Naccache* for his encouragement, *Philippe Proust* who has been so supportive, *Nathalie Feyt* who always had the right words and *Jean-François Dhem* for his technical advice but most precious of all, for his friendship. I also have a special thought for my former and current colleagues and friends at Gemplus, including *Michel Agoyan* for his precious technical inputs and pointers. Moreover, I want to thank all those I came across and worked with at the Computer Lab including *Ross Anderson*, *Markus Khun*, *Lise Gough* and, of course, my colleagues from the Computer Architecture group. I would also like to thank *Julie Powley*, from Trinity Hall, for her precious help despite my 'erratic' comings and goings. I hope that I didn't forget anyone, but in case I did, I just wanted to say that I didn't do it on purpose and that I am most thankful.

Nomenclature

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
ATM	Automated Teller Machines
CISC	Complex Instruction Set Computer
CPA	Correlation Power Analysis
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DES	Data Encryption Standard
DPA	Differential Power Analysis
DRM	Digital Rights Management
DSP	Digital Signal Processor
ECC	Elliptic Curve Cryptography
EEPROM	Electrically Erasable Programmable ROM
EMV	Europay Mastercard Visa
FIB	Focussed Ion Beam
FPOS	Floating Point Operations per Second
GLOPS	Giga Floating point Operations Per Second
GPC	General Purpose Co-processor
GPP	General Purpose Processor
HSM	Hardware Security Module
ILP	Instruction Level Parallel
ISA	Instruction Set Architecture

LFSR Linear Feedback Shift Registers
LSB Least Significant Byte
LSb Least Significant bit
MIMD Multiple Instruction Multiple Data
MSb Most Significant bit
NVM Non-Volatile Memory
OS Operating System
PK Public Key
PKC Public Key Cryptography
RAM Random Access Memory
RISC Reduced Instruction Set Computer
ROM Read Only Memory
RSA Rivest Shamir Adleman algorithm
SIM Subscriber Identity Module
SIMD Single Instruction Multiple Data
SISD Single Instruction Single Data
SK Secret Key
SPA Simple Power Analysis
SSL Secure Sockets Layer
TLS Transport Layer Security
TPM Trusted Platform Modules
VeMICry Vector MIPS for Cryptography
VLIW Very Long Instruction Word
VPU Vector Processing Unit

Contents

1	Introduction	15
1.1	Organization of this document	16
1.2	Refereed Publications	17
2	Security Devices: Functions & Architectures	19
2.1	Description of Security Devices	19
2.2	Requirements for Security Devices	21
2.2.1	Cryptographic protocols & algorithms	22
2.2.2	Certification of security devices	25
2.2.3	Attacks and Countermeasures	27
2.3	Architectures used in Embedded Systems	32
2.3.1	Classification of computer architectures	35
2.3.2	Vector architectures	37
2.3.3	Cryptography on parallel architectures	40
2.4	Conclusion on Security Devices	41
3	Vectorizing Cryptographic Algorithms	43
3.1	Requirements for Secure Chips	43
3.1.1	Requirements for cryptographic applications	44
3.1.2	Security Requirements	46
3.1.3	Introduction to the detailed case studies	48
3.2	Architecture Definition for Parallel Cryptographic Calculations	48
3.2.1	A case for general purpose programmable architectures	48
3.2.2	A case for vector architectures	50
3.2.3	Having a MIPS-based approach	51
3.3	Case Study: AES encryption algorithm	52
3.3.1	The Advanced Encryption Standard	52
3.3.2	Performance study	53
3.3.3	Improvement study	55
3.4	Case Study: Elliptic Curve Cryptography	59
3.4.1	A brief overview of ECC theory	60
3.4.2	Scalar multiplication over \mathbb{F}_{2^m}	61
3.4.3	Performance study	66
3.4.4	Improvement Study	68
3.5	Summary	69

CONTENTS

4	VeMICry Architecture & Functional Simulation	71
4.1	The VeMICry Architecture	71
4.1.1	Architecture specification	71
4.1.2	Vector instructions set architecture	73
4.2	The VeMICry Functional Simulator	80
4.2.1	Simulation tools	80
4.2.2	Building the functional VeMICry	81
4.2.3	Simulating the AES	82
4.2.4	Modular multiplication in binary fields	83
4.2.5	Modular multiplication for RSA	84
4.3	Quantitative Analysis	86
4.3.1	Tools set-up & methodology	86
4.3.2	Varying key size	87
4.3.3	Changing depth of vector registers	87
4.3.4	Varying the number of lanes	89
4.4	Summary and Conclusions	90
5	Cycle Accurate Verilog Model of VeMICry	93
5.1	Verilog Implementation of Scalar Processor	93
5.1.1	Scalar Instruction Set	95
5.1.2	Implementation details	99
5.1.3	Performance and area figures	103
5.2	Verilog Model of the Vector Co-processor	104
5.2.1	Vector instruction execution	107
5.2.2	Implemented vector instructions	108
5.2.3	Vector pipeline hazards	109
5.3	VeMICry system	112
5.3.1	Rules for executing scalar and vector code	113
5.4	Summary	114
6	Analysis of Verilog VeMICry	115
6.1	Functional <i>versus</i> Cycle-accurate Simulation	117
6.2	Comparing Expected and Measured Results	117
6.3	Performance <i>versus</i> Vector Register Depth	119
6.4	Performance <i>versus</i> Number of Lanes	119
6.5	Area \times Performance <i>versus</i> Number of Lanes	119
6.6	Power Characteristics of VeMICry	120
6.7	Security issues	122
7	Conclusion	129
A	Vector Instructions	133
B	Vector Codes	137
B.1	AES vector code	137
B.2	ECC vector code	138
B.3	RSA vector code	139

C	VeMICry Test Code	141
D	VeMICry Verilog	143
	D.1 Scalar Processor Verilog	143
	D.2 Vector Processor Verilog	154
E	Measurements on Cycle-accurate VeMICry	163
	References	165

CONTENTS

List of Figures

2.1	Architecture of a smart-card	19
2.2	Architecture of a Hardware Security Module	21
2.3	Square and Multiply Algorithm for RSA	24
2.4	CRT Algorithm for RSA signature	25
2.5	DEMA during an XOR on the asynchronous XAP	28
2.6	Power Glitch Attack on the asynchronous XAP	30
2.7	Architecture Design Space based on <i>Flynn's taxonomy</i>	36
2.8	Modified Architecture Design Space	37
2.9	The Vector Processing Paradigm	38
3.1	(<i>Flynn's</i>) Architecture Design Space	50
3.2	AES-128 structure	54
3.3	Double and Add algorithm	62
3.4	Point addition in Jacobian coordinates	64
3.5	Point doubling in Jacobian coordinates	64
3.6	Montgomery Modular Multiplication on a 32-bit architecture	65
3.7	Modular Multiplication Implementation	67
4.1	Vector Register File	72
4.2	Timing relationship between scalar & vector executions	74
4.3	Execution of VADDU instruction	78
4.4	Montgomery Modular Multiplication on a 32-bit architecture	83
4.5	Modular multiplication for RSA with the CIOS method	85
4.6	Modular Multiplication for RSA with FIOS method	85
4.7	Number of cycles <i>versus</i> key size (Vector Depth= 32)	87
4.8	Number of cycles <i>versus</i> key size for different vector depths p	88
4.9	Performance <i>versus</i> Vector Depth	89
4.10	Rate of change of performance <i>versus</i> $\log_2(\text{VectorDepth})$	90
4.11	Performance <i>versus</i> number of lanes for different vector depths	90
4.12	Rate of change in performance <i>versus</i> $\log_2 r$	91
5.1	Architecture of the scalar MIPS	94
5.2	Scalar Instructions' Structures	96
5.3	Execution of a taken branch instruction	101
5.4	RAW Hazards & Register Bypasses	103
5.5	LOAD stall on scalar processor	104
5.6	Vector Register File	105

LIST OF FIGURES

5.7	Architecture of the Vector co-processor	106
5.8	Architecture of a Vector Lane	107
5.9	Timing relationship between scalar & vector executions	107
5.10	Pipeline stall for RAW data hazard when $v_r = v_p$	111
5.11	RAW data hazard when $v_r < v_p$	112
5.12	Final Architecture with scalar & vector parts	113
6.1	Area of vector co-processor <i>versus</i> number of lanes	116
6.2	Max frequencies <i>versus</i> number of lanes	116
6.3	FIOS Method for Montgomery's Modular Multiplication	117
6.4	Comparing functional and cycle accurate simulators	117
6.5	Ratio of theoretical over measured variations in performance	118
6.6	Performance <i>versus</i> depth on 2048-bit data	119
6.7	Performance <i>versus</i> number of lanes on 1024-bit data ($p = 32$)	120
6.8	Area \times Performance <i>versus</i> number of lanes	121
6.9	Mean power <i>versus</i> depth p for one lane	122
6.10	Power <i>versus</i> number of lanes for $p = 16$	123
6.11	Performance \times Power <i>versus</i> number of lanes ($p = 32$)	124
6.12	Difference Power for varying Hamming weights of B on 512 bits	126
6.13	Power Traces for different Hamming Weight data on 256 bits	127

Chapter 1

Introduction

At the dawn of this new century, mankind has fully stepped into the digital age. Everything around us has become information, a string of zeros and ones. This dematerialization of things has raised new security issues. Our money and bank account details are no longer stored in a heavily guarded vault in a bank but on (may-be-heavily-guarded-as-well) computer servers and in the small chips of our bank cards. Our personal things like mail, personal contacts, family pictures, video recordings are all in digital format on our laptops, phone, PDA or portable game, at the mercy of anyone having access to them. Payments and communications are increasingly wireless making them easy for discrete, un-noticed spying. Identification documents are all going digital as well raising the awkward question: *How can we distinguish between an original string of zeros and ones and a copied or falsified identical string of zeros and ones?*

Fortunately these risks have been well studied and understood and adequate solutions have been proposed. In the industry, special electronic devices like smart-cards (for portable systems) and Hardware Security Modules (for computers, servers, ATMs...) are the secure platforms used for their respective domains of application. Other industry driven initiatives like the Trusted Computing Group work to define the security requirements pertaining to the deployment of new applications. The number of security related conferences has exploded during the past decade (CHES, CARDIS, ACNS, ARITH, ASAP, DRMTICS, RSA, FSE, ESMART...) with strong participation from academic research groups. Security issues in digital applications is now a hot topic both for academia and industry.

Until now there have been what could be qualified as being two worlds of secure computer hardware, each defined by their respective constraints. The first one is that of “conventional” computing like “fixed” computers, HSMs (Hardware Security Modules) or servers where the challenge is to achieve high data throughput for security-related computations like encryption, signing or hashing. Such devices integrate enough hardware resources to run cryptographic algorithms rapidly and physical tamper-proof shields to protect any sensitive unit. The second “world” is that of embedded systems where the hardware constraints are different. Such devices are limited in size, power and packaging capabilities. These have a direct impact on performance and of course on the level of security that could be reached. This second “world” has been dominated by smart-cards.

1. INTRODUCTION

But now, we are witnessing the emergence of a new breed of computer hardware which is the convergence of the two worlds of “conventional” and “embedded” computing. This means that portable devices like PDAs, laptops or mobile phones are expected to be equipped with “Trusted Platforms” that could offer the same high level of security as smart-cards while targeting the high computing powers of devices like HSMs. In this view, the aim of this research work is to propose an architecture for cryptography satisfying the requirements of this new generation of computers:

- **Performance**, in this work, relates to the ability of a given circuit to achieve a high data throughput for commonly used cryptographic algorithms.
- **Security** has a direct impact on performance since software countermeasures increase the time taken to execute cryptographic algorithms and hardware countermeasures increase the critical path of the circuit (i.e. lower the maximum frequency at which the circuit could be clocked) or increase the power of the chip. So this research favors the concept of adaptive security in the sense that hardware resources are made available to optimize the implementation of countermeasures in software depending on the level of security required by the target application.
- **Scalability** is the key point here. For different applications and devices, the area and power constraints may not be the same. The architecture can be resized to allow area, power and performance tradeoffs.

In the research described in this thesis, these constraints are studied and an architecture based on the concept of a programmable vector microprocessor is proposed. The adaptation of some commonly used cryptographic algorithms to such an architecture is illustrated. As a proof-of-concept, a cycle-accurate synthesisable Verilog model of the vector microprocessor (called VeMICry) is implemented. The described features are limited to modular multiplications in binary fields (for applications like Elliptic Curve Cryptography), which is enough to study all the pipelining and parallelization issues of such a design. The measurements made are extrapolated to analyze the behavior of such a design for modular multiplications on large numbers (e.g. 1024-2048 bits RSA).

1.1 Organization of this document

The organization of this report reflects the chronological order in which this research work has been conducted.

Chapter 2 provides background information. First, an overview of security-related applications is given and the corresponding requirements are defined, leading to the study of required cryptographic algorithms, the potential attacks on them and some of the countermeasures that are proposed. A brief survey of hardware architectures used in today’s security devices is then given before going back to a general exploration of the hardware architecture design space resulting in a focus on vector processors. This chapter ends with the study of some of the attempts that have been made in the past to parallelize cryptographic calculations.

Chapter 3 defines the features and requirements upon which this research focusses. A case for choosing a vector processing approach is built. A focus on algorithms like the AES and modular multiplication in binary and prime fields is made to show how such algorithms can be expressed as operations on vectors of data. The latter analysis helps to define a first set of vector instructions that would have to be implemented on VeMICry.

Chapter 4 provides a theoretical definition of the proposed vector microprocessor. This chapter also defines the way in which the design can be made scalable and what the parameters are. The pipelined architecture of the vector co-processor is studied. In order to validate the functional relevance of the identified vector instructions, a simulator was built using an architecture simulation tool called ArchC. The simulator was used to test the vector codes written for AES and modular multiplication. This chapter ends with an analysis of how performance (in terms of instructions issued) is affected by playing with the different parameters of the vector architecture, which provides a first order approximation of the kind of behaviour to expect from such an architecture.

Chapter 5 describes the implementation of a cycle accurate Verilog model of VeMICry. The detailed implementations of the scalar general purpose processor and that of the vector co-processor are given. Both architectures were individually synthesized into TSMC's 90nm technology. In both cases, pipelining issues and the handling of hazards were taken into account with the aim of ultimately reaching an instruction issue rate of one instruction per clock-cycle. This chapter also describes what scalar and vector instructions are implemented on this model, how the two kinds of instructions interact with each other and what the restrictions to these interactions are.

Chapter 6 describes the quantitative analysis carried on the Verilog model of VeMICry. The impact of changing the design parameters of the vector co-processor were analysed in terms of performance, area and power. The study was also extended to scenarios where the size of the data being worked on was varied. Finally a brief look was taken at the kind of side-channel power information leakage that can be expected from such a circuit.

Chapter 7 is the conclusion in which the main achievements of this research work are summarised.

The appendices provide details of the vector instructions and the vector codes for the AES algorithm and the modular multiplication routines for ECC and RSA. The Verilog source code of the scalar and vector units of the cycle accurate model of VeMICry are also provided.

1.2 Refereed Publications

Several aspects of the research work in this thesis have been presented at refereed conferences. The concept of having a vector approach to cryptography was first described in [Fournier & Moore (2006b)]. In this paper, I talked about why and how cryptographic computations like AES and modular multiplication can be expressed in a vector form (c.f. Chapter 3). I also gave a high level description of the hardware architecture of VeMICry.

1. INTRODUCTION

In [Oikonomakos *et al.* (2006)] I explained how a vector based co-processor for Elliptic Curve Cryptography can be used for secure key exchanges in a secure display application. In [Fournier & Moore (2006a)] I detailed the architecture of VeMICry used for Public Key Cryptography and published simulation results obtained from the instruction set simulator built from ArchC (c.f. Chapter 4).

Prior to the above publications, I have studied the security of asynchronous circuits for smart-card applications. During this research work (which is not presented in this thesis), I have put into practice attacks like Correlation Power Analysis (CPA) and fault injections. In this context, I have co-authored several refereed papers as in [Fournier *et al.* (2003); Moore *et al.* (2003)] and have been invited to publish an article in a French technical journal [Fournier & Moore (2004)]. More recently, I have also been looking at new attack techniques based on side-channel information leakage resulting from the use of caches in smart-cards' processors [Fournier & Tunstall (2006)].

Chapter 2

Security Devices: Functions & Architectures

2.1 Description of Security Devices

In the digital world, the devices managing and implementing security can be regrouped into two categories which are rather characteristic of two (*up-to-now*) distinct families of the computer world:

- In the *embedded* (portable) world of mobile phones and PDA-like devices, there are smart-cards into which we will also include devices like RFID tags and USB tokens.
- In the world of *classical* computing (PCs, servers, network computers...), there are dedicated security modules called HSMs (Hardware Security Modules).

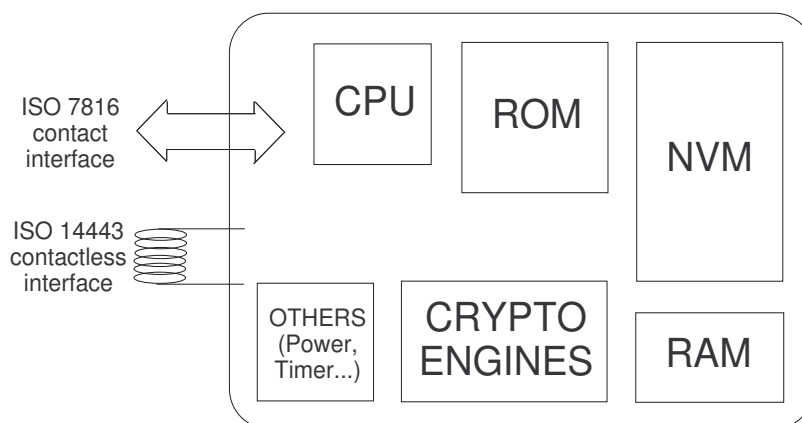


Figure 2.1: Architecture of a smart-card

Smart-cards have become a widely used commodity. In countries like France, such devices have been used for decades in pay phone applications (simple memory cards with a secured access to data stored in these memories) and in bank cards (cards with a microprocessor, volatile memory, non-volatile memory and cryptographic capabilities).

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

Nowadays, standards like the Europay Mastercard Visa standard [EMV (2004)] are being deployed throughout the world under the common name of Chip & PIN which are microprocessor based smart-cards. The chip of such smart-cards would usually have the features illustrated in Figure 2.1. The type of smart-cards illustrated in Figure 2.1 has also been deployed in electronic identity cards. For example, Belgium has already started the deployment of identity cards for its citizens [BelgianID (2006)]. Similar identity cards holding information like picture, personal details and biometric data are to be deployed in the UK from 2009. The contactless version of smart-card chips are being fitted into passports for securely holding similar information. However smart-cards are more widely deployed in the SIM (Subscriber Identity Module) used in mobile phones: in the GSM or 3G mobile communication schemes, the SIM card is used to identify the end-user to the network, to hold some personal data like phone-book, short-messages etc. and hence is used by the network operator for billing purposes. Pilot initiatives have recently been launched to also use the coupled mobile phone + SIM card for contactless payment applications or for accessing mobile TV networks similar to the way they are used in pay TV applications. According to Eurosmart (an industry driven consortium of major players of the smart-card industry), the volume of smart-cards being sold has been steadily growing by around 20-24% every year. For the telecommunications industry alone (mainly under the form of SIM cards) the annual volume of smart-cards has steadily been growing from 430 million in 2002 to 1.39 billion in 2005. All this illustrates that with smart-cards, what we are looking at here is a huge deployment of secure tokens used in most everyday life situations. Recently, smart-card manufacturers have been proposing a new type of smart-cards that integrates high density memories (like NAND Flash memories) for secure data storage along with interfaces that offer much higher communication speeds (e.g. the USB interface is currently being standardized by the ETSI as a smart-card standard). Such smart-cards are often called MegaSIM, HD-SIM or MultiMedia SIM.

The other main family of security devices is that of HSMs (Hardware Security Modules or also called *Host Security Modules*). HSMs offer secure storage and cryptographic capabilities for PCs, for securing communications and sensitive data storage in banks or for the personalization of other security devices like smart-cards. An example for the architecture of an HSM is given in Figure 2.2. HSMs like the HSM8000 (Thales) or the IBM4690 are used in ATMs (Automated Teller Machines) as well as in POS (Point of Sale) Terminals used in shops for paying with a bank card. nCipher, another HSM manufacturer, has for its part recently launched the miniHSM which is the first HSM designed for embedded applications for devices like POS but also for TPMs (Trusted Platform Modules) for laptops or other portable devices like mobile phones or PDAs. The TPM itself is the concept of a secure device defined by a consortium of major industry players called the Trusted Computing Group (TCG - <http://www.trustedcomputinggroup.org/home>). The applications targeted by such devices are Digital Rights Management (DRM), protection of on-line games, identity protection, security against viruses and spywares, biometric authentication and management and verification of digital certificates and signatures. More interestingly, the definition of the TPM has also been extended to the embedded world for specific mobile applications like accessing numerous networks (wireless, ADSL...), performing “mobile” financial transactions, protecting against malicious embedded software or transferring confidential data among devices [TPG (2006)].

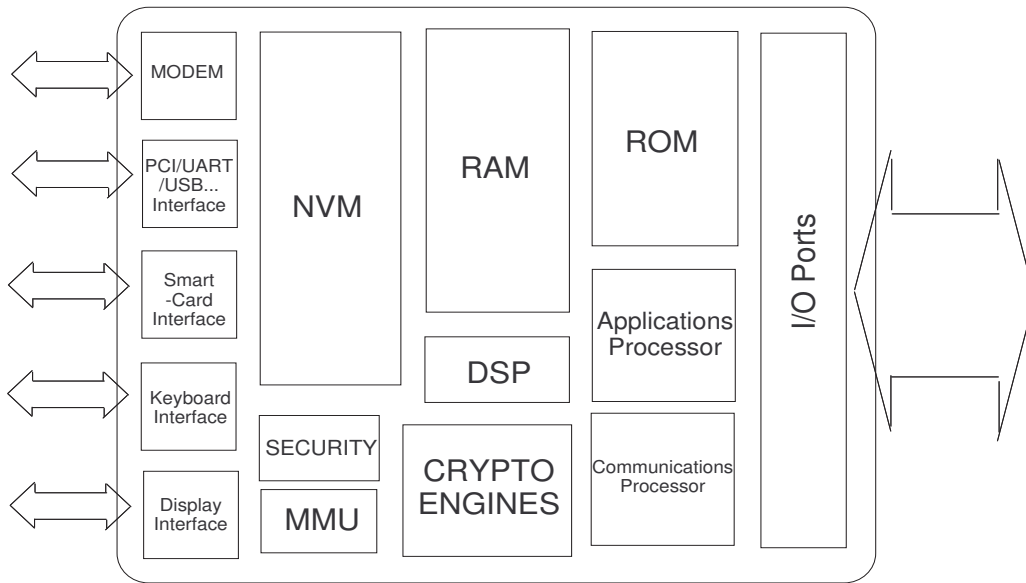


Figure 2.2: Architecture of a Hardware Security Module

2.2 Requirements for Security Devices

The world of smart-cards (with devices like `MultiMedia SIM`) and that of HSMs (with devices like `miniHSM`) are converging to a new family of security devices that are small enough to be fitted into portable devices and with enough features to target as many security applications as possible. Such security applications are based on well identified use-cases, often involving standardized secure communication and data management schemes. The latter themselves implement a plethora of cryptographic algorithms meaning that such security devices handle and store sensitive data. Hence such devices have to offer protection against any attempt to retrieve or corrupt the sensitive data. In this section, we look at the security requirements for such devices and how the conformance to these requirements is measured.

The most talked-about use-case, given the power of media conglomerates backing it, is that of Digital Rights Management (DRM) [OMA (2005)]. The aim of this scheme is to protect the intellectual property linked to the use of software and the portability of digital media. In this scenario, the secure device guarantees the identity of the user and binds the media contents to that user. The secure device stores the authentication keys of the user and provides a tamper-resistant environment to *sign* the purchase of media content (and its associated rights). The Right Objects (which contain sensitive information like the encryption keys of the content, the number of times the content can be accessed etc.) can be stored in the secure device whose role is then to check these rights every time the data is accessed and to decrypt it. For devices like the `MultiMedia SIM`, the SIM card would both store the data and securely manage its rights, and all this into one same

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

portable device such that the user can easily transfer his media from one mobile phone to another. From the latter description, we can see that another use of these secure devices is to securely store user data like professional e-mails and even personal data to ensure the privacy of the user's content. Another emerging market is that of mobile ticketing or mobile payment: in this example the mobile phone is used, via a contactless interface, to perform electronic payments for transport applications or in shops for small purchases. The mobile phone, in this case, is fitted with a secure device, which can be an HSM-like chip or a smart-card, which contains the banking information and performs the banking transaction. The latter operation is performed by the secure device itself thanks to the fact that the secure device is also a platform whose integrity is guaranteed. Hence the integrity and security of the applications running on the devices can also be guaranteed. For open platforms like *JavaCard*, software can be securely downloaded onto such secure devices and installed to offer new services to users.

2.2.1 Cryptographic protocols & algorithms

In most use-cases, the core of the problem is for the devices to securely communicate with the external world by guaranteeing the authenticity of the end-user and preserving the confidentiality of the data being transmitted. Such secure communications are based on protocols of which some commonly known ones are the following:

- SSL (Secure Sockets Layer), more recently transformed into TLS (Transport Layer Security) defines a set of cryptographic protocols for secure on-line transactions [[OpenSSL \(2000\)](#)]. The cryptographic protocols cover all the possible applications of cryptography going from digital signature schemes (RSA, DH, DSA and involving Hash functions like SHA and MD5), block ciphers (DES, AES, IDEA, RC2, RC4. . .) and stream ciphers.
- IPSec (IP Security) provides a set of protocols to secure IP packets. IPSec defines the key establishment between two entities and the way the packets are encrypted, involving the same lot of secret and public key cryptographic algorithms as those described in SSL.

The main tool used by security devices to offer such services is cryptography. Cryptography can be described as the art of transforming input data, called a *message*, into an output, called a *cipher*, sometimes using a secret *key* such that knowing the *cipher* no information can be inferred about the *message*. The aim here is neither to detail the history of cryptography (like [[Singh \(2000\)](#)]) nor to go into the mathematics behind cryptography (see [[Menezes et al. \(1997\)](#); [Schneier \(1995\)](#)]) but to give an overview of the principal cryptographic techniques. There are four major types of cryptographic functions. First there are Hash functions which transform data of arbitrary length into a *hash* of fixed length: the input of arbitrary length is decomposed into elementary data blocks on which arithmetic, Boolean or permutation operations are applied in order to obtain an output whose length, which is fixed, is smaller than the input message. Some examples of Hash functions are given in [[NIST \(2002\)](#)] in which the NIST (National Institute of Standards and Technology) has defined FIPS (Federal Information Processing Standard) approved

2.2 Requirements for Security Devices

Hash functions, namely the SHA algorithms. Other examples are MD4, MD5 and MAC functions [Menezes *et al.* (1997)]. The major properties of Hash function are uniqueness (that is two different input data must not produce the same output, if such an incident happens then a *collision* is said to have been produced) and one-way-ness (that is given a *hash* one must not be able to infer what the input data was).

A second category of cryptographic functions is that of stream ciphers. As its name suggests, a stream cipher encrypts the individual bits of the message (which can be of any length) into a stream of bits of the cipher which is of the same length as the message in such a way that the transformation applied to each bit varies at every cycle. Details about stream ciphers can be found in [Menezes *et al.* (1997)], some examples of stream ciphers being those based on LFSRs (Linear Feedback Shift Registers). Block ciphers, also called *Secret Key (SK)* algorithms constitute a third category of cryptographic algorithms. A Secret Key algorithm transforms an input message of fixed length into a cipher text of the same length based on a secret key (encryption). For a “good” block cipher, even if its algorithm is known, an attacker must not be able to decrypt the cipher text unless she has the secret key. Numerous block ciphers have been proposed in the past and are being used today, some being “secret” algorithms while others like RC6, IDEA or Skipjack are public. But the two which are more widely deployed and more commonly used are the FIPS-approved DES (Data Encryption Standard) or Triple-DES as defined in [NIST (1993)] and the FIPS-approved AES (Advanced Encryption Standard) [NIST (2001b)]. The DES algorithm is based on a secret key of 56 bits onto which straight-forward attacks like exhaustive search is made easy today given the power of available computers. For this reason, DES is meant to progressively disappear in favour of the AES which can use keys of 128, 196 or 256 bits [NIST (2001b)].

Since Block ciphers are among the fastest encryption algorithms, they are the most common tool used to implement a secure communication channel. For example, if Alice and Bob want to exchange data securely using a Secret Key scheme, they both must share the same secret key. One can see that there is the issue of ‘distributing’ the key between Alice and Bob, in other words how they manage to securely decide on which key to use while having Charlie in between eavesdropping their communication channel. A solution to this is offered by the forth class of cryptographic algorithms called *Public Key (PK)* Algorithms (or *asymmetric* algorithms). For such algorithms, two keys are used: a *public* one which is used for encryption and a *private* one which is used for decryption. The mathematical algorithms used in PK Algorithms are such that even if an attacker (for example Charlie) has the public key, it is computationally impractical for him to derive the private key. For Alice and Bob’s problem, each one will generate her/his own key pair and they will share their respective public keys while secretly keeping their private keys. Hence Alice will encrypt the data with Bob’s public key and send the encrypted data to Bob who will be the only one to be able to decrypt it using his private key. In such schemes, the decryption process is also known as *signature generation* because if Alice wants to know if Bob is really on the other end of the communication channel, she would send some ‘random’ message to Bob and ask Bob to encrypt it with his private key (just like physically “signing” a document). Alice would then *verify* the encrypted document using Bob’s public key and be certain that it was truly Bob on the other end.

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

Usually to strengthen the security of such schemes a third party, called a *Certification Authority* is involved to guarantee to Alice that the public key she has been using to verify Bob's encrypted message was truly Bob's. The drawback with PK cryptography is that they take much more time than SK algorithms. This is why, for example, PK algorithms are used to help Alice and Bob exchange a common secret key which they use in an SK algorithm to encrypt the rest of their communication. The first example of such asymmetric schemes was proposed by Diffie and Hellman in [Diffie & Hellman (1976)].

The most commonly used Public Key algorithm is RSA [Rivest *et al.* (1978)]. RSA is based on modular exponentiation algorithms on large integers. In such a scheme the public key (also known as the *public exponent* denoted e is used for encryption/verification following equation 2.1 and the private key (or *private exponent*), denoted d is used for decryption/signing following equation 2.2.

$$c = m^e \bmod n \tag{2.1}$$

$$m = c^d \bmod n \tag{2.2}$$

where n is a large modulus (usually of 1024 bits) chosen such that it is the product of two large primes p and q . Note that n is part of the public key as well. A straight-forward way to implement such a modular exponentiation is to use the Square-and-Multiply algorithm shown in Figure 2.3.

Input	: m, n and $d = \{d_{l-1}d_{l-2} \dots d_2d_1d_0\}_2$
Output	: $c = m^d \bmod n$

1. $c \leftarrow 1$
2. **for** $j = l - 1$ **downto** 0 **do**
3. $c \leftarrow c^2 \bmod n$
4. **if** $d_j = 1$ **then** $c \leftarrow c \times m \bmod n$
5. **endfor**
6. **return** c

Figure 2.3: Square and Multiply Algorithm for RSA

A faster method is to use arithmetic based on the *Chinese Remainder Theorem* (CRT) which states that if p and q are relatively prime to each other, for $a < p$ and $b < q$, there is a unique $x < p.q$ solution to the equations $x = a \bmod p$ and $x = b \bmod q$. Hence we have the algorithm in Figure 2.4 for implementing modular multiplication using the CRT algorithm [Menezes *et al.* (1997)]. The main advantage with this method is that the modular exponentiations are done $(\bmod p)$ and $(\bmod q)$ where p and q are approximately half the size of n .

During the past decade, Elliptic Curve Cryptography (ECC) [Blake *et al.* (1999)] has been emerging as an attractive alternative to RSA, attractive because ECC work on

Input : $m, p, q, d_p = d \bmod (p-1), d_q = d \bmod (q-1)$: and $I_q = q^{-1} \bmod p$
Output : $S_n = m^d \bmod n$
<ol style="list-style-type: none"> 1. $s_p \leftarrow m^{d_p} \bmod p$ 2. $s_q \leftarrow m^{d_q} \bmod q$ 3. $S_n \leftarrow s_q + (((s_p - s_q) \cdot I_q) \bmod p) \cdot q$ 4. return S_n

Figure 2.4: CRT Algorithm for RSA signature

smaller data values than RSA for an equivalent level of security (for example the security provided by a 163-bit ECC is equivalent to that provided by a 1024-bit RSA). ECC is, therefore, faster and consumes less memory. In ECC calculations are done on an elliptic curve. The computations are done over a field \mathbb{K} which can be a prime (\mathbb{F}_p where p is a prime) or a binary (\mathbb{F}_{2^m} where m is a positive integer) field. The elliptic curve for a prime field is given by equation 2.3.

$$y^2 = x^3 + ax + b \quad \text{with } \{a, b\} \in \mathbb{F}_p \quad \text{for } p > 3 \quad (2.3)$$

In a PK cryptosystem based on ECC, encryption/verification consists in performing the multiplication between a given point G on the elliptic curve and a scalar value k to have another point on the elliptic curve called Q .

$$Q = k.G \quad (2.4)$$

The robustness of this scheme relies on the fact if we have Q and G , finding k is considered to be a very difficult problem (called the *Elliptic Curve Discrete Logarithm Problem*), just like in the case of RSA (equation 2.2), of we know c and m it is a difficult to find d .

2.2.2 Certification of security devices

In order to verify that security devices are compliant with the stringent functional and security features required by some applications, certification schemes have been devised. Such schemes provide guidelines on what such devices should incorporate and the levels of security that are required. The certification schemes not only check that a given security device can offer the computation functionalities to perform cryptographic computations but also that the computations are ‘securely’ executed and that the secret data manipulated and stored within the security device are protected against attempts of finding or corrupting them. We first have a look at the two main certification schemes that exist today and then detail the kind of attacks which the security device should offer protection against.

The Common Criteria (CC) scheme is an industry driven scheme that provides guidelines and recommendations to evaluation centers on how to certify a security device [CC

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

(1999)]. The device's level of security is measured based on the definition of a Protection Profile (PP) which defines the security functionalities and the countermeasures to be tested on the Target Of Evaluation (TOE). Seven security levels are defined:

- *EAL 1*: The evaluator performs functional tests.
- *EAL 2*: The evaluator does structural tests requiring the developer's cooperation.
- *EAL 3*: The evaluator methodically checks and tests the TOE, searching for potential vulnerabilities.
- *EAL 4*: The evaluator verifies that the TOE has been methodically designed, tested and reviewed and that it provides a high level of security.
- *EAL 5*: The evaluator ensures that the TOE has been semi-formally tested. The TOE must offer protection against attackers with "moderate" capabilities.
- *EAL 6*: The evaluator requires that the design has been semi-formally verified and tested. The TOE must offer protection against attackers with "high" capabilities.
- *EAL 7*: The evaluator requires that the design has been formally verified and tested. The TOE must be able to still offer highly secure services even in extremely high risk situations.

In the world of smart-cards, the most commonly used Protection Profile is the PP9806 [CC (1998)], EAL4 or EAL5 being the security levels that are usually required.

The second certification model is specified by the Federal Information Processing Standards (FIPS), which focusses more on the hardware itself. The FIPS 140-2 [NIST (2001a)] defines the cryptographic algorithms, the interfaces, the life cycle, the testing and validation procedures, the physical security and the key management for cryptographic modules. Four levels of security are specified:

- *Level 1* only requires that the computation functionalities are met, no specific countermeasure is required, for e.g. a PC.
- *Level 2* requires tamper-evident casing, authentication to the cryptographic module and an OS at least equivalent to a Common Criteria EAL 2 security level.
- *Level 3* imposes tamper resistance (i.e. detecting and responding to physical access), requires that the authentication to the cryptographic module is identity-based, regulates the personalization of the security device and demands that the OS be at least equivalent to a Common Criteria EAL 3 security level.
- *Level 4* adds the requirement that the security device can be used in a non-trusted environment. This means that upon any attempt to access the security device, the secret assets shall be deleted. The security device has to detect abnormal physical working conditions (voltage, temperature). The OS must be at least equivalent to a Common Criteria EAL 4 security level.

In addition, the FIPS 140-2 also defines how random numbers are generated and how cryptographic keys are generated and personalized.

2.2.3 Attacks and Countermeasures

Both the CC and the FIPS schemes refer to attacks against which the security device must offer protection. A good survey of possible attacks on security devices is given in [Anderson & Kuhn (1996, 1997)]. The first class of attacks on cryptographic algorithms is one called *cryptanalysis* [Sinkov (1966)]. In cryptanalysis, an attacker tries to find weaknesses in the construction of the algorithm under study and tries to infer information either about the secret key used or to establish a predictable correlation between the input message and the cipher text. A recent example of cryptanalysis that has been shaking the world of cryptography recently has been the set of collision attacks that have been published against MD5 and SHA1 (both being Hash algorithms used to generate SSL web server certificates) since 2004. Since such attacks are directly linked to the mathematical definitions of the cryptographic algorithms, they are of a lesser concern to the research work described in this thesis. My primary concern is the devices which run the cryptographic algorithms, how an attacker might exploit weaknesses in the algorithms' implementation or in the way the secret data is stored and handled by the devices.

Another family of attacks is that of *invasive attacks* whereby the attacker has physical access to the security device and tries to directly observe the signals from within the chip (through techniques like micro-probing) or read the bits from within the memories themselves or to use tools like a FIB (Focussed Ion Beam) to short-circuit some of the security sensors of the circuit or re-connect some other parts which had intentionally been disconnected for security purposes [Walker & Alibhai-Sanghrajka (2004)]. Such techniques could be destructive, but the attacker would have gained knowledge either on the way the security device has been implemented (both from a hardware and a software point of view) or on the secret keys it stores, which would be particularly interesting if this same key is shared with other security devices. Such attacks are not a primary concern to this thesis because the latter focusses on architectural techniques for cryptography which is independent from the transistor-level considerations or packaging aspects that are directly linked to invasive attacks.

A less destructive class of attacks is based on *side channel information leakage* where there is, in most cases, no need to tamper with the security device. They are usually called *non-invasive attacks*. Information about the internal processes of the chip and the data it is manipulating can be derived by observing external physical characteristics of the chip (like the power consumed or the Electromagnetic waves emitted or the time taken by a given process). *Timing Attacks* are a class of side-channel attack as described in [Dhem *et al.* (1998); Kocher (1996)]. An easy example of such attacks targeted what we call now "naïve" implementations of RSA: a straight-forward and fast way of implementing the modular exponentiation, given by equation 2.2, on devices where all data have a binary representation is to perform recursive square and multiplies by scanning the bits of the d as given in Figure 2.3. From the latter algorithm, we can see that for each bit of d , the time taken when d_j is equal to 1 is longer than the time taken when d_j is equal to 0. Hence by measuring only one power curve during such an exponentiation, one can retrieve the entire private exponent d . In this particular case, the attacker can also look at the timing taken by each `for` loop but may also look at the power profile in cases where the square operation of line 3 generates a different power profile from the multiply operation

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

of line 4. This is an example of a *Simple Power Analysis*. *Power Analysis* attacks were first published by Paul Kocher in 1999 [Kocher *et al.* (1999)] and since then side channel attacks have been a major concern to the security world, in particular for the smart-card industry. *Differential Power Analysis* mainly targets Secret Key algorithms: such attacks are based on the fact that for each bit of the secret key, the power consumed by a 0 is different from the power consumed by a 1. Later researchers showed that Electromagnetic radiation from a chip could also be used as a source of side-channel for SPA or DPA to give what they call SEMA (Simple Electromagnetic Analysis) and DEMA (Differential Electromagnetic Analysis) [Gandolfi *et al.* (2001); Quisquater & Samyde (2001)]. More recently, *Correlation Power Analysis* has been proposed to quantitatively correlate the power or Electromagnetic waves measured with the Hamming Weight (in other words the number of ones) in a data being manipulated by a security device [Brier *et al.* (2003, 2004)]. For example, we applied these techniques when we evaluated the security of the asynchronous XAP processor on the Springbank chip. The Springbank chip [Moore *et al.* (2002)] was composed of five different versions of the 16-bit XAP processor. One of those versions was an asynchronous implementation integrating some security mechanisms (like *dual rail encoding* and fault detection mechanisms) for smart-card applications [Moore *et al.* (2003)]. In the stage of characterizing the information leakage of such a processor, we applied DEMA techniques on the Electromagnetic waves emitted by the asynchronous XAP processor during a simple XOR operation.

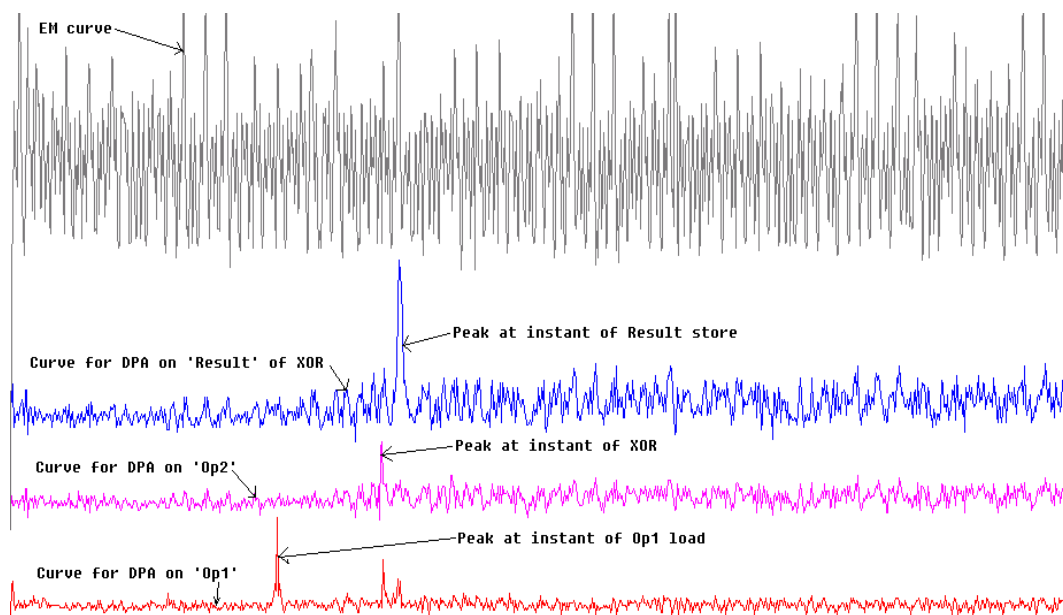


Figure 2.5: DEMA during an XOR on the asynchronous XAP

In Figure 2.5, the uppermost curve illustrates one of the Electromagnetic curves measured. The lowermost curve shows the resulting curve when DEMA was done on the first operand. The curve just on top of the latter is the same applied to the second operand. We can clearly see that DEMA peaks are obtained at the instants when these operands are manipulated and loaded. Finally an even higher DEMA peak is obtained when we cor-

related the Electromagnetic curves with the expected result of the XOR operation. Despite showing that the asynchronous XAP leaked information, we also showed that the leakage was still less important than for the synchronous XAP demonstrating some security improvement [Fournier *et al.* (2003)]. Other architecture-related attacks have recently been published. One example of such attacks is cache-based side-channel analysis as described in [Fournier & Tunstall (2006)]: in this case we show how, by analyzing the timing and power profile characteristics of *cache hits* and *cache misses* occurring during the table look-up stage of the AES algorithm, we can infer information about the secret key and hence reducing the exhaustive search spectrum for the secret key. Another example of an architecture-based attack as proposed in [Aciicmez *et al.* (2006a,b)] is based on the side-channel leakage induced by hardware branch prediction mechanisms.

Finally, another class of attacks are *Fault Attacks*. With this technique an attacker will try to corrupt the data being transported over a bus or stored in a register or memory at a specific time of a cryptographic calculation. From the results of a correct execution and those of a corrupted execution, the attacker will then try to retrieve part or all of the secret key by using techniques like *Differential Fault Analysis* as described in [Biham & Shamir (1997)]. To inject faults, an attacker will play on the physical conditions into which the chip is functioning by, for example, causing glitches on the power supply, or by irradiating the chip with a laser source or even simply light [Skorobogatov & Anderson (2002)]. Fault attacks can also be used to corrupt the correct flow of a program in order to make it take a given branch of the program or to cause memory dumps. For example, we experimented with such techniques on the asynchronous XAP of the Springbank chip. Using a laser source, we tried to corrupt the execution of the XOR operation at several instants in time. During most of the time, the security feature implemented in the dual rail encoding [Moore *et al.* (2002)] seemed to work in the sense that the faults generated were detected [Fournier *et al.* (2003)]. We then tried power glitches. The power was cut for a certain amount of time and restored. We made sure that once the power was restored the normal program flow was resumed.

In Figure 2.6, the upper curve shows a power profile during a normal execution flow and the lower curve shows a power profile when a fault has been injected. In the latter case, when the power is recovered, the “normal” flow of the program is resumed but in the meantime the data stored in some control registers seemed to have changed resulting in a large part of the RAM being dumped onto the serial port of the chip. The value contained in the register containing the size of the UART’s buffer had been corrupted leading to larger part of the RAM being dumped during the communication that followed the fault injection.

The attacks that we have been describing until now (invasive, side-channel and fault attacks) are powerful threats against security devices and against the sensitive cryptographic algorithms that they execute. We had the opportunity to put into practice two of them (side-channel and fault attacks) on the secure asynchronous XAP chip to actually realize how dangerous and efficient such attacks are if no countermeasure is applied. Security devices implement countermeasures against such attacks, whether they are of a hardware nature or of a software one. For obvious reasons of security and Intellectual

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

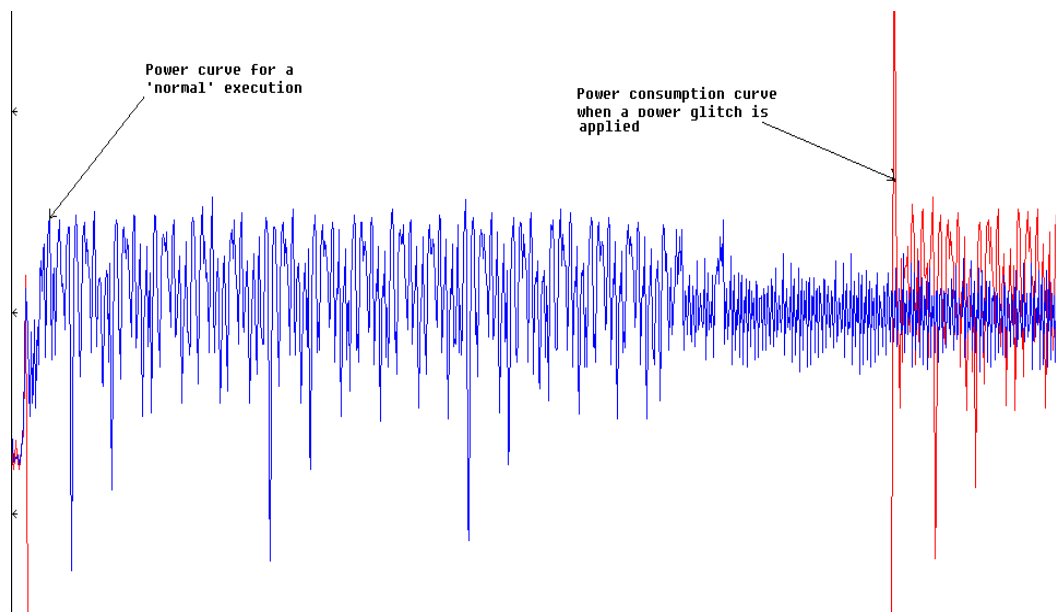


Figure 2.6: Power Glitch Attack on the asynchronous XAP

Property protection, little detailed information is publicly available about the countermeasures that are actually implemented in security devices like smart-cards or HSMs. This is why we mainly focussed on the mechanisms proposed by the research community.

Hardware Countermeasures. Against invasive attacks like those described in [Anderson & Kuhn (1996, 1997)], the authors in [Kömmerring & Kuhn (1999)] propose techniques like randomized clock signals or “randomized multithreading” to make synchronization more difficult, random combination logic to hinder reverse-engineering, use of sensors with built-in self tests and other design practices like removing all test logic and circuitry to avoid leaving a door open to easy probing. As we already mentioned, the use of asynchronous design techniques as described in [Moore *et al.* (2002, 2003)] can help to reduce side-channel information leakage and can provide coding techniques that can detect some types of fault injections. The efficiency and limitations of this latter technique has been demonstrated in [Fournier *et al.* (2003)]. This work on asynchronous circuits has been inspiring other research groups whose work are being now published. For example in [Kulikowski *et al.* (2006)], the authors propose a design tool that implements cryptographic processors in quasi delay insensitive asynchronous logic with the “balancing” of power consumption to hinder side-channel information leakage. The use of balanced logic has also been proposed in [Tiri & Verbauwhede (2003)] where the authors describe a balanced logic structure which they use to implement a DES hardware accelerator. Another proposed hardware solution is based on the insertion of parity checks on the data paths in order to detect errors introduced by malicious fault injections. In [Bertoni *et al.* (2002); Karri *et al.* (2003)], such techniques are applied on hardware implementations of the AES algorithm. Another way of handling the security problem in hardware is to analyze the circuit at design time and adapt the countermeasures until all attacks fail at least at simulation time. This design-time security analysis is proposed in [Li *et al.* (2005)]. The

problem with hardware countermeasures is that even though simulations may show that they are efficient, there is no absolute guarantee that this will be the case on the final chip where other considerations like final place & route, manufacturing processes etc. may have a huge impact on security. And by the time the final chips are obtained, if a security flaw is identified, it would be expensive to have another iteration of the design cycle to “patch” the design in hardware (in some cases metal fixes may be used but this would not apply to any part of the circuit). Or given that attack techniques are constantly being upgraded, it is difficult for hardware designers to keep pace with the evolving attack techniques [Clavier *et al.* (2000)]. It is hence highly recommended to use these hardware countermeasures with software ones.

Software Countermeasures. The basic principle behind most software countermeasures is

- to make sure that all the calculations’ timings are independent from the data or key being manipulated.
- to ‘hide’ the internal calculations of the cryptographic algorithm so that the attacker has no mastery of the data being manipulated and no means to understand what is happening.

In the case of Public Key Algorithms like RSA, to achieve constant timing for exponentiation algorithms, constant-time “square and multiply” algorithms have been proposed: in Figure 2.3 for example, the multiplication can always be done even when $d_j = 0$ where a ‘fake’ multiplication is executed. Likewise, for ECC, constant-time “double and always add” algorithms exist [Coron (1999)]. Or, as suggested in [Clavier & Joye (2001)], the use of addition chains can inherently provide a constant time exponentiation algorithm. To protect RSA against power analysis attacks as presented in [Messerges *et al.* (1999)], techniques like *message blinding* and *exponent blinding* have been proposed [Kocher (1996); Messerges *et al.* (1999)]. The principle behind the blinding technique is, prior to the exponentiation algorithm, the message is multiplied by a random value and to the exponent we add a random multiple of $\phi(n)$ ¹. With this the attacker cannot master the inputs to the exponentiation algorithm and is less likely to find any correlation between the power (or Electromagnetic even) curves measured and the inputs he fed to the algorithm. This is because the attacker is likely to perform a *chosen message attack* by having a small message to accentuate the difference between the power profiles of the square operation and those of the multiplication operation. Other message blinding techniques are proposed in [Akkar (2004)]. Similar blinding countermeasure techniques have been proposed for Secret Key Algorithms where this time the technique is called *data randomisation* or *data masking* [Akkar & Giraud (2001)]. The data and tables involved in algorithms like DES and AES are XORed with random masks in such a way that, again, the attacker cannot find any correlation between the power or Electromagnetic curves measured and inputs of the algorithm.

¹ $\phi(n)$ is the Totient function of n . Given that $n = p \times q$ and that p and q are primes then $\phi(n) = (p - 1)(q - 1)$

2.3 Architectures used in Embedded Systems

Application use-cases, certification schemes, cryptographic algorithms, attacks and countermeasures all help to define what is expected from security devices like smart-cards and HSMs. However the implementation options (that is their hardware and software architectures) can be numerous and diverse. In this thesis I mainly focus on the hardware issues because the performance and security issues pertaining to the sensitive cryptographic algorithms mainly depend on the way the latter interact with the hardware. The details of the hardware architectures of smart-cards and HSMs are often kept confidential due to the sensitive nature of the applications and also for Intellectual Property protection reasons. Yet, the general outlines of the hardware architectures are available enough to help us proceed during the exploration phase of our research work. Moreover, a few scientific publications do provide insight into what could lie underneath the ‘golden’ contacts of a smart-card or the thick metal shields of an HSM. I further narrowed my interest to embedded security devices (most of which today are smart-cards) because such devices face challenging constraints of cost, area, power, performance and security: area because it is directly correlated to the cost of the chip and that the latter has to be cost effective (for smart-cards, the chip’s surface area is limited to $25mm^2$); power because the batteries used today in portable devices like mobile phones have a limited energy storage capacity (for example, GSM SIM cards can consume a maximum of 6mA peak current); performance because the processing time of the device has to be “user-friendly” (i.e. not take too long); and security because such embedded chips are within the reach of everyone, in particular of hackers. In the next paragraphs we shall be talking about performance for an RSA signature which will be given at different frequencies.

The chosen point of comparison in this thesis is the number of clock cycles taken for a 1024-bit modular multiplication. In order to compare the architectures discussed in this thesis, for cases where the actual number of clock-cycles is not given, extrapolations are made. For this we make the following assumption: if we suppose that in a 1024-bit value we have as many zeros as there are ones, then doing a exponentiation would result in doing 1024 squares and 512 multiplications. If we further suppose that the square is implemented as a multiplication, then a 1024-bit RSA signature could be approximated to 1536 modular multiplications. Then the number of clock-cycles taken by a 1024-bit modular multiplication is given by the following equation (f is the number of MHz of clock frequency and t is the number of microseconds taken by the RSA signature):

$$T_{1024} = \frac{f \cdot t}{1536} \text{ cycles} \quad (2.5)$$

Most smart-cards consist of the building blocks illustrated in Figure 2.1: a CPU (or microprocessor) to run the OS or execute simple operations, ROM to store the OS and constant data, NVM (which can be EEPROM or Flash) to store other pieces of code and personalization data like secret keys, RAM to store transient data, cryptographic engines to accelerate the computation demanding cryptographic algorithms, the interfaces (which can be contact or contactless) to communicate with the external world among other things like power regulators, timers or data and address buses [Rankl & Effing (2000)]. Most of the smart-cards on the field today embed 8-bit microprocessors, having a CISC architec-

ture, usually based on an Intel 8051 Instruction Set Architecture (ISA): for example the **SmartMX™** products from Philips Semiconductors (now called NXP), the **SLE66** from Infineon or the **ST16** from ST Microelectronics. A few smart-cards have 16-bit CPUs like in the **SmartXA™** family from NXP. Today's high end smart-cards embed 32-bit microprocessors. To our knowledge, the first 32-bit smart-card was designed in the CASCADE project between 1993 and 1996 [Dhem (1998)]. In this project the smart-card was based on an ARM7 processor. Since then ARM has been developing its offer for microprocessors for security devices through its **SecurCore™** family, of which chip manufacturers like Atmel, Samsung or Philips are publicly known licensees. MIPS Technologies has also developed a secure version of its MIPS-IV architecture, the **SmartMIPS™** [MIPS (2005)] which has been deployed in the **HiPerSmart™** chips of Philips. Other chip manufacturers have been developing their own 32-bit CPU like Infineon in its **SLE88** chips [Infineon (2003)].

The latter 32-bit microprocessors can also be used to run cryptographic operations. However, in order to improve the performance of some commonly used, computation intensive cryptographic algorithms, cryptographic engines are usually added to the smart-card architecture. The engines usually appear as ASIC-like co-processors, slave to the main microprocessor. For Secret Key algorithms like DES or AES, the implementations can be rather straight-forward and easy to implement [Gaj & Chodowicz (2002)]. Some flexibility may be found in the hardware countermeasures that can be added (like those proposed in [Kömmerling & Kuhn (1999)]) or in the decomposition of the cipher's structure into pipelines to improve throughput. For Public Key algorithms, there is a high degree of mathematical flexibility in the way basic operations like modular multiplication may be implemented (e.g. using Montgomery's method or Quisquater's method, Barrett's method or Sedlak's approach, all of which are explained in [Dhem (1998)]). The first (and only one to our knowledge) survey made on commercial Public Key accelerators for smart-cards was carried out by Naccache and M'Raihi in [Naccache & M'Raihi (1996)]. Even though this survey relies on figures collected more than 10 years ago, this paper provides a good panorama of the main families of Public Key accelerators implemented on smart-cards back then. Each chip manufacturer made design choices based on a specific reduction algorithm, for example Montgomery's method for Motorola and Thomson (ST), Quisquater's method for Philips (NXP), Barrett's method for Amtel and Sedlak's method for Siemens (Infineon). Some of the *best-in-class* solutions presented in this paper are

- the one from Philips which proposed an accelerator of $2.5mm^2$ in $1.2\mu m$ CMOS technology with which a 1024-bit signature could be generated in $2000ms$ with a clock of $5MHz$, which, by equation 2.5, gives a T_{1024}^{PHS} of 6510 clock-cycles.
- the one from Siemens (on an SLE44) whose area is $24.5mm^2$ in $0.7\mu m$ CMOS with a performance of $630ms$ for a signature on 1024 bits at $5MHz$. By equation 2.5, we have a T_{1024}^{IFX44} of 2050 clock-cycles.

In the latter paper we see that the crypto-accelerators were already limited in the size of RSA keys that could be used (at that time, the common sizes were 512-1024 bits). Two years later [Handschuh & Paillier (2000)], Handschuh and Paillier proposed a follow-up to [Naccache & M'Raihi (1996)] with some updated information including

- on an ST19 chip, with an internal clock of $10MHz$, a signature of 1024 bits was executed in $380ms$ which allows us to infer that by equation 2.5, we have a T_{1024}^{ST19} of

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

2473 clock-cycles. A signature of 2048 bits could on the same chip be done in 780 ms.

- on an SLE66 chip, at 5MHz, a 1024-bit signature was done in 880ms (and hence we have a T_{1024}^{IFX66} of 2864 clock-cycles) while a 2048-bit signature took 1475ms.

Some performance figures can be obtained for some more recent architectures from the manufacturer's websites:

- **MIPS4KscTM** (general purpose processor): 1024-bit RSA signature (with CRT and using Montgomery's modular multiplication) performed in less than 15ms with a clock frequency of 200MHz [MIPS (2005)]. By equation 2.5 and since the modular multiplication was implemented using the CRT method we have the time for a 512-bit modular multiplication $T_{512}^{MIPS} = 1953$ clock-cycles. Since it is Montgomery's method applied to a 32-bit architecture, doubling the data size would mean multiplying the time taken by 4. Hence we would have $T_{1024}^{MIPS} = 7812$ clock-cycles.
- **ARM SC200** (general purpose processor): 1024-bit RSA signature without CRT in 980ms at 20MHz [ARM (2002)]. By equation 2.5, we have $T_{1024}^{SC200} = 12760$ clock-cycles.
- **Philips SmartMx** (contactless smart-cards): 2048-bit RSA in 25ms (asynchronous *Tangram* implementation of the FAMEXe), 128-bit AES in 11 μ s and 3-key TripleDES in less than 50 μ s [Philips (2004)].
- **Infineon Crypto@1408** (on SLE88CFx4000P): RSA signature (with CRT) on 1024 bits in 14ms and on 2048 bits in 58ms at 66MHz [Infineon (2003)], which means that $T_{1024}^{IFX88} = 1245$ clock-cycles.

Note that in most of the above examples, the 1024-bit signatures are done in the straight-forward conventional way while the 2048-bit signatures were done using the Chinese Remainder Theorem (CRT). In the latter method (as given in Figure 2.4), in the case where $n = p.q$, the multiplications are done (mod p) and (mod q) on data lengths which are twice as short and the individual results are re-combined to give the end result (mod n). With such a method, 2048-bit modular multiplication can be done even on a hardware that was initially limited to 1024-bit data. A more recent survey was proposed on Public Key accelerators by Batina *et al.* in 2002 (published in 2003) [Batina *et al.* (2003)]. In this paper, the authors focus more on the architectures that have been proposed and published by the Research community. This paper mentions two other approaches to the design of Public Key accelerators. The first one is based on the use of *systolic arrays* for performing modular multiplication. In this approach bit multipliers are arranged in a rectangular array to allow a matrix multiplication between the two operands while doing the reduction as well. One can easily foresee that by resizing the size of the matrix, one could re-scale such systolic arrays to work on data of any length. This leads to the second approach mentioned in [Batina *et al.* (2003)] which is that of scalable architectures like the ones proposed in [Savaş *et al.* (2000); Tenca & Çetin K. Koç (1999)]. In the latter papers, Koç *et al.* propose an architecture that implement Montgomery's algorithm where scalability is done at the level of the size of the modular multiplier that is implemented. In [Batina *et al.* (2003)] the authors illustrate that the same approaches

have been applied to accelerators for Elliptic Curve Cryptography, specially with dual field multipliers as described in [Gutub *et al.* (2003); Savaş *et al.* (2000)].

Another attractive approach for the implementation of cryptography in embedded chips is instead of having bulky, highly power consuming co-processors, we could add dedicated instructions to more General Purpose Processors. A commercially known example for this is the SmartMIPS™ which is a derivative of the MIPS-IV architecture to which instructions have been added to accelerate both Secret Key and Public Key algorithms [MIPS (2005)]. A similar approach is proposed in [Großschädl & Kamendje (2003)] while in [Eberle *et al.* (2005)] the authors illustrate how dedicated instructions can be added to a Sparc processor in order to accelerate Elliptic Curve Cryptography in Binary Fields.

2.3.1 Classification of computer architectures

In order to address the constraints of performance, size, power, scalability and security pertaining to embedded cryptographic hardware platforms, let us first have a look at the possibilities in terms of high performance architectures.

In [Flynn (1995)], Michael Flynn mentions two factors that define the architecture of a processor, namely

- the definition of the hardware resources, or the “*model*”, of the processor.
- the type of work that must be performed by the processor, in other words the “*instruction level workload*”.

These two elements form the orthogonal axes of an architecture design space that Flynn defined in [Flynn (1972)]. Such a design space, known as *Flynn’s taxonomy* is illustrated in Figure 2.7.

In his decomposition, Flynn distinguishes among four families of computer architectures:

- **SISD** (Single Instruction Single Data) would refer to ‘uniprocessors’ where a processor executes only one stream of program. The latter program simply consists of a sequential flow of instructions where each instruction works on a single set of data at a time (performs only one calculation at a time). Scalar processors like Motorola’s 68000, Intel’s x86, the ARM or the MIPS are typical examples of SISD architectures.
- **SIMD** (Single Instruction Multiple Data) would also execute from a single stream of instructions but each instruction is defined to work on several data elements in parallel. The same calculation is performed on different sets of data at the same time. Examples of SIMD architectures, also called *Data Parallel Architectures*, are vector processors. The logic behind SIMD processors comes from the observation that in most software loops the same operation is performed on several data and that having a hardware that could be programmed to apply these operations to a set of data in parallel would result in significant performance gains through, for example,

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

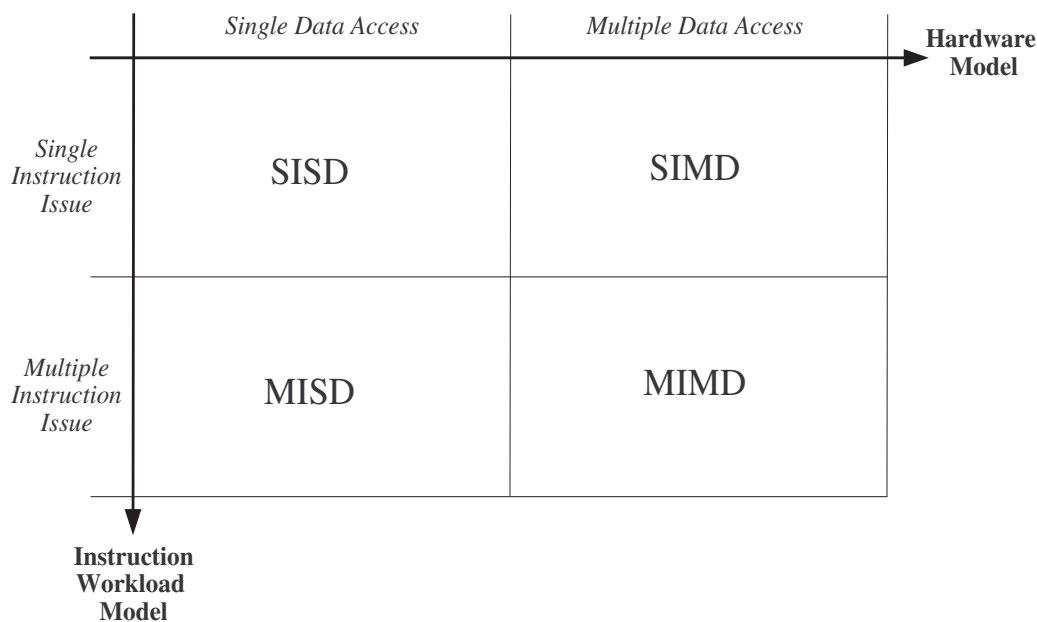


Figure 2.7: Architecture Design Space based on *Flynn's taxonomy*

the removal of the “loop controls”. Hence this helps to improve code density and boost instruction issue rate.

- **MISD** (Multiple Instruction Single Data) would not correspond, to my best knowledge, to any real application today. One could, to some extent, assimilate redundant fault tolerant systems (as used in planes or space craft and satellites) to such an architecture. A given operation would be done in different ways in parallel on the same set of data to guarantee fault tolerant results.
- **MIMD** (Multiple Instruction Multiple Data) Multiple Instruction Multiple Data would refer to multi-processor cores where several program streams are executed in parallel on the individual processors, with or without synchronization between the different processors, for example dual processor cores like ARM's MPCore, IBM's Power4 or Intel's Core Duo.

The above architecture design space can be slightly modified to illustrate the notion of *Instruction Level Parallel* (ILP) architectures as illustrated in Figure 2.8. In such architectures, the program is executed from a single flow of instructions but each instruction can be considered as a group of independent ‘scalar-like’ instructions working on separate data in parallel. This is why, in Figure 2.8, ILP architectures are put as an intersection between the SIMD space and the MIMD space. The ‘shaded’ regions illustrate ‘uniprocessor’ architectures.

One type of ILP is the VLIW (*Very Long Instruction Word*) architecture. For a such processor, the parallelism among the instructions is determined at software compilation time. The compiler is tailored to suit the hardware resources available and to determine

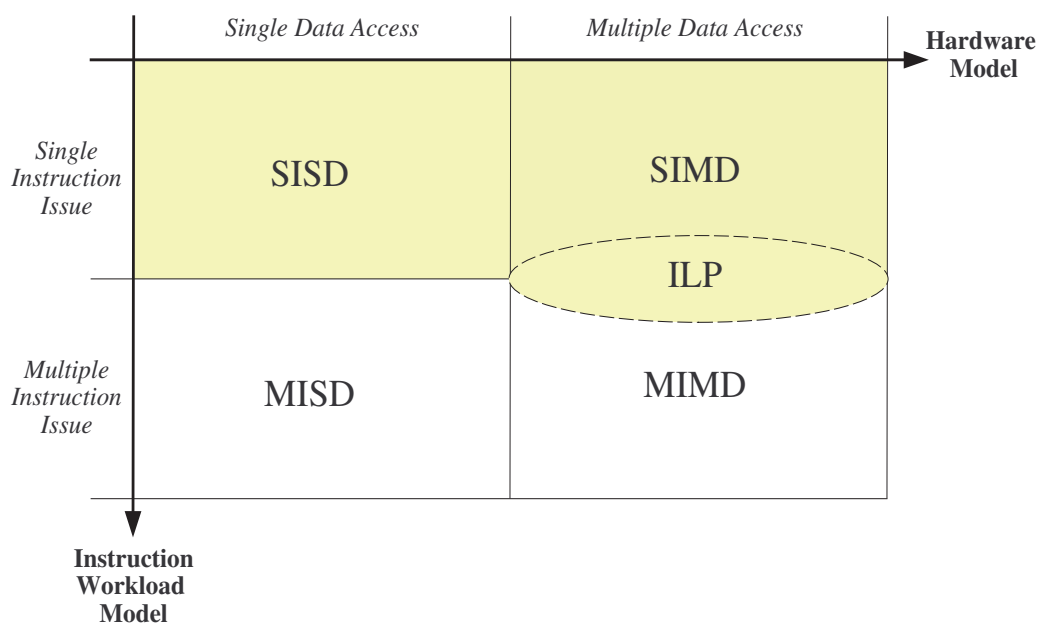


Figure 2.8: Modified Architecture Design Space

the data dependencies. Then the instructions that can be executed in parallel are re-grouped by the compiler into VLIW machine words. The processor will then fetch one “machine word” at a time. The hardware itself does not have to manage the scheduling among the instructions as this would have been taken care of by the compiler. In contrast to that, in *Superscalar* architectures, which are also ILP architectures, everything is managed by the hardware [Flynn (1995); Hennessy & Patterson (2003)]. The parallelization is determined at execution time by the hardware itself. The compiler just compiles and links the instructions as a sequence of machine words. During execution, the processor will pre-fetch several instructions and dynamically determine the dependency among them and will dispatch them to the parallel computation units.

2.3.2 Vector architectures

In Chapter 3, I will explain why and how cryptography can be implemented on a vector co-processor by decomposing the operations on long precision numbers into parallel operations on smaller data units. This comes from the observation that when cryptography is implemented on a scalar (*Sequential*) processor, especially for Public Key algorithms, the resulting code involves loops working on elementary data units.

Vector processors are designed to work on linear arrays. A typical vector architecture can be considered to be organized into elementary *lanes* [Asanović (1998)] which can be replicated to offer a scalable degree of parallelism. Each lane would consist of at least one computation unit (also called a Vector Processing Unit - VPU), vector registers, control units and internal RAM memory (which can be caches). The lanes could then be made to interface with a shared memory or with a scalar control processor. Vector processors can be *Memory-Memory Vector Processors* where the vector instructions work

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

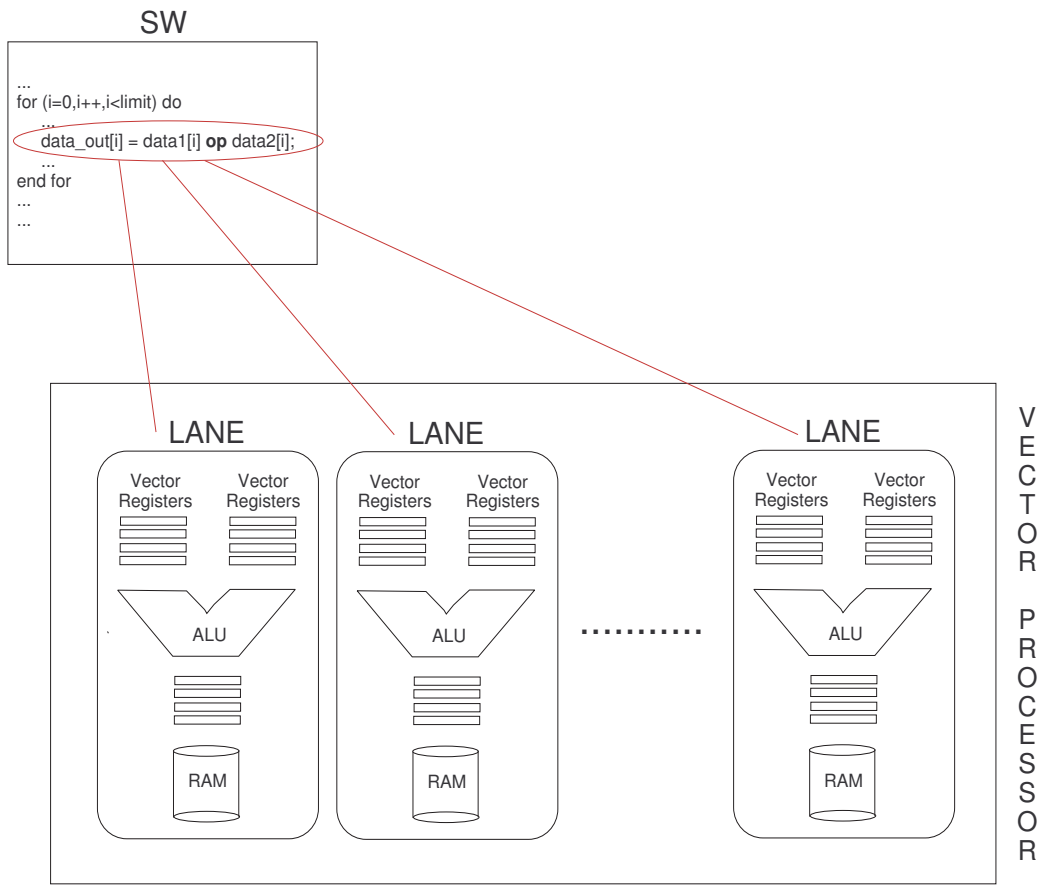


Figure 2.9: The Vector Processing Paradigm

directly on data banked in memories or *Register-Register Vector Processors* where the vector instructions work on data buffered in registers. Vector processors are characterized by their vector instructions such that

- each vector instruction replaces a software loop (as illustrated in Figure 2.9) where the computation of each result is independent from previous results.
- each vector instruction corresponds to an entire software loop, hence reducing instruction bandwidth.
- control hazards are reduced as loops are removed and so are the branch controls associated with them.

Other advantages of vector architectures, as described in [Espasa *et al.* (1998)], are the simple control units, the deterministic timing of vector instructions thus simplifying their scheduling, the possibility of switching off other blocks during vector computations and scalability both in terms of performance (by increasing the number of Vector Processing Units (VPUs)) and in terms of balancing performance in terms of power consumption. In [Flynn (1995)], Flynn points out that the performance of a vector processor is also determined by the throughput of each VPU (which in turn depends on its pipelining),

the number of vector registers, the data caching mechanism between the scalar part and the vector processor, the vector start-up cost (which also depends on the pipeline) and above all, from the software perspective, the amount of the program that can be effectively transformed into vector operations. Asanovič [Asanovič (1998)] explores how the performance of a vector processor is determined by the properties of the software it runs in terms of:

- data dependencies, i.e. how much a given calculation depends on the previous one.
- name dependencies, i.e. what are commonly known as *Data Hazards*, i.e. whether a given operation will try to access a given register that has just been modified.
- control dependencies, i.e. to what extent control instructions like branches depend on the instructions being executed.

The latter issues also depend on the compiler techniques that are used. Compiler techniques and optimizations for extracting data parallelism out of a scalar software is well understood. Some of these techniques, including common techniques like register renaming or instruction re-ordering, are summarized in [Padua & Wolfe (1986)].

The availability of efficient compilers has been a constant major issue in the history of vector processors. For example it was one of the drawbacks for the legendary Cray machine. The CRAY-I [Russell (1978)] machine was among the first super-computers to implement vector processing techniques back in 1977. The CRAY-I had 12 functional units, 16 memory banks and 4KBytes of registers. It had an instruction issue rate of 1 instruction per clock cycle and could target a clock of 80MHz. The CRAY-I could reach a throughput of 138 million FPOS (Floating Point Operations per Second) on average and 250 million FPOS when in a burst mode. Another example of a “historical” vector processor was the vector part that was added to the IBM370 [Padegs *et al.* (1988)] as an extension to the scalar processor. Then there is of course the MMX extension to the Intel architecture [Peleg & Weiser (1996)] for multimedia applications. By then, there were clear indications that vector processors had stopped being the ownership of the “super-computing” world and that they were being vulgarized onto the world of microprocessors, especially for digital signal processing. In [Kozyrakis & Patterson (2002)] the authors present the VIRAM architecture (32 vector registers, 4 lanes with 2 arithmetic units and two DRAM banks per lane) where they show that vector processing techniques were better suited than VLIW approaches for multimedia applications: smaller code, twice as fast, 10 times less power consumption and more scalable. Further examples are given in [Esapasa *et al.* (2002)] where the authors propose a Floating Point vector extension (consisting of 32 functional units) to the Alpha processor. In [Krashinsky *et al.* (2004)], the authors present a vector microprocessor, the SCALE-0 which consists of a scalar 32-bit MIPSII-compatible-ISA processor and a Vector Thread Unit which consists of a command management unit, a stream unit and 4 parallel lanes. Each lane consists of an instruction buffer, computation units, local register files and inter-lane data transfer units. This vector microprocessor can work at 400MHz to reach a throughput of 3.2 GLOPS (Giga Floating point Operations Per Second).

2.3.3 Cryptography on parallel architectures

When looking back at the previous work on embedded parallel processors, most of them targeted applications like multi-media signal processing applications. Attempts at having a parallel approach to the implementation of cryptography have been relatively scarce, not to mention attempts to actually design a parallel cryptographic engine.

Most of the known examples of parallel implementations of cryptography are in non-embedded computers. In the latter ‘conventional’ or non-embedded computing world, most of the research has concentrated on parallelizing the cryptographic operations in order to take advantage of the SIMD architecture originally developed for media applications [Aoki *et al.* (2001)]:

- In [Page & Smart (2004)], the authors implement a long precision modular multiplication on a Pentium4 using the SSE2 (*Streaming SIMD Extensions 2*) instructions. The authors execute four exponentiations in parallel, each exponentiation being implemented using a Redundant Representation of Montgomery’s multiplication. The authors report that a 1024-bit modular multiplication takes $60\mu s$, which roughly corresponds to 120000 clock cycles for a 2GHz Pentium4 processor.
- Crandall and Klivington illustrate in [Crandall & Klivington (1999)] how the Velocity Engine of the PowerPC can be used to implement long precision multiplications for RSA. Based on the figure given in the paper, we can infer that a 1024-bit multiplication takes about 3600 clock cycles with their approach. However, no figures were reported for a full modular multiplication.
- The AltiVec [Diefendorff *et al.* (2000)] extension to the PowerPC was originally developed to target media applications. This vector extension is made of 32 128-bit vector registers. AltiVec also offers some superscalar capabilities since instructions belonging to different ‘classes’ can be executed in parallel. Galois Field arithmetics have been implemented on the AltiVec in [Bhaskar *et al.* (2003)]. In the latter paper, the authors show how the Rijndael algorithm [NIST (2001b)] can execute in 162 clock cycles on the AltiVec or, even better, in only 100 clock cycles if a bit-sliced approach is used.
- Parallel approaches have also been studied in order to factorize large integers for “breaking” curves used for Elliptic Curve Cryptography [Dixon & Lenstra (1992)]. Implementations on the Fujitsu AP100 (with 128 RISC processors working in parallel), on the Fujitsu VP220/10 vector processor and on the Fujitsu VPP500 (made up of four interconnected vector processors) consist in having each processor perform calculations independently until a factor is found [Eldershaw & Brent (1995)]. But the operations used during each calculation (for example the multiplications and divisions) are not themselves vectorized.

For embedded applications, studies around the use of SIMD architectures for cryptography are even more scarce:

- In the embedded world, Data Parallel architectures are mostly deployed in DSPs (*Digital Signal Processors*) for signal processing. In [Itoh *et al.* (1999)], the authors

describe how modular multiplication based on Montgomery's method [Montgomery (1985)] can be implemented on a TMS320C6 201 [TI (2004)] DSP. With their approach, a 1024-bit RSA verification (with $e = 2^{16} + 1$) takes 1.2ms. If we suppose that we need approximately 17 modular multiplications for this, then, with a processor clock at 200MHz, we can infer that the one 1024-bit modular multiplication takes about 14000 clock cycles. Applying the same reasoning to other data given in the paper, we find out that one 2048-bit modular multiplication takes 53000 clock cycles on this architecture.

- In the fascinating world of smart-cards, some work on parallel architectures for cryptography has been reported in [Fischer *et al.* (2002); Izu & Takagi (2002)]. In both papers, the authors focus on fast elliptic curve multiplications. In [Izu & Takagi (2002)], the authors show how, with a projective coordinates representation [Blake *et al.* (1999)], calculations can be parallelized on the Crypto2000. On the other hand Fisher *et al.* [Fischer *et al.* (2002)] focus more on elliptic curve implementations resistant to side channel attacks.
- Some research groups are currently undertaking some work on designing parallel architectures for cryptography. For example, very recently, Sakiyama *et al.*, in [Sakiyama *et al.* (2006a,b)] have proposed a programmable superscalar scalable co-processor for fast Elliptic Curve Cryptography. In this work, the co-processor is designed to interface with an 8051 processor. The core of the approach revolves around the design of a modular multiplier whose data sizes are scalable. However, the modular multiplier is designed for one particular curve. The design of this co-processor required the design of a dedicated tool called **Gezel** for simulation and performance analysis.

2.4 Conclusion on Security Devices

The emergence of new applications like Digital Rights Management (DRM), on-line payment transactions and mobile banking has given birth to a new breed of security devices which can be viewed as the convergence of small portable devices like smart-cards and larger ones like HSMs. Such devices are required to run a whole plethora of cryptographic algorithms, namely Hash functions, Secret Key or Public Key algorithms and offer tamper resistance capabilities to protect the execution of cryptographic algorithms and to guarantee the confidentiality of the secret data being manipulated. The level of security is measured against the different attacks that can be performed during the execution of such algorithms.

From a hardware point of view, side channel and fault attacks are the dangerous ones. The countermeasures against such attacks can be implemented in hardware or in software. With respect to all of these requirements, we went back to exploring the architecture design space in order to seek for an architecture that would offer a choice between security, performance, area and power for a particular application. Vector architectures seem to be suited to that kind of approach: performance, area and power can be controlled by varying the number of units working in parallel while the security part can be handled

2. SECURITY DEVICES: FUNCTIONS & ARCHITECTURES

by making sure that the hardware performance figures can good enough to implement all the required software countermeasures. Until now, vector architectures have been mainly deployed for media applications like in DSPs. There have been attempts to implement cryptography on DSPs but to the best of my knowledge, no one has tried to define a vector architecture for cryptography.

Chapter 3

Vectorizing Cryptographic Algorithms

Security chips are widely deployed today in products like smart-cards, hardware security modules (used in secure readers or trusted platforms, ATMs or secure personalization facilities for such secure products), electronic passports and tags. Given the different physical, power, size constraints pertaining to those different cases, the corresponding chips used may have totally different architectures, leading to different designs for the security and cryptography modules and hence different performance characteristics and levels of security.

In this chapter, I first define the security and cryptography requirements upon which to focus in order to propose a generic architecture applicable to all of these cases. I then present the advantages of having a vector processing approach. Next I talk about the cryptographic algorithms on which this thesis focuses: the AES (Advanced Encryption Standard) and ECC (Elliptic Curve Cryptography). In these 2 cases, I study how the internal calculations can be modified to suit a data parallel architecture and define the required vector instruction set architecture.

3.1 Requirements for Secure Chips

When it comes to cryptography and security, devices like smart-cards are bench-marked on the following items:

1. The performance of the critical cryptographic algorithms, i.e. the speed with which the algorithms like DES (Data Encryption Standard), AES and 1024-bit RSA (Rivest-Shamir-Adleman scheme) based signature/verification are executed. 2048-bit RSA and Elliptic Curve based Public Key Cryptography are starting to be deployed.
2. The level of security of the chips, i.e. their ability to resist invasive attacks like probing, non-invasive attacks like Power or Electro-Magnetic Analysis or semi-invasive attacks like fault injections.

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

These items have been discussed in Chapter 2. In the following sections, I explain which of these factors are addressed in this thesis and why.

3.1.1 Requirements for cryptographic applications

The performance of cryptographic algorithms is not only linked to the structure of the algorithm itself but also to the resources available on the processor chip and above all, in particular for embedded systems, to the energy consumed by the resulting chip. For embedded processors, power consumption determines the maximum frequency at which the chip is clocked. In such cases we have to look for the best trade-off between hardware and software. Designers of secure systems must be able to achieve a high degree of flexibility (in terms of choice of algorithm), high performance and a low power-consumption.

In this study, the benchmarks used look at only some of the most commonly used cryptographic algorithms and do not target all the algorithms used in protocols like SSL [OpenSSL (2000)] or PKCS#11-v2 [RSA (2000)]. Moreover, most cryptographic algorithms can be viewed to use the same ‘basic’ operations like permutations, table look-ups or modular multiplications.

Cryptographic algorithms can be classified into three families¹:

- Secret Key (SK) algorithms like DES, AES, RC2/4/5/6, IDEA, Skipjack which are fast encryption/decryption algorithms, usually involving data/key blocks of several hundred bits.
- Hash Functions which map binary strings of arbitrary lengths to one of fixed length, examples of which are MD5 and SHA1
- Public Key (PK) algorithms like RSA, DSA, DH or ECC which are mainly used for authentication, signing and verification.

By grouping the algorithms like that, we are able to identify basic operations that are common to them and which would be worth implementing as specific instructions.

Secret key algorithms and Hash functions

We regroup SK algorithms and Hash ones because they require more or less the same basic operations. Identified operations/instructions are:

Cyclic Shift Operations Data dependent/independent variable ‘rotation’ operations are used in most of these algorithms. Such instructions are usually already implemented on GPP (General Purpose Processors).

¹Random Number Generators (RNG) have deliberately been put aside as they are more optimally implemented as separate peripherals since they usually have physical and timing characteristics of their own.

Permutations Permutation operations are a useful way of shuffling bits within a word, as used in algorithms like DES [NIST (1993)]. Work on efficient permutation instructions already exists. In [Shi *et al.* (2003)], the authors implement a 64-bit permutation in 2 clock cycles and in [McGregor & Lee (2003)], the authors work on permuting 64 bits with repetition in 11 clock cycles. In both cases, these permutations are done at the expense of quite cumbersome hardware. There could be room for progress if for example we are working on smaller data paths (16-32 bits) or also find a better trade-off between the cost of the hardware and ‘speed’ at which these permutations are done - like probably working on smaller ‘butterfly structures’ as exposed in [Shi *et al.* (2003)].

Table Look-ups This is another widely used operation in SK algorithms. The principle used is extremely simple but implementing this in software is very time and memory consuming. Most, if not all, of these substitution operations work on bytes. If we have a 32-bit architecture, for example, the challenge is to perform 4 byte substitutions in parallel from *any* address. Performing packed memory accesses through one instruction have recently been proposed in [Fiskiran & Lee (2005)].

Byte-wise modular operations Algorithms like AES [NIST (2001b)] involve the addition and multiplication of bytes in $GF(2^m)$. It would be interesting to have such operations modulo any primitive polynomial of degree 8. Better still if, say we are on a 32-bit architecture, we can manage to perform four of those in parallel.

Public Key Cryptography

This class of cryptographic operations involves the use of long precision numbers. A significant amount of work has already been done on how to manipulate long precision numbers on GPPs with ‘small’ data paths [Dhem (1998)]. When we look at the modular multiplication of two large numbers in RSA or that of two long polynomials in ECC, we see that there are several efficient ways of implementing such operations. As explained in [Dhem (1998)], the Barrett algorithm (and one of its variant which is the Quisquater algorithm) along with Montgomery algorithm [Montgomery (1985)] constitute the most interesting possibilities. Both algorithms involve steps where we perform word by word multiply and add operations. The authors in [Tenca & Çetin K. Koç (1999)] have already looked at ways of tailoring multipliers to suit the Montgomery Multiplication. They make the multiplier ‘scalable’ by resizing the basic multiplier that can be replicated to suit the data path needed. In [Savaş *et al.* (2000)], the same authors go from their first idea and make the functional block suitable for multiplications both in $GF(p)$ and $GF(2^m)$. In this case, the design of an appropriate dual-field multiplier is mandatory. The challenges lie in the trade-off to be found between the data path (in other words the size of the word to work on), the design’s size and its performance, without impacting the execution of simpler basic operations.

Another time-consuming operation which is often overlooked is loop-counting. This could apply to any kind algorithm in general but it is particularly true for PK algorithms where multiplication of two long numbers involves the use of nested loops. Hence counter’s increment and decrement, test instructions and branch instructions are often

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

called thousands of times during a Public Key signature, exponentiation algorithm for RSA or multiplication by a scalar for ECC. In terms of performance, a panorama of the calculation speeds of some commercially used security devices is given in Section 2.3.

3.1.2 Security Requirements

As explained in Chapter 2, the security of devices like smart-cards are measured by certified security laboratories who perform tests based on requirements set by Certifying bodies like the Common Criteria Certification [CC (1998, 1999)] or the FIPS [NIST (2001a)]. These processes not only focus on the security of the hardware used, but also on the security of the software and protocols used. Since this thesis focusses on the hardware architecture, we concentrate on the different threats and attacks linked to the hardware itself.

Invasive Attacks

Invasive attacks require that we have physical access to the chip for doing things like micro-probing, scanning memories to identify bit values, scanning the chip to reverse-engineer architectures or physically modifying the values contained in memory cells or registers. To circumvent such attacks, one needs to work on the technology itself and on the manufacturing processes used: use of ‘intelligent’ defense grids, use of special manufacturing or lay-out processes among others, which are outside the scope of this thesis.

Side channel information leakage

Since the first publications on timing attacks [Dhem *et al.* (1998); Kocher (1996)], side-channel information leakage has been of primary concern to designers of secure systems. By externally observing the variations in power consumed, timings taken or Electro-Magnetic waves emitted, one can infer information about the type of operations being executed, the value of the bits being manipulated or the value or Hamming Weight of the data¹ being manipulated. Recently, an interesting quantitative method of measuring the side-channel information leakage, called Correlation Power Analysis (CPA), has been proposed in [Brier *et al.* (2003, 2004)] whereby, based on statistical methods, we can compare different systems by looking quantitatively at how their side-channels correlate with the value of the data being manipulated. Other architecture-related attacks, like cache-based side-channel analysis [Bernstein (2004); Bertoni *et al.* (2005); Fournier & Tunstall (2006); Page (2002, 2004); Tsunoo *et al.* (2003)] or branch prediction attacks [Aciicmez *et al.* (2006a,b)], have been proposed to illustrate how a chip’s architecture can also be used to perform side-channel attacks.

The *sine-qua-non* conditions for performing successful side-channel attacks are

- Having a deterministic process where we can identify the portions of code being executed in order to synchronize our measuring tool with the part that we are

¹The Hamming Weight of a data corresponds to the number of ones present in its binary representation.

interested in.

- Having an architecture whereby the time taken, the power consumed or the EM radiated depends on the data (whether its value or its Hamming Weight) being transported over the buses or stored into registers.

A potentially secure design should take the above factors into consideration by, for example:

1. having functional units and data paths whose timings are independent from the data or instruction.
2. having a scheduling that is random from one execution to another in order to ‘hide’ the actual processes being executed. For example random instruction scheduling has been proposed in [Leadbitter *et al.* (2007); May *et al.* (2001)]. A judicious trade-off has to be found between the timing penalty induced by such randomization and the fact the latter has to resist to any statistical analysis by an attacker.
3. having a data path which is able to ‘hide’ the value or Hamming weight of the data it transports and uses.

Fault Attacks

The ‘timing issue’ is also vital when it comes to attacks linked to the generation of faults inside a chip. Fault attacks consist in corrupting the data being transported over a bus or stored in a register or memory at a specific time to corrupt a sensitive calculation in order to bypass a security mechanism or to retrieve part or all of the secret key being used (by differential cryptanalysis for example [Biham & Shamir (1997)]). Hence, to have a successful fault attack, one must have a device onto which any fault injected remains undetected and one needs to be able to synchronize with the program so as to corrupt the part of the cryptographic algorithm to allow to ‘cryptanalyze’ or reverse-engineer the latter.

In practice, such a mandatory synchronization is done by observing external physical characteristics like timing, power consumption profiles or Electro-Magnetic waveforms. In that sense, protecting against fault analysis and protecting against side-channel information leakage could lead to a common set of countermeasures built on processes’ randomization for example .

The second aspect to consider is how to construct a fault resistant system. We then face the terrible dilemma of either having a fault-detecting system or a fault-correcting one. The latter being extremely tedious and expensive to make, we could favour the study of a fault-detecting architecture. The first right steps have already been taken in systems like Dual Rail encoding as depicted in [Moore *et al.* (2002)] where the principle was adapted to an asynchronous circuit. The security evaluation of such a system has then been carried out [Fournier *et al.* (2003)]. The latter study showed that, putting aside the ‘design flaws’ for which solutions have since been proposed, dual encoding is effective in detecting certain types of faults but that complementary solutions have yet

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

to be sought. In fault-tolerant systems where the fault is corrected, we must assume that the system is first able to detect the fault. However, initial studies show that the simplest of error/fault-correcting systems would require a circuit which is twice bigger, which is unacceptable price to pay for embedded processors.

As a first approach, detecting the fault would be enough. Most fault attack scenarios exploit the faulty result obtained when a cryptographic algorithm is corrupted. Detecting the fault involves a mechanism that triggers an exception/alert upon **any** attempt of fault attacks. We could then leave it to the Operating System or software to securely manage that exception to stop the process for example.

3.1.3 Introduction to the detailed case studies

We have browsed through the performance and security issues for processors executing cryptography. From a purely hardware point of view, while the first aspect can be handled in a universal sense, the security aspect is very much architecture and technology dependent. Moreover, a lot of work has already been done to propose software countermeasures for most of the commonly used cryptographic algorithms [Akkar & Giraud (2001); Bertoni *et al.* (2002); Clavier & Joye (2001); Zambreno *et al.* (2004)]. Hence from now on, we will mainly focus on the performance aspect while providing flexibility for the implementation of these software countermeasures. In later sections, we focus on two case studies where we evaluate the performance of these two algorithms on a general purpose processor, identify the bottlenecks and identify in what way a data parallel approach could improve the performance of those algorithms on such a general purpose processor. But before that we will choose an architectural approach to induce such performance enhancements.

3.2 Architecture Definition for Parallel Cryptographic Calculations

In Section 2.3.1, a general presentation of computer architectures has been given, with a focus on vector processors. In this section, the arguments in favour of Data Parallel architectures are given.

3.2.1 A case for general purpose programmable architectures

Secure embedded tokens like smart-cards have to face tight timing constraints for the execution of the cryptographic algorithms they execute. For that purpose, these systems usually embed dedicated cryptographic co-processors that either implement a complete secret key algorithm (like DES or AES) or which performs modular operations on long precision numbers (512-1024 bits) for Public Key Cryptography (mainly for RSA). Some publications might even infer that hardware implementations “*can be secured more easily*” [Kuo & Verbauwhede (2001)]. But with such systems we can usually associate the following inconveniences:

1. Co-processors are usually bulky. For example, in Table 3b of [Naccache & M'Raihi (1996)], we can see that the area of arithmetic co-processors for cryptography is

3.2 Architecture Definition for Parallel Cryptographic Calculations

between 10-20% of the chip's total area.

2. The above point also implies that the power consumed by these co-processors is huge. In smart-card applications like GSM where the current is limited to 6mA and in 3G where we are granted a maximum of 10mA, this implies that either the clock has to be slowed down or that the main processor has to be switched off during the cryptographic calculation (or both in some cases).
3. Such hardware systems are usually difficult to protect against side-channel or fault attacks. Even if countermeasures are added at design time, only tests performed on the final chips can prove the effectiveness of the countermeasures (and it's usually too late to make changes then). In addition to this, updating such hardware countermeasures when new attacks are published usually means re-designing the chip.
4. Most of these co-processors are proprietary designs for which circuit designers and eventually end-users have to pay royalties. Moreover, there is usually no software compatibility from one design to another.
5. Such implementations also provide little flexibility in terms of evolution of the cryptographic algorithms used (like increasing the key length of RSA or ECC).

For all of these reasons, it would be more interesting to integrate crypto-oriented instructions into the instruction set of a general purpose co-processors (GPCs). The first steps have been taken by academia like in [McGregor & Lee (2003)] where bit permutation instructions are proposed. Instruction set extensions for Elliptic Curve Cryptography on *pseudo-Mersenne* curves have been proposed by [Großschädl *et al.* (2004)] but these instructions are not suitable for any curve. A reconfigurable architecture is depicted in [Kumar & Paar (2004)], but is based on a multiplier limited to 163 bits. Dedicated instructions for ECC on any curve for 32-bit architectures have been suggested [Großschädl & Kamendje (2003); Großschädl & Savas (2004)] but the proposed hardware is not scalable. A similar approach is used on the SmartMIPSTM [MIPS (2005)]. Dedicated instructions for Elliptic Curve Cryptography have also been implemented on the Sparc processor [Eberle *et al.* (2005)], instructions which can also be used to accelerate the AES algorithm [Tillich & Großschädl (2005, 2006)]. But, to the best of my knowledge, none has yet embraced the problem by going back to the computer design space and look for the architecture that would integrate the functionality together with the growing need for adaptive security. Such evolutionary moves are not new to the computer industry. For example, some 20 years ago, Floating Point arithmetic was done by peripheral co-processors and today, given the common use of such calculations, the associated functionality has been added to the processor core itself. Examples of these are the way in which processors like ARM, MIPS, x86... have evolved.

I propose to go from an available General Purpose Processor, which is currently largely used in the embedded computer industry, and design new processing units that handle cryptography in the most efficient and flexible way in terms of performance, area and power consumption.

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

3.2.2 A case for vector architectures

As already mentioned, the basic architectural requirements behind the system we are trying to build are those of

- *Scalability*: being able to adapt the algorithm itself, the size of keys or the level of security as a function of the software countermeasures implemented.
- *Performance*: being able to offer high performance.
- *Low Power*: being able to adapt, through “architectural adjustments”, the power consumed by the final design to the targeted application.
- *Security*: in our case, providing the proper hardware configuration to implement efficient countermeasures.

In Chapter 2, I presented a decomposition of the computer architecture design space as illustrated in Figure 3.1.

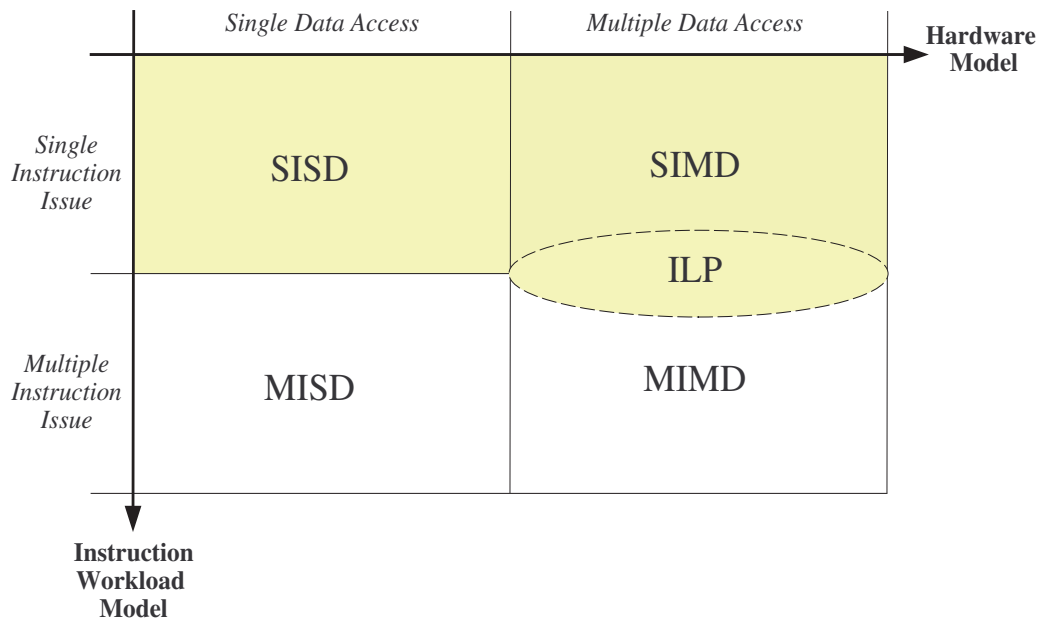


Figure 3.1: (*Flynn's*) Architecture Design Space

Single Instruction Issue processors are chosen in order to maintain compatibility with existing smart-card chips. Having a *Multiple Instruction Issue* processor would imply having a multi-processor system which does not fit with actual power and size constraints on embedded chips. *Instruction Level Parallel* architectures were also put aside because having parallel instruction executions:

- requires complicated instruction decoding and scheduling units, which do not help reducing complexity.

3.2 Architecture Definition for Parallel Cryptographic Calculations

- implies the use of very sophisticated instruction decoders and issuers, which consume a lot of power as illustrated in [Folegnani & González (2001)]. In the latter paper, the authors show that in a superscalar microprocessor, the instruction issue and queue logic accounts for nearly one quarter of the total energy consumed by the processor while another quarter is accounted for by the instructions' reorder buffers.
- is not well suited for these particular applications: most cryptographic algorithms involve the sequential use of precise instructions/operations leaving little room for parallelism at this level.

A *Data Level Parallel* approach was chosen because

- the data used by these cryptographic algorithms can be decomposed into a vector of shorter data onto which operations can be applied in parallel (or *partially-parallel*) as it will be illustrated in Sections 3.3 and 3.4.
- The instruction decoding is simpler, i.e. no dedicated logic is required for dynamic instructions' schedule and reordering.
- The extra logic required for the parallel processing resides mainly in the data path and not in the control path.
- In terms of security, working on data in parallel can in theory reduce the relative contribution of each data piece to the external power consumption as announced in [Brier *et al.* (2003)].

Hence Data Level Parallel techniques are used to design the cryptographic processing unit described in this thesis. The vector machine is controlled by a general purpose processor (GPP) which also allows the optimal execution of 'scalar' codes¹.

3.2.3 Having a MIPS-based approach

The first choice made was relative to the GPP upon which our architecture had to be based. We decided to go for the MIPS4K architecture because:

- The MIPS32TM architecture is simple, open with a consequent amount of documentation available off the web like [MIPS (2001a)], [MIPS (2001b)] and [MIPS (2001c)].
- A derivative of the MIPS4KcTM is already used in the smart-card industry and it makes sense to try to stick to an architecture which is already in use.
- This architecture also provides support for optional Application Specific Extensions (ASE) and an interface for optional co-processor (namely the CP2)
- It has a simple 5-stage pipeline which is easy to understand:
 - **Instruction Fetch (IF):** instruction cache fetch and virtual-to-physical address translation

¹In this paper, a *scalar code* is an algorithm's code implemented on a scalar machine (MIPS-I) and a *vector code* is an algorithm's code implemented on a vector machine.

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

- **Instruction Decode (ID)**: instruction decode, check for inter-lock conditions, instruction virtual address calculation, next instruction calculation and register file read
 - **Execution (EX)**: arithmetic & logic operations, branch compare, data virtual address calculation
 - **Data Cache Read (DC)**: data cache read, load data alignment, data cache tag check.
 - **Write Back (WB)**: data cache write, register file write.
- The MIPS4KscTM (SmartMIPSTM) will be a good way of benchmarking the performance of the proposed architecture. Some independent research work has also been done on adding scalar crypto-oriented instructions like in [Großschädl & Kamendje (2003)] and this would be another way of assessing the quality of the work done in our research.
 - Some simulation tools (like SimpleScalar or ArchC) already propose MIPS-like ISAs for simulation

From there, we make the choice of having a MIPS-based architecture to have a vector approach to implement cryptography. For AES and modular multiplications, relevant vector instructions are identified and they now have to be integrated into a simple MIPS structure. In Sections 3.3 and 3.4, we will see that for the two strategic cases of AES and modular multiplications, the inner cryptographic calculations could be made to work on vectors of data, suggesting that there is room for parallelism within such operations.

3.3 Case Study: AES encryption algorithm

The AES algorithm is described in [NIST (2001b)]. The algorithm can involve keys of 128, 192 or 256 bits but we concentrate on the 128-bit version of the AES as it is very representative of what is happening in the algorithm. Moreover, if we look more closely to the algorithm’s specifications in [NIST (2001b)], the difference between the AES-128, AES-192 and AES-256 lies in the key-schedule where we have longer keys: for the encryption/decryption processes, the data length is always on 128 bits and only the number of rounds changes (respectively 10, 12 and 14). Moreover, we only focus on the encryption process as the decryption process uses almost the same operations but in a different order, except for the INV_MIXCOLUMNS operation which slightly differs from the MIXCOLUMNS used during encryption, but their ‘constituent’ operations are the same.

In this case study, we look at the implementation of the AES-128 on a MIPS processor architecture: implement and simulate the algorithm on a MIPS-I compatible ISA, identify the most time-consuming operations in the algorithm’s structure and from there identify which points are worth upgrading in order to obtain maximum speed-up.

3.3.1 The Advanced Encryption Standard

The structure of the AES-128 is given in Figure 3.2. In our test implementation, the key-schedule is done first and the sub-keys stored in RAM memory: the encryption/decryption

3.3 Case Study: AES encryption algorithm

processes are then done and for each round the corresponding sub-keys were fetched from the RAM memory.

The 128-bit data or key is considered as a matrix of 4×4 bytes. The operations are performed column-wise except for the **SHIFTRROWS** which is defined in rows. Hence, on a MIPS-I 32-bit architecture, we represent our matrix on four 32-bit words where each word is one column of the 4×4 matrix.

Key Schedule. This simple operation is shown in Figure 3.2. The **TRANSFORM** done on *K3* consists of

- Rotating *K3* by 1 byte to the left.
- Then performing a **SUBBYTE** operation of the individual bytes.
- Then **XORing** the left-most byte by x^{i-1} in $\text{GF}(2^8)$ where i is the sub-key's index. The various x^{i-1} s are simply pre-calculated and accessed through a look-up table.

Add-Round-Key. The **ADDRNDKEY** operation is illustrated in Figure 3.2 by the ‘big’ **XOR** symbol. It consists of a byte-wise **XOR** between the data matrix and the corresponding sub-key matrix.

SubByte. The **SUBBYTE** is specified as a look-up table of 256 bytes where to each byte of the data matrix corresponds to a substitution byte in this look-up table.

ShiftRows. The **SHIFTRROWS** operation is a simple leftwise rotation operation on each row of the data matrix: the first row is unchanged; the second row is rotated by 1 byte; the third row by 2 bytes; and the forth row by 3 bytes.

MixColumns. The **MIXCOLUMNS** operation is a matrix multiplication working on each column as defined below:

$$\begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \bullet \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \quad (3.1)$$

Each of the individual byte multiplications is done in the field $\text{GF}(2^8)$, modulo the irreducible polynomial given by

$$m(x) = x^8 + x^4 + x^3 + x + 1 \quad (3.2)$$

whose binary representation is given by the hexadecimal value 11B.

3.3.2 Performance study

The AES-128 is implemented using the MIPS-I instruction set as defined in [MIPS (1996, 2001b)]. No countermeasure is implemented. As already mentioned, the key schedule is executed first, the sub-keys stored in RAM memory and then the encryption/decryption

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

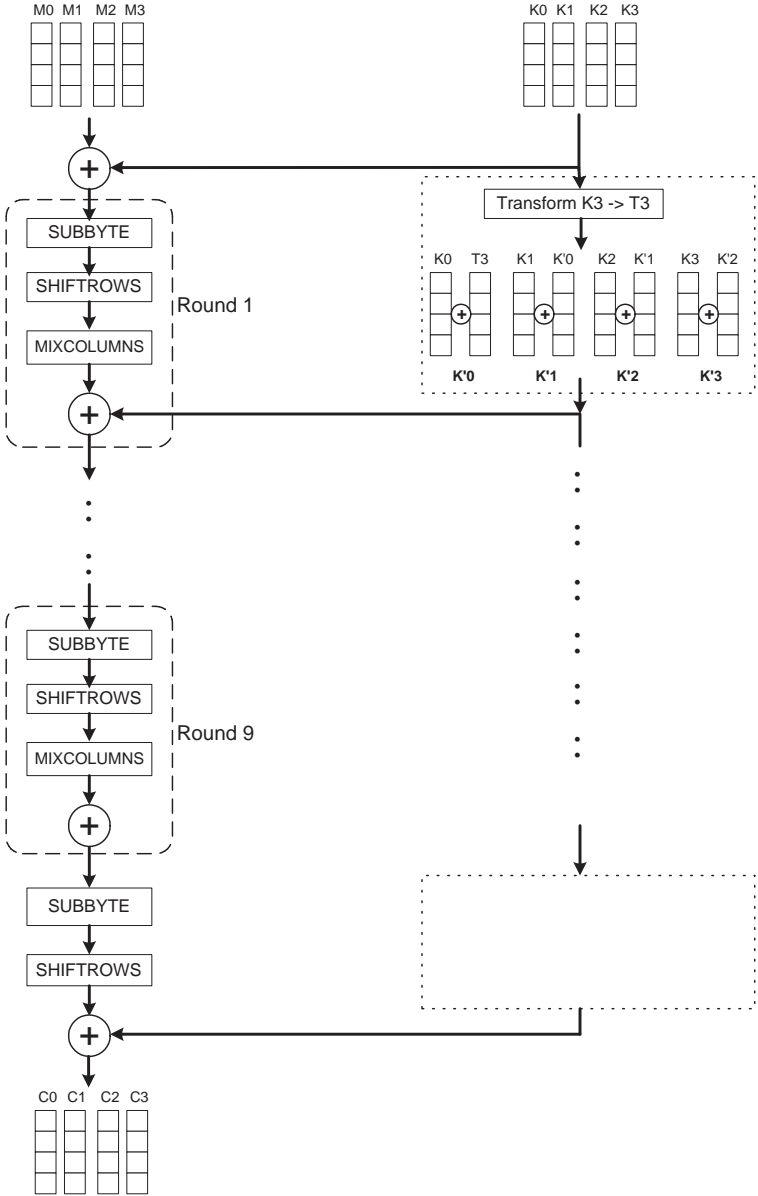


Figure 3.2: AES-128 structure

3.3 Case Study: AES encryption algorithm

processes executed. Table 3.1 gives the number of clock cycles taken by the different processes: the programs' execution times were simulated using the MIPSSimTM simulator tool.

AES-128	Number of clock-cycles
ENCRYPTION	3042
DECRYPTION	5134

Table 3.1: Performance of the AES-128 on MIPS-I

In table 3.2, we detail the time taken by each different process: the most time consuming operations are the SUBBYTE and the MIXCOLUMNS operations.

Sub-Process	clock-cycles	times called	Total	% of total encryption
KEY-SCHEDULE	508	1	508	16
ADDRNDKEY	16	11	176	6
SUBBYTE	68	10	680	22
SHIFTRWS	26	10	260	8.5
MIXCOLUMNS	143	9	1287	42

Table 3.2: Decomposition of the AES-128 encryption

3.3.3 Improvement study

Based on the data in table 3.2, we can look at what kind of instructions (like those given in Appendix A), if added to a general purpose co-processor, could enhance the performance of the encryption algorithm. We study how a vector processor¹ could enhance the execution time. It makes sense to leave the key-schedule operation for the time being because on one side it does not represent the most critical operation in terms of performance and on the other side in some application cases, the AES-128 is used to encipher large tables of data using one key, in which case the key-schedule is done once for all and the encryption process executed several times.

In the following study, we will assume that each vector instruction can be issued at every clock cycle and hence take an effective 1 clock-cycle. This is supported by the way these instructions are executed and pipelined. This assumption provides an upper bound for the improvement that can be achieved.

¹The justification for having a vector approach is given in Section 3.2.

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

Add Round Key

The ADDRNDKEY operation is applied to each column of the data matrix. On each column we do

```
LOAD    R0, 0(adr_key)
LOAD    R1, 0(adr_data)
XOR     R2, R1, R0
STORE   R2, 0(adr_data)
```

where `adr_key` is the register containing the address of the key and `adr_data` is the register containing the address of the message. Since there are four columns this is repeated four times. On a vector architecture, assuming that the vector instructions as described in the Appendix A take 1 clock-cycle¹, the ADDRNDKEY can be implemented in only four instructions for all four columns of the data matrix

```
VLOAD   V0, (adr_key), 4
VLOAD   V1, (adr_data), 4
VXOR    V2, V0, V1
VSTORE  V2, (adr_data), 4
```

The gain in performance is calculated according to the following equation:

$$GAIN = \frac{(Time_s - Time_v) \times occurrence}{Time_{TOTAL}}$$

where $Time_s$ is the time taken by the sub-process on the scalar MIPS architecture, $Time_v$ is the time taken by the sub-process with the proposed vector instructions, $occurrence$ is the number of times the sub-process is called during the entire encryption algorithm and $Time_{TOTAL}$ is the time taken for the entire AES-128 encryption on the scalar MIPS. For the ADDRNDKEY the maximum gain that could be expected is

$$GAIN_{ADDRNDKEY} = \frac{(16 - 4) \times 11}{3042} = 4\%$$

Substitute Byte

The SUBBYTE is a byte-wise look-up process. For this purpose we have a VBYTELD Vx, Ry, m instruction as explained in Appendix A. Such an instruction can be implemented if we suppose we have the memory organization described in Section 4.1. Note that this optimization is also useful for the KEY-SCHEDULE. The expected maximum gain would

¹Architectural and Implementation issues will be considered in subsequent chapters. For the time being, we focus on the functional definition of these vector instructions

then be

$$GAIN_{SUBBYTE} = \frac{(68 - 1) \times 10}{3042} = 22\%$$

Shift Rows

Originally the SHIFTRROWS function is composed of left rotations on each row of the data matrix and if we had represented each row of the data matrix on a 32-bit word, the SHIFTRROWS would have been very simple. But in our implementation, each 32-bit word is one column of the data matrix, hence the difficulty of implementing this operation. Suppose we have the operations VTRANSP and VBCROTR (Vector-Bit-Conditional-Rotate-Right) as described in Appendix A, the SHIFTRROWS operations can be implemented as follows:

```

VLOAD   V0, (adr_data), 4   # loads data into V0
VTRANSP V1, V0, 4           # V1 = V0 transposed
ADDIU   R11, 0x000E
MTVCR   R11                  # VCR = 1110b
VBCROTR V2, V1, 24          # V2 = V1 whose words indexed 1,2,3
                                           # have been rotated right by 24 bits

ADDIU   R11, 0x000C
MTVCR   R11                  # VCR = 1100b
VBCROTR V1, V2, 24          # V1 = V2 whose words indexed 2,3
                                           # have been rotated right by 24 bits

ADDIU   R11, 0x0008
MTVCR   R11                  # VCR = 1000b
VBCROTR V2, V1, 24          # V2 = V1 whose word indexed 3
                                           # is rotated right by 24 bits

VMOVE   V0, V2, 4           # V0 = V2 transposed
VSTORE  (adr_data), V0, 4   # stores result into memory

```

Assuming that each of the above vector instructions takes one clock cycle, the maximum performance gain that could be achieved is

$$GAIN_{SUBBYTE} = \frac{(26 - 10) \times 10}{3042} = 5\%$$

Mix Columns

The MIXCOLUMNS operation is the most time consuming one accounting for 42 % of the AES-128 encryption as shown in Table 3.2. The MIXCOLUMNS operation is a matrix multiplication acting on every column of the data matrix as shown by equation 3.1. Each byte of the column can be viewed as a polynomial in x of degree 7 which is multiplied by a matrix of polynomials in x of degree 7. Hence equation 3.1 can be switched into a

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

polynomial representation and further developed into the following:

$$\begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \begin{pmatrix} x \cdot a + (x + 1) \cdot b + c + d \\ a + x \cdot b + (x + 1) \cdot c + d \\ a + b + x \cdot c + (x + 1) \cdot d \\ (x + 1) \cdot a + b + c + x \cdot d \end{pmatrix} = \begin{pmatrix} x(a \oplus b) \oplus b \oplus c \oplus d \\ x(b \oplus c) \oplus a \oplus c \oplus d \\ x(c \oplus d) \oplus a \oplus b \oplus d \\ x(a \oplus d) \oplus a \oplus b \oplus c \end{pmatrix} \quad (3.3)$$

Equation 3.3 shows that the central operation in the MIXCOLUMNS is the multiplication operation by x modulo $m(x)$ (which is given in equation 3.2), specially since this multiplication is done on the individual bytes within every 32-bit word. For this purpose, we defined the vector instruction VPMUL (Vector-Modulo-Polynomial-Multiplication): this instruction works on every byte within a vector register and multiplies each of them by x in $GF(2^8)$, i.e. modulo an irreducible polynomial of degree 9 stored in one of the processor's scalar registers. Given these instructions, the MIXCOLUMNS operation can be implemented as follows

```

VLOAD    V0, (adr_data), 4    # loads data into V0.
                                     # V0 is made up of bytes (a,b,c,d)

ADDIU    R11, R0, 0xFFFF
MTVCR    R11                  # VCR = 0xFFFF
VBCROTR  V1, V0, 8           # Each word of V1 = (d,a,b,c)
VBCROTR  V2, V0, 16          # Each word of V2 = (c,d,a,b)
VBCROTR  V3, V0, 24          # Each word of V3 = (b,c,d,a)
VXOR     V4, V0, V3          # Each word of V4 = (a+b,b+c,c+d,d+a)
ADDIU    R11, R0, 0x011B
MTVCR    R11                  # VCR = 0x011B
VPMUL    V5, V4, R0          # Each byte of V4 is shifted
                                     # by 1 bit left and XORed with
                                     # last byte of VCR if outgoing
                                     # bit is 1. Mult by 'x' mod 0x011B.
                                     # 1 word of V5 =
                                     # (x(a+b),x(b+c),x(c+d),x(d+a))
VXOR     V0, V5, V1          # 1 word of V0 = (x(a+b)+d,x(b+c)+a,
                                     # x(c+d)+b,x(d+a)+c)
VXOR     V0, V0, V2          # 1 word of V0 =
                                     # (x(a+b)+d+c,x(b+c)+a+d,
                                     # x(c+d)+b+a,x(d+a)+c+b)
VXOR     V0, V0, V3          # 1 word of V0 =
                                     # (x(a+b)+d+c+b,x(b+c)+a+d+c,
                                     # x(c+d)+b+a+d,x(d+a)+c+b+a)

VSTORE   V0, (adr_data), 4

```

From this we can infer that the maximum gain in performance would be

$$GAIN_{MIXCOLUMNS} = \frac{(143 - 12) \times 9}{3042} = 39\%$$

Note that the `VPMUL` instruction in itself is very simple. The Polynomial multiplication by x on a byte comes down to XORing the 8 LSB (Least Significant bits) of the ‘byte-shifted-by-one-position-to-the-left’ with the 8 LSB of the modulus if the MSb (Most Significant bit) of the original byte is 1. We could work on 4 bytes (32 bits) for every clock cycle and pipeline with the next instruction from one 32-bit word to another.

Overall Performance Gain

Based on the vector instructions defined in Appendix A, the maximum overall gain in performance that could be achieved by switching from a scalar implementation to a vector one is

$$GAIN_{TOTAL} = \frac{(143-12) \times 9 + (68-1) \times 10 + (26-10) \times 10 + (16-4) \times 11}{3042} = 70\%$$

Under those same assumptions, we could further increase the gain by factorizing the vector loads and stores and work directly with the intermediate ‘states’ in vector registers.

In the above calculations we suppose that we have to encrypt 128 bits of data. If the vector register’s depth is p (i.e. each vector register is an array of p 32-bit words), we could work on $\frac{p}{4}$ different data matrices inputs at the same time. The time taken for ciphering $\frac{p}{4}$ data matrices on the scalar processor is then given by

$$Time_{\frac{p}{4}} = 508 + (3042 - 508) \times \frac{p}{4} = 508 + 633.5p \quad \text{clock-cycles} \quad (3.4)$$

where we have factorized the key schedule. The gain on four data matrices input can be calculated as

$$GAIN_{\frac{p}{4}} = \frac{\frac{p}{4} \times (16 \times 11 + 68 \times 10 + 26 \times 10 + 143 \times 9) - (4 \times 11 + 1 \times 10 + 10 \times 10 + 12 \times 9)}{508 + 633.5p} = \frac{600.75p - 262}{508 + 633.5p} \quad (3.5)$$

Hence from equation 3.5, we can derive the following gain table as a function of the depth of the vector registers.

3.4 Case Study: Elliptic Curve Cryptography

The second case study is based on the critical modular multiplication used for Public Key Cryptography. There are two main schemes for implementing Public Key Cryptography: one based on integer modular multiplications as used in RSA and the other based on polynomial arithmetics in finite fields over a given elliptic curve as used in ECC. The

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

Depth of vector register p	GAIN
8	81%
16	87%
32	91%

Table 3.3: Gain in performance for varying vector register depths

main reason for choosing ECC against RSA is that for the past decade, Elliptic Curve Cryptography has been promoted as an attractive alternative to RSA as a scheme for Public Key Cryptography especially for the embedded world. RSA is based on modular exponentiation algorithms involving large values of 512-2048 bits (or even higher if need be). ECC relies on the scalar multiplications of points found on an elliptic curve such that the size of the data (of the order of 200 bits) involved is much smaller than those used in RSA. For example the level of security attained with a 1024-bit RSA is the same as that obtained doing an ECC on 163 bits. This means that the computation time for ECC-based PKC is very small when compared to that for RSA-based PKC as both use the same modular arithmetic algorithms.

We also have to decide on which kind of ECC implementation to choose. Elliptic Curves can be implemented on different fields, with different representations and different bases. In the next sections, we give a brief overview of the theory behind ECC (more details can be gathered from [Blake *et al.* (1999)]) before giving the motivations behind the choices made to code our test program.

3.4.1 A brief overview of ECC theory

In Elliptic Curve Cryptography, calculations are done on an elliptic curve over a field \mathbb{K} given by the general equation

$$y^2 + a_0xy + a_1y = x^3 + a_2x^2 + a_3x + a_4 \quad \text{with } a_i \in \mathbb{K} \quad (3.6)$$

The field in question can either be a prime finite field or a binary finite field.

Prime Finite Field: The elliptic curve is defined over \mathbb{F}_p where p is prime. Operations are done *mod* p over the curve defined by

$$y^2 = x^3 + ax + b \quad \text{with } \{a, b\} \in \mathbb{F}_p \quad \text{for } p > 3 \quad (3.7)$$

Binary Finite Field: The elliptic curve is defined over \mathbb{F}_{2^m} where m is a positive integer. The curve's equation is given by

$$y^2 + xy = x^3 + ax^2 + b \quad \text{with } b \neq 0 \text{ and } (a, b) \in \mathbb{F}_{2^m} \quad (3.8)$$

As \mathbb{F}_{2^m} is isomorphic to $\mathbb{F}_2[x]/f(x)$, all calculations in a binary field representation are

3.4 Case Study: Elliptic Curve Cryptography

done modulo the irreducible polynomial $f(x)$ of degree m and with coefficients in \mathbb{F}_2 . Each point on the binary field elliptic curve can have two possible representations:

- *Polynomial Basis Representation* where the coordinates of each point are represented over $\{x^{m-1}, x^{m-2}, \dots, x^2, x, 1\}$
- *Normal Basis Representation* where the coordinates of each point are represented over $\{\theta^{2^{m-1}}, \theta^{2^{m-2}}, \dots, \theta^2, \theta, \theta\}$ where θ is an element of \mathbb{F}_{2^m}

Choosing the binary field representation

In our study, we focus on elliptic curves in binary finite fields. We use a polynomial representation which is simpler to map to machine words. Nevertheless, working with ECC in \mathbb{F}_{2^m} is the slowest case when working on General Purpose Processors. As detailed in the following sections, the basic operations involved are polynomial multiplications which are rare operations on processors like the MIPS as opposed to integer multiplications used for ECC in \mathbb{F}_p . Thus

1. By looking at binary fields, we deal with the slowest case, at least on General Purpose Processors. In a more general sense, working on \mathbb{F}_{2^m} is slower than working in \mathbb{F}_p for an equivalent level of security.
2. Once we have defined schemes, architectures and instructions to optimize ECC in \mathbb{F}_{2^m} , it is easy to transpose these principles to work on \mathbb{F}_p since in the latter we only need to propagate the carry for multiplication operations.
3. Once we are working in \mathbb{F}_p , we also have schemes to work on RSA-based cryptography as the same basic mathematical principles apply except that we would be working on larger sizes of data.

The above transpositions are made possible because for the basic modular multiplication, we use Montgomery's algorithm whose structure is exactly the same whether we are doing a RSA on 1024 bits or an ECC in \mathbb{F}_{2^m} or an ECC in \mathbb{F}_p .

3.4.2 Scalar multiplication over \mathbb{F}_{2^m}

We focus on fields of characteristic 2 for which the general equation for an elliptic curve is given by (3.8). Each coordinate of a point on the curve defined by (3.8) is represented by a polynomial of degree $m - 1$: the corresponding m binary coefficients hence constitute an m -bit word array. All operations are performed modulo an irreducible polynomial $f(x)$. For further details please refer to [Blake *et al.* (1999)].

When doing ECC, the critical operation is the multiplication between a scalar and a point on the curve. For example in an ECDSA scheme [ANSI (1997)], the signature is generated using the x coordinate of the point resulting from such a scalar multiplication operation. The easiest way of doing this scalar multiplication operation is to do the straight forward double and add algorithm.

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

```

Input  : Point  $P$ , scalar  $k = (k_{m-1}k_{m-2}\dots k_0)_2$ 
Output :  $Q = kP$  in  $\mathbb{F}_{2^m}$ 

```

```

Suppose  $k_{m-1} = 1$ 
   $Q \leftarrow P$ 
  for  $i = m - 2$  downto  $0$  do
     $Q \leftarrow 2Q$ 
    if  $(k_i = 1)$  then  $Q \leftarrow Q + P$ 
  endfor
return  $Q$ 

```

Figure 3.3: Double and Add algorithm

From Figure 3.3, if we suppose that on average k can contain an evenly distributed number of zeros and ones, we can deduce that one scalar multiplication over \mathbb{F}_{2^m} requires $(m - 1)$ point doublings and $\frac{m-1}{2}$ point additions. There are certainly other ways of doing such multiplications, especially if we want to have an algorithm which is resistant to power attacks. An example of a secure implementation is given in [Joye & Tymen (2001)] where Joye and Tymen propose methods to “randomize” P to hide information leakage due to the signature of the base point P . In addition to the latter method one could ensure constant timing by implementing a *Double and Add always* by having a fake addition for the case when $k_i = 0$. However, the point here is only to show that the basic operations involved are the point addition and the point doubling and our aim is to accelerate those two critical operations. As we will see in the next sections, the way of representing these points will directly influence the performances of the point addition and doubling.

Moreover, for representing a point on an Elliptic Curve, one can either use its affine coordinates in (x, y) or one can use its projective coordinates in (X, Y, Z) . In the following sections, we investigate these two alternatives in terms of number of basic operations involved before actually looking closely at those basic operations and evaluate their costs in terms of performance.

Point addition and doubling in \mathbb{F}_{2^m} using affine coordinates

Operations in \mathbb{F}_{2^m} are performed modulo an irreducible $f(x)$ of degree m . Since we use a polynomial representation, adding two points $P(P_x, P_y)$ and $Q(Q_x, Q_y)$ from the curve defined by equation (3.8) yields a point R whose affine coordinates R_x and R_y are given by the following equations:

$$\begin{aligned}
 R_x &= \lambda^2 + \lambda + P_x + Q_x + a \\
 R_y &= \lambda(P_x + R_x) + R_x + P_y \\
 &\text{with } \lambda = \frac{P_y + Q_y}{P_x + Q_x}
 \end{aligned} \tag{3.9}$$

Note that with such a representation additions mod $f(x)$ are simple bitwise XOR opera-

3.4 Case Study: Elliptic Curve Cryptography

tions. Moreover, we see that this point addition operation requires 2 modular multiplications, 1 modular square, 9 modular additions and 1 inversion. Likewise the coordinates of R obtained by doubling a point $P(P_x, P_y)$ are calculated using the following equations:

$$\begin{aligned} R_x &= \lambda^2 + \lambda + a \\ R_y &= (\lambda + 1)R_x + P_x^2 \\ \text{with } \lambda &= P_x + \frac{P_y}{P_x} \end{aligned} \tag{3.10}$$

In this case we see that we need 2 modular multiplications, 2 modular squares, 5 modular additions and 1 modular inverse.

Point addition and doubling in \mathbb{F}_{2^m} using projective coordinates

Instead of representing a point P with its x and y coordinates P_x and P_y , P is represented using three coordinates P_X , P_Y and P_Z as explained in [Blake *et al.* (1999)] or [Lopez & Dahab (1999)].

Notations: From now on, we will use small caps x and y to denote the x and y coordinates for an affine representation and large caps X , Y and Z to denote the x, y and z coordinates for a projective representation.

In [Hankerson *et al.* (2000)], the authors provide a comprehensive study of the performance figures associated with several possible projective coordinates. The one which seems to be the most efficient is the one using the *Jacobian* representation. The Jacobian projective coordinates do offer very interesting performance figures. In such a representation, we have the following conversion equations $P_x = \frac{P_X}{P_Z^2}$ and $P_y = \frac{P_Y}{P_Z^3}$. Using these conversion equations, the general binary elliptic curve equation 3.8 becomes:

$$Y^2 + XYZ + = X^3 + aX^2Z^2 + bZ^6 \tag{3.11}$$

Putting those same conversion equations into equations (3.9) and (3.10), we can show that using the Jacobian projective coordinates a point addition can be done into 15 multiplications, 3 squares and 8 additions as illustrated in Figure (3.4).

Moreover, a point doubling using Jacobian coordinates would require 5 multiplications, 5 squares and 4 additions as seen in Figure (3.5).

To sum up, we gather the following Table 3.4 regarding the different the number of operations needed for the a point addition and a point doubling. These operations are all operations done on data of m -bits modulo an irreducible polynomial $f(x)$.

The point to remember is that calculating a modular inverse is extremely expensive: an inverse can be 35-40 times longer than a modular multiplication for example. This is why

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

Calculate: $R(R_X, R_Y, R_Z) = P(P_X, P_Y, P_Z) + Q(Q_X, Q_Y, Q_Z)$

$$\begin{aligned} \lambda_4 &= P_Z^2 \\ \lambda_5 &= Q_Z^2 \\ \lambda_0 &= P_X Q_Z^2 + Q_X P_Z^2 \\ \lambda_1 &= P_Y Q_Z^3 + Q_Y P_Z^3 \\ \lambda_2 &= Q_Z \lambda_0 \\ R_Z &= P_Z \lambda_2 \\ \lambda_3 &= \lambda_1 + R_Z \\ R_X &= \lambda_1 \lambda_3 + a R_Z + \lambda_0^3 \\ R_Y &= R_X \lambda_3 + \lambda_3^2 [\lambda_1 P_X + \lambda_0 P_Y] \end{aligned}$$

Figure 3.4: Point addition in Jacobian coordinates

Calculate: $R(R_X, R_Y, R_Z) = 2P(P_X, P_Y, P_Z)$

$$\begin{aligned} \lambda_0 &= P_Z^2 \\ \lambda_1 &= P_X^2 \\ \lambda_2 &= \lambda_1^2 \\ R_Z &= P_X \lambda_0 \\ R_X &= \lambda_2 + b \lambda_0^4 \\ R_Y &= R_Z \lambda_2 + R_X [\lambda_1 + P_Y P_Z + R_Z] \end{aligned}$$

Figure 3.5: Point doubling in Jacobian coordinates

projective coordinates are usually preferred since the point additions and doublings do not require any modular inverse operation. In our study, we can hence focus on the modular multiplication operation (a modular square can be viewed as a modular multiplication where both operands are the same) for our analysis. For modular additions these are simple XOR operations.

Modular multiplications in \mathbb{F}_{2^m}

Modular multiplications have been thoroughly studied and optimized methods like those proposed in [Koç & Acar (1998)] based on Montgomery’s method are quite rapid algorithms, especially since we are concerned with rather short words of only a few hundred bits. The same practical methods can be applied to modular squares. As already mentioned, the structure of this algorithm, as it will be depicted for modular multiplication in \mathbb{F}_{2^m} , also applies for modular multiplications in \mathbb{F}_p and by extension to the RSA algorithm.

A modular multiplication process can be divided into two phases, namely the multiplication phase followed by the reduction phase. The scheme proposed in Montgomery’s method [Montgomery (1985)] basically accelerates the reduction phase. However, in order to improve performance and reduce storage space, the multiplication and reduction phases can be interleaved into what we will refer to as *Montgomery’s Algorithm*. As explained in [Koç & Acar (1998)], Montgomery’s algorithm can be adapted to work on any data

3.4 Case Study: Elliptic Curve Cryptography

Modular Operation	Addition	Inverse	Multiplication	Square
Affine Point Addition	8	1	2	1
Affine Point Doubling	6	1	2	2
Projective Point Addition	8	0	15	3
Projective Point Doubling	4	0	5	5

Table 3.4: Number of operations for ECC point additions and doublings

word size, in particular on data which is decomposed into 32-bit words as we expect them to be on a typical MIPS-I architecture. In the latter paper, the authors show that if we want to multiply two polynomials $a(x)$ and $b(x)$ modulo an irreducible polynomial $f(x)$, we can use Montgomery's algorithm bearing in mind that the latter will give the result $c(x)$ such that $c(x) = a(x) \cdot b(x) \cdot r(x)^{-1} \bmod f(x)$. Given that we are working in the field \mathbb{F}_{2^m} , the polynomials involved in this algorithm are of length m , the authors in [Koç & Acar (1998)] show that $r(x)$ can be chosen such that:

$$r(x) = x^k \text{ where } k = 32M \text{ and } M = \left\lceil \frac{m}{32} \right\rceil \quad (3.12)$$

If we suppose that the multiplicand $a(x)$ can be decomposed into a linear combination of 32-bit polynomials denoted by $A_i(x)$ such that

$$a(x) = A_{M-1}(x) \cdot x^{32(M-1)} + A_{M-2}(x) \cdot x^{32(M-2)} + \dots + A_1(x) \cdot x^{32} + A_0(x) \quad (3.13)$$

we then have the following algorithm for the implementation of Montgomery's Modular Multiplication on our 32-bit MIPS-I architecture:

```

Input  :  $a(x), b(x), f(x), M$  and  $N_0(x)$ 
Output :  $c(x) = a(x) \cdot b(x) \cdot x^{-32M} \bmod f(x)$ 


---


 $c(x) \leftarrow 0$ 
for  $j = 0$  to  $M - 1$  do
   $c(x) \leftarrow c(x) + A_j(x) \cdot b(x)$ 
   $M(x) \leftarrow C_0(x) \cdot N_0(x) \bmod x^{32}$ 
   $c(x) \leftarrow c(x) + M(x) \cdot f(x)$ 
   $c(x) \leftarrow c(x) / x^{32}$ 
endfor
return  $c(x)$ 


---



```

Figure 3.6: Montgomery Modular Multiplication on a 32-bit architecture

where $C_0(x)$ is the least significant 32-bit word of the polynomial $c(x)$ and $N_0(x)$ is the 'Montgomery's constant', which is pre-calculated, such that $N_0(x) \cdot F_0(x) \bmod x^{32} = 1$. As

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

we can see, $N_0(x)$ depends only on the modulus and can be pre-calculated at the beginning of the Double&Add algorithm. Moreover, if we work with projective coordinates, we get rid of the ‘inversion’ problem since we don’t need any (except if we need to change the points’ representation at the beginning of the Double&Add algorithm from affine to projective, but then only one inversion is needed). Hence the operation that really impacts the performance of the points’ multiplication and doubling (and hence the Double&Add algorithm) is the modular multiplication as detailed in Figure 3.6. In the next sections, we will study the performance of the latter scheme and identify the bottlenecks and/or the critical operations, hereby allowing us to suggest ways of improving the modular multiplication algorithm.

3.4.3 Performance study

In this section, we study the implementation of a modular multiplication in \mathbb{F}_{2^m} based on Montgomery’s algorithm as given in Figure 3.6. The target processor runs a MIPS-I ISA (Instruction Set Architecture). If we suppose that each polynomial on which we work can be decomposed into arrays of 32-bit words such that:

$$\begin{aligned}
 a(x) &= A_{M-1}(x).x^{32(M-1)} + A_{M-2}(x).x^{32(M-2)} + \dots + A_1(x).x^{32} + A_0(x) \\
 b(x) &= B_{M-1}(x).x^{32(M-1)} + B_{M-2}(x).x^{32(M-2)} + \dots + B_1(x).x^{32} + B_0(x) \\
 f(x) &= F_{M-1}(x).x^{32(M-1)} + F_{M-2}(x).x^{32(M-2)} + \dots + F_1(x).x^{32} + F_0(x) \\
 c(x) &= C_{M-1}(x).x^{32(M-1)} + C_{M-2}(x).x^{32(M-2)} + \dots + C_1(x).x^{32} + C_0(x)
 \end{aligned} \tag{3.14}$$

as a result of which each polynomial can be represented by arrays of 32-bit words:

$$\begin{aligned}
 a &= [A_{M-1}, A_{M-2}, \dots, A_1, A_0] \\
 b &= [B_{M-1}, B_{M-2}, \dots, B_1, B_0] \\
 f &= [F_{M-1}, F_{M-2}, \dots, F_1, F_0] \\
 c &= [C_{M-1}, C_{M-2}, \dots, C_1, C_0]
 \end{aligned} \tag{3.15}$$

In our implementation, c will be temporarily stored on $M + 1$ words. The algorithm implemented has the following structure:

The ‘pseudo instruction’ **PolyMultAdd** performs the polynomial multiplication (i.e. a multiplication where from one bit to another the carry is not propagated and where the addition is replaced by an XOR) between the two registers, putting the most significant 32 bits of the result into the scalar register $t7$ and XORing the least significant 32 bits of the result with the contents of $t6$ and writing the result into $t6$. The division by x^{32} (i.e. here it is a shift by 32 bits) from Figure 3.6 is taken into account in line 05 of Figure 3.7. Implementing the **PolyMultAdd** routine on such a processor (where we only have arithmetic multiplications, i.e. with carry propagations) is not so straightforward. The latter alone takes about 412 clock-cycles when implemented with the MIPS-I ISA. As a result of that, the entire modular multiplication takes about 22300 clock-cycles.

```

01. for  $i = 0$  to  $M - 1$  do
02.    $(t7, t6) \leftarrow (0, 0)$ 
03.   for  $j = 0$  to  $M - 1$  do
04.      $(t7, t6) \leftarrow \text{PolyMultAdd}(A_i, B_j)$ 
05.      $C_j \leftarrow C_{j+1} \oplus t6$ 
06.      $t6 \leftarrow t7$ 
07.   endfor
08.    $C_M \leftarrow t7$ 
09.    $(t7, t6) \leftarrow (0, 0)$ 
10.    $(t7, t6) \leftarrow \text{PolyMultAdd}(N_0, C_0)$ 
11.    $t1 \leftarrow t6$ 
12.    $(t7, t6) \leftarrow (0, 0)$ 
13.   for  $j = 0$  to  $M - 1$  do
14.      $(t7, t6) \leftarrow \text{PolyMultAdd}(t1, F_j)$ 
15.      $C_j \leftarrow C_j \oplus t6$ 
16.      $t6 \leftarrow t7$ 
17.   endfor
18.    $C_M \leftarrow C_M \oplus t7$ 
19. endfor
20. return  $c = [C_M, C_{M-1}, \dots, C_2, C_1]$ 

```

Figure 3.7: Modular Multiplication Implementation

In this test program, we used test values from the field $\mathbb{F}_{2^{191}}$ with a modulus $f(x) = x^{191} + x^9 + 1$. Hence each of the polynomials were represented by an array of 6 32-bit words. As a result, for some of these values, we could store them directly into registers, thus sparing additional memory accesses.

There are several ways of further improving the implementation of modular multiplications over binary fields, some of which are explained in [Blake *et al.* (1999)]. Most of those techniques however make a lot of assumptions about the modulus used (like knowing the exact locations of the non-zero elements of the irreducible polynomial). But the algorithm as depicted in Figure 3.7 works for any $f(x)$. Moreover it can also be easily transposed to the case where we work in \mathbb{F}_p , and by extension for the RSA algorithm, as we will briefly see in the next paragraph.

Modular multiplication in \mathbb{F}_p

The same Modular Multiplication implementation in \mathbb{F}_p takes far fewer clock-cycles because here we are concerned with arithmetic multiplications: we can use the MULTU operation between two registers, an instruction which is already available on MIPS-I architectures. The difficulty in this case comes from the fact the MIPS-I architecture does not handle the carry bit. Unfortunately, this has to be emulated every time an addition has to be executed. The **PolyMultAdd** routine at lines 04, 10 and 14 of Figure 3.7 have to be replaced by an **ArithMultAdd** where the XORs are replaced by ADD operations and the addition of the carries on registers $t6$ and $t7$ with the addition of a third register

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

t_9 to store an eventual carry bit. Manipulating a carry bit costs 10 clock-cycles per addition. As a result of which, the **ArithMultAdd** takes 40 clock-cycles. Hence the entire Modular Multiplication in \mathbb{F}_p takes about 3200 clock-cycles (which is much less than the 22K clock-cycles for the same operation in $\mathbb{F}_{2^{191}}$ for the same data lengths).

3.4.4 Improvement Study

Like for the AES case study, we look at vector instructions that can help to enhance the execution of this ‘interleaved’ Montgomery Modular Multiplication. As a result of which, the code for Modular Multiplication in $\mathbb{F}_{2^{191}}$ would be:

```
.global MultBinPoly
.ent    MultBinPoly
MultBinPoly:
    lw      $24, 16($29)      # loading value of M (size of data)
    lw      $2, 20($29)      # loading the value of N0
    vload   $v0, $5, 6       # v0 <= b(x)
    vload   $v1, $6, 6       # v1 <= f(x)
    vsmove  $v3, $0, 8       # v3 cleared; (v3 == c(x))
    addiu   $15, $0, 0       # 'j' init. for main loop
    sll     $24, $24, 2      # $24 <= 4M

LoopBin:
    add     $8, $15, $4      # $8 <= j-th word of a(x)
    lw      $8, 0($8)       # loading j-th word of a(x)
    vspmult $v5, $v0, $8     # v5 <= a[j] * v0; (v0 == b(x))
    vxor    $v3, $v5, $v3    # v3 <= v5 + v3
    vextract $9, $v3, 1     # $9 <= C_0
    vsmove  $v2, $9, 1      # v2[0] <= $9 ($9 == C_0)
    vspmult $v4, $v2, $2     # v4 <= N0 * v2
    vextract $9, $v4, 1     # $9 <= M(x)
    vspmult $v5, $v1, $9     # v5 <= v4[0] * v1; (v1 == f(x))
    vxor    $v3, $v3, $v5    # v3 <= v3 + v5
    vwshr   $v3, $v3, 1     # v3 <= v3 shifted right by 1 word
    addi    $15, $15, 4     # Increase index by 4 bytes
    bne     $15, $24, LoopBin
    nop
    vstore  $7, $v3, 5      # writes result in memory
    j       $31             # return from sub-routine
    nop
.end      MultBinPoly
```

For the time being, we will not look at how these vector instructions can be pipelined and cascaded for optimum performance. The above code is also valid for Modular Multiplications in \mathbb{F}_p if the **VSPMULT** instruction is replaced by **VSAMULT** and the **VXOR** operation by

VADDU. And by extension, once we are in \mathbb{F}_p , we can generalize the code for any algorithm based on arithmetic modular operations like in RSA, given that instead of working on 6 words of the vector registers, we would involve 32 of them for a RSA-1024.

Performance: Even to forecast the best case scenario, it would not be a realistic assumption to suppose that the arithmetic vector instructions used in this chapter are capable taking one clock cycle each. This is because most of these instructions do not strictly work on independent sets of data within each vector. This issue is further discussed in Chapter 4 where we not only figure out how those instructions function but also the number of clock-cycles they effectively take and whether they can be pipelined. But in case each instruction can effectively be issued every clock cycle, the routine would take 90 clock cycles.

3.5 Summary

In this section, we have browsed through the basic requirements for hardware accelerators for embedded cryptography, details of which had been given in Chapter 2. We then explained why we chose to have a vector approach. We then illustrated how algorithms like AES and modular multiplications as used in RSA and ECC could be implemented by having a vector representation of the data. We defined vector instructions and we evaluated the kind of performance gains that could be expected for each of these algorithms compared to a purely scalar approach on a MIPS-I architecture. The point of comparing the vector approach with the scalar one is to compare two approaches which inherently offer the same degree of software flexibility in terms of functional and secure implementations. For an AES-128 encryption, the performance would drop from 3042 clock-cycles to 912 clock-cycles. For modular multiplication in $GF(2^m)$ on 192 bits would drop from 22300 clock-cycles on the MIPS-I to 90 clock-cycles with a vector co-processor.

3. VECTORIZING CRYPTOGRAPHIC ALGORITHMS

Chapter 4

VeMICry Architecture & Functional Simulation

The ‘long’ data involved in calculations of algorithms like AES or of the modular multiplications for RSA and ECC can be decomposed into vectors of smaller data on which calculations can be done in parallel. In order to have such an approach while satisfying the requirements of scalability, flexibility and high performance, a hardware architecture based on vector processing is favoured as explained in Section 3.2. In order to also satisfy high performance for scalar code, a scalar General Purpose Processor is also used. Hence the VeMICry (**V**ector **M**IPS for **C**ryptography) is defined: a simple MIPS-like scalar processor and a general purpose vector co-processor that can execute the cryptographic vector instructions and which has the attractive feature of being scalable.

In this chapter, we define the architecture of the vector part of VeMICry (the scalar part being a MIPS-like architecture), notably how the vector instructions defined in Appendix A can be pipelined. We then present the functional simulation model built for the VeMICry. For each of the simulated cryptographic algorithms, we perform performance studies on the scalar MIPS-I architecture to which we compare the performances obtained on this first model of the VeMICry. Performance figures are given in terms of instructions cycles¹. We also show how a *quantitative* analysis can be performed to have a first order approximation of the impact on performance implications of varying the parameters of our design. With this functional simulator we demonstrate that the vector codes defined for AES and modular multiplication do work and we have a first idea about the performances to be expected from such a scalable design.

4.1 The VeMICry Architecture

4.1.1 Architecture specification

The VeMICry architecture is composed of a scalar MIPS-like General Purpose Processor and of a vector co-processor. The core of the vector co-processor depends on how the register files are organized and how the processing units interface with these vector reg-

¹i.e. the number of instructions executed

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

isters. The architecture of the register file, as shown in Figure 4.1, is determined by the following factors:

- m : The size of each element of the vector elements. Currently $m = 32$.
- q : The number of such vector registers.
- p : The number of elements in each vector register. This will be called the *depth* of each vector.
- r : The number of lanes into which the vector registers are organized. The concept of “stride” is explained in [Flynn (1995)]. It is associated to the number of VPUs (Vector Processing Units) available to the VeMICry. We have as many lanes as there are VPUs. Ideally we would have $r = p$ allowing us to work on the p elements in parallel: the j^{th} VPU for example would be ‘associated’ with a register file made of all the j^{th} elements of all the vector registers. However, in some cases, size and power constraints mean that we cannot have p VPUs. We leave r as a parameter for our analysis as to what would be the best performance to size trade-off. As a result, the j^{th} VPU will be associated not only to the j^{th} elements across the register file but also $j + r^{\text{th}}$, $j + 2r^{\text{th}}$... as illustrated in Figure 4.1.

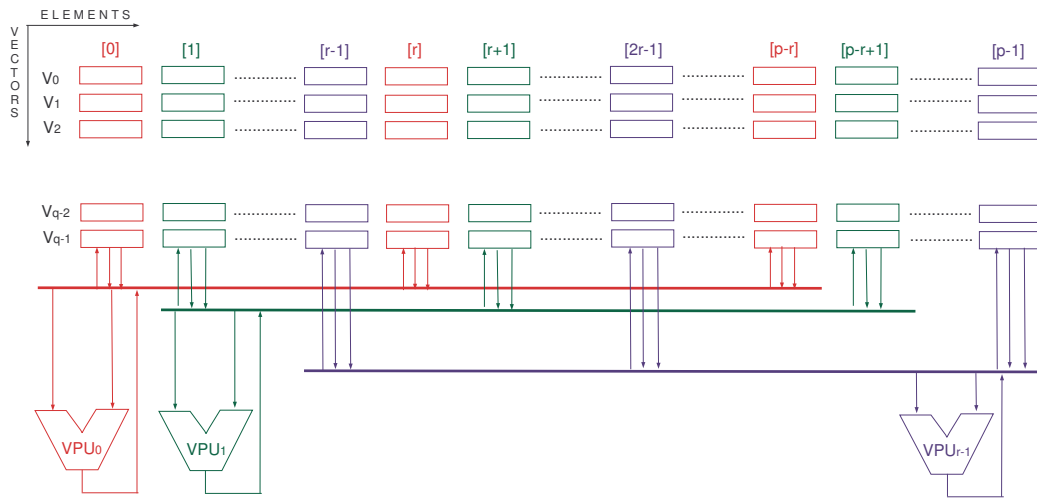


Figure 4.1: Vector Register File

Note that in this definition, each operation is applied to all p elements of each vector register. In the initial definition of the vector co-processor, we thought about the possibility of defining another parameter l which would be the number of elements onto which each instruction is applied (with $l \leq p$). But in the current implementation of the VeMICry such functionality was not mandatory and given the complexity of the corresponding control unit, we decided to leave it as such for the time being.

For the vector processor memory interface we chose a simple scheme. We suppose that the memory is a one-cycle access fast memory (RAM) that could, on a final chip, be cache

memory. Eventually, we could have a software managed memory bank *per* lane. Within each ‘bank’ we have 4 parallel concurrently accessible byte arrays of say 1 kilobytes each. Such a structure allows each VPU to smartly fetch four bytes in parallel for the VBYTELD instruction.

4.1.2 Vector instructions set architecture

Among the vector instructions defined in Appendix A, we can distinguish among three classes:

Definition 4.1 (GIVI) *A Genuinely Independent Vector Instruction is one where the transformation applied to every element of the operand vectors is independent from the application of that same transformation to neighbouring elements.*

Definition 4.2 (PIVI) *A Partially Independent Vector Instruction is one where the transformation applied to every element of the operand vectors depends partially on the result of the same operation applied to one of its neighbors.*

Definition 4.3 (MAVI) *A Memory Accessing Vector Instruction is a vector register-memory instruction where a memory access is required for the application of the required transformation on every element of the operand vectors.*

The instruction decoding is handled by the scalar MIPS as part of its conventional five stage pipeline.

- **IF:** Instruction Fetch.
- **ID:** Instruction Decode.
- **EX:** (Scalar) Execution Stage.
- **MEM:** Memory access stage.
- **WB:** Write Back stage.

Upon the detection of a vector instruction, each VPU enters into its own four stage pipeline defined as follows:

- **Data Fetch (VDF)** stage where each VPU fetches the two (depending on the instruction) elements from the target vector registers. If a scalar register is involved, the value is fetched from the latter scalar register and written back into a register of the VPU called SBI (Scalar Buffer Interface).
- **Execute-Multiply (VEXM)** stage where the VPU performs the corresponding multiplication or addition calculation for a PIVI. For a GIVI or a MAVI, nothing is done.

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

- **Execute-Carry (VEXC)** stage where the ‘carry’ selection is done for the PIVIs and the latter’s calculation is completed. For a GIVI or a MAVI, the corresponding calculation/manipulation is done onto the arguments fetched in stage VDF.
- **Write Back (VWB)** stage where the result from the VPU is written back to the corresponding element of the destination vector register.

Let’s study how each of the classes of vector instructions can be decomposed into each of the above vector pipeline stages.

GIVI execution

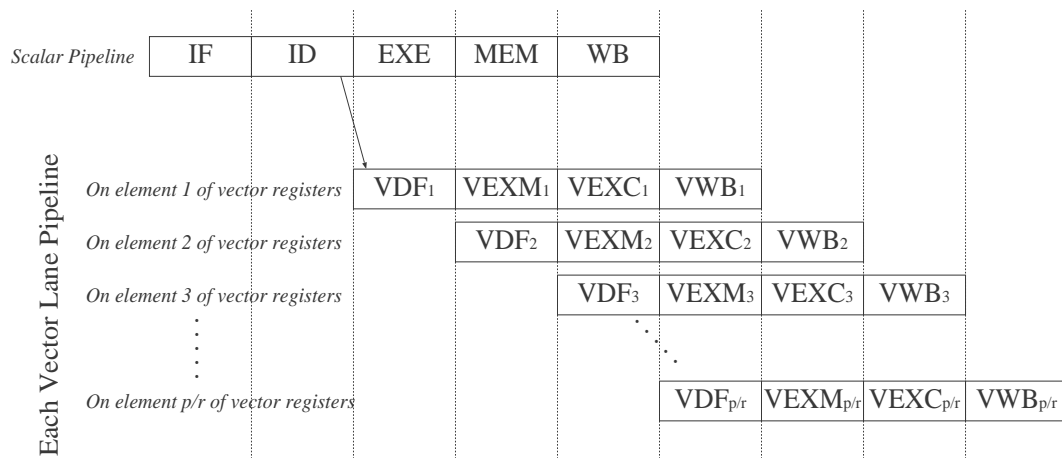


Figure 4.2: Timing relationship between scalar & vector executions

Let’s consider the general case where p is ‘too’ large and that we only have r VPUs where $r \leq p$ (could be specially true for embedded processors). This means each VPU will have to perform $\lceil \frac{p}{r} \rceil$ times the same operation in order to apply the required transformation on all p elements of the targeted vector registers as shown in Figure 4.2¹. Hence the next vector instruction will only be issued $\lceil \frac{p}{r} \rceil$ cycles later.

In Figure 4.2, at each i^{th} stage (comprised of VDF_i , $VEXM_i$, $VEXC_i$ and VWB_i), within each vector register, elements ranging between ir and $(i + 1)r - 1$ are processed. Note that the $i + 1^{st}$ stage can be started at the very next clock cycle because we assume we have two read ports and one write port per VPU or lane (in our simple case each lane has only one VPU). If we refer back to the instructions defined in Appendix A, we can build the following table (Note we don’t show the VEXM stage because nothing is done during that stage for a GIVI).

¹In the Figure 4.2, VDF_i means that it is the first element of the vector register that is found in the VDF stage. The same nomenclature applies to the other vector pipeline stages illustrated in this figure.

GIVI Instructions	VDF	VEXC	VWB	Issue Rate
VBCROTR V_2, V_1, n	Reads r elements from V_1 . Extracts corresponding r bits from VCR	For each element, performs right rotation by n bits if corresponding VCR bit = 1	Writes resulting r words to corresponding element positions in V_2	$\lfloor \frac{p}{r} \rfloor$ cycles
VEEXTRACT R_2, V_1, n	Reads the n^{th} element of V_1	Copies the read value to the SBI register	Transfer of result from the SBI to R_2	1 cycle
VMPMUL V_2, V_1, R_0	Reads r elements from V_1 and value from R_0 to SBI	Executes the byte-wise mod multiplication on each element	Writes resulting r words to corresponding element positions in V_2	$\lfloor \frac{p}{r} \rfloor$ cycles
VSADDU V_2, V_1, R_0	Reads r elements from V_1 and value from R_0 to SBI	Does the unsigned addition of SBI to each element with carry stored	Writes resulting r words to corresponding elements in V_2 and carry bits to CAR	$\lfloor \frac{p}{r} \rfloor$ cycles
VSMOVE V_2, R_1, n	Reads n and copies value from R_0 to SBI	Calculates the vector indices for which $n < r$ and transfers SBI to outputs of VPUs	Writes resulting r words to corresponding element positions in V_2	$\lfloor \frac{p}{r} \rfloor$ cycles

Table 4.1: Genuinely Independent Vector Instructions

GIVI Instructions	VDF	VEXC	VWB	Issue Rate
VXOR V_2, V_1, V_0	Reads r elements from V_1 and from V_0	Executes the word-wise XOR	Writes resulting r words to corresponding element positions in V_2	$\lceil \frac{p}{r} \rceil$ cycles
VWSHL V_2, V_1	Reads r elements from V_1	Calculates the new indices (shifted by 1 left) for target vector	Writes read r words to corresponding element positions in V_2 and outgoing word to CAR	$\lceil \frac{p}{r} \rceil$ cycles
VWSHR V_2, V_1	Reads r elements from V_1	Calculates the new indices (shifted by 1 right) for target vector and taking rightmost incoming word from CAR	Writes read r words to corresponding element positions in V_2	$\lceil \frac{L}{r} \rceil$ cycles
MTVCR R_2	Reads R_2 and copies to SBI	NOP	Writes value to VCR	1 cycle
MFVCR R_2	Copies value from VCR to SBI	NOP	Writes value from SBI to R_2	1 cycle

Table 4.2: Genuinely Independent Vector Instructions (cont.)

PIVI execution

In a *Partially Independent Vector Instruction*, the calculation on every element of the vector instruction depends on the calculation of the neighboring elements: the functions concerned by this category are **VADDU**, **VSPMULT**, **VSAMULT** and **VMOVE**. For optimal scheduling of the PIVI instructions we will assume that each VPU has an internal ‘temporary’ 32-bit register.

We assume that each VPU has a 32-bit Carry Select Adder (CSA): for each addition operation, the addition is performed for both cases where ‘incoming’ carry is 0 or 1 and the ‘correct’ output is determined once the correct carry is known as illustrated in Figure 4.3. The bottleneck here would be the propagation of the carry during the VEXC stage. The larger r is, the longer the critical path will be thus most probably reducing the maximum clock frequency.

VADDU V_2, V_1, V_0 : The **VADDU** instruction performs the addition of two long precision numbers made up of the l -least significant elements of V_0 and V_1 .

- *VDF stage*: Fetching the r elements of V_1, V_0 .
- *VEXM stage*: Addition of the corresponding i^{th} elements using the CSA. The CSA performs two addition operations in parallel: one for an incoming 0 carry and one for an incoming 1 carry.
- *VEXC stage*: The correct i^{th} result is chosen as a function of the carry from the $i - 1^{st}$ addition. For the least significant word (index 0 for example) the carry is taken from the CAR register.
- *VWB stage*: The result is written to V_2 .

Following instructions can be issued $\lceil \frac{L}{r} \rceil$ cycles later.

VSPMULT V_2, V_1, R_0 : The **VSPMULT** instruction multiplies R_0 to V_1 where the scalar register is considered as a polynomial of degree 31 (max) in a Galois Field of characteristic 2 and the vector register as a polynomial whose coefficients are represented by the different elements of the vector (the element 0 being the least significant terms). The basic operations involved are

- Multiplication of each element j of V_1 by R_0 , giving in each case a 64-bit value (so holding on two words which we will call from now on the ‘Lower’ (LO_j) and the ‘Higher’(HI_j) words for respectively the 32 least significant bits and the 32 most significant bits of the result.
- For each element j , XORing the Lower word of the result with the Higher word of the from position $j - 1$. The result is written as the j^{th} element of the destination vector register.

Actually this is the simplest of the PIVIs to be executed in the sense that there is no carry propagation. We ‘only’ have to propagate once the Higher words to the successive elements of the vector register. Hence we have the following schedule for each stage i of the VPU’s pipeline:

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

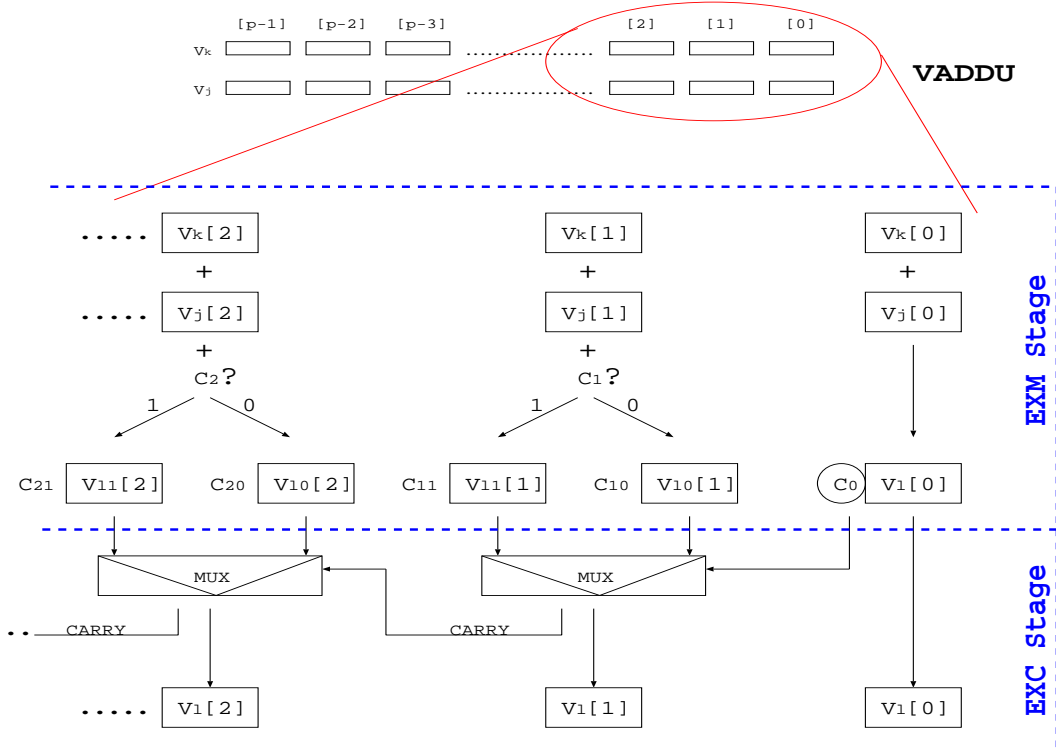


Figure 4.3: Execution of VADDU instruction

- *VDF stage*: Fetching the r elements of V_1 and R_0 (which is written to SBI).
- *VEXM stage*: Polynomially multiplying each element i of the r elements of V_1 with SBI producing for each i a LO_i and a HI_i .
- *VEXC stage*: Each element LO_i is XORed with HI_{i-1} and (LO_0) with CAR.
- *VWB stage*: For each i the result is written to V_2 . (HI_r) is written to CAR.

Following instruction can be issued $\lceil \frac{p}{r} \rceil$ clock-cycles later.

VSAMULT V_2, V_1, R_0 : The sequence of operations performed in this operation is more or less similar to the previous one except that this time instead of working on polynomials we execute arithmetic operations and the XOR operations are replaced by additions. We hence have the following decomposition:

- *VDF stage*: Fetching the r elements of V_1 and R_0 (which is written to SBI).
- *VEXM stage*: Multiplying each element i of the r elements of V_1 with SBI producing for each i a LO_i and a HI_i .
- *VEXC stage*: Each element LO_i is added to HI_{i-1} (and (LO_0) with CAR) with the CSA. The carry selection is then done.
- *VWB stage*: For each i the result is written to V_2 . (HI_r) is written to CAR.

Following instruction can be issued $\lceil \frac{p}{r} \rceil$ clock-cycles later.

MAVI execution

Looking back at the Appendix A, we have three MAVI instructions: VBYTELD, VLOAD and VSTORE. Each VPU would have its own software managed memory through which the VPU performs accesses by bytes (with 4 bytes in parallel) for the VBYTELD instruction and by 32-bit words for VLOAD and VSTORE. With such an arrangement the issue rate would be $\lceil \frac{p}{r} \rceil$.

Chaining of vector instructions & hazard elimination

At this stage, the main type of hazard we are confronted with is the data hazard. Data hazards occur when the instruction I has as an operand the result from the preceding instruction ($I - 1$). With our vector operations, data hazards occur when an instruction takes only 1 or 2 cycles. For instructions having a larger number of iterations, the latency incurred by the multi-iteration process diffuses the data dependency. Table 4.3 summarises the hazards. Note that whenever a pipeline stall occurs after the Instruction Decode stage of instruction I , this stall is echoed to instruction ($I + 1$) which suffers from a similar stall after the Instruction Fetch stage.

I-1	I	Description	Pipeline Stall	Bypass Required
GIVI	GIVI	Calculation done at the VEXC stage	No stall	Data forwarded from the VEXC stage of $I - 1$ to the VEXC stage of I
GIVI	PIVI	The PIVI needs the result at its VEXM stage	After ID stage	Data forwarded from the VEXC stage of $I - 1$ to the VEXM stage of I
PIVI	GIVI	The GIVI needs the result at its VEXC stage	No stall	Data forwarded from the VEXC stage of $I - 1$ to the VEXC stage of I
PIVI	PIVI	The PIVI needs result at the VEXM stage	After ID stage	Data forwarded from the VEXC stage of $I - 1$ to the VEXM stage of I

Table 4.3: Data Dependencies on the vector instructions

4.2 The VeMICry Functional Simulator

This section discusses simulation tools, the functional simulation of VeMICry and results obtained.

4.2.1 Simulation tools

The SimpleScalar simulation tool

SimpleScalar is an architecture simulation tool which allows software modelling of micro-architectures for program performance analysis [Burger & Austin (2004)]. The point in this section is not to give the details of this tool but to explain why we started to look at it before dropping it in favour of ArchC. For further details about the tool itself, obtaining the source and install files, please refer to [Burger & Austin (2004)] or to the website www.simplescalar.com.

We used the (*cycle-accurate*) `sim-outorder` simulator to parameterize the PISA¹ to make it as close as possible to the MIPS modelled in MIPSsimTM (which is the cycle accurate instruction set simulator from MIPS Technologies) used to test the scalar codes for AES and for modular multiplication. SimpleScalar also provides a program compiler and assembler which is a “modified” version of GNU’s GCC which can target the PISA’s instruction set architecture. Using the latter tools, we compiled the scalar AES previously simulated on MIPSsimTM. For the generated code, we had the following figures.

Scalar AES	MIPSsim TM	SimpleScalar
Code Size(bytes)	9232	18384
# of instruction executed	2308	2298
# of clock cycles	3264	4128

Table 4.4: Scalar AES-128

From Table 4.4 we see that AES running on PISA had a CPI (Cycles Per Instruction) which was 1.5 times too high even when we removed the cache from the architecture. The explanation for this major difference is that PISA has a pipeline where most instructions complete in 5 stages except for the LOAD/STORE instructions which each take 6 stages. The additional stage is the MEMORY SCHEDULER in between the DISPATCH and the EXECUTION stages. A further analysis showed that the simulated AES code contains 593 LOAD instructions and 375 STORE instructions giving a total of 968 which, in terms of order of magnitude, is close to the difference in number of clock-cycles in Table 4.4.

PISA is an implementation of the 32-bit MIPS-IV architecture but each instruction is coded on 64 bits. This explains why SimpleScalar yielded a code whose size was twice bigger than that of MIPSsimTM.

¹PISA is an architecture implemented in SimpleScalar to simulate a MIPS-compatible Instruction Set Architecture.

We then tried to modify SimpleScalar to insert new scalar instructions (namely MOVZ and MOVN) to first see how difficult it is to ‘hack’ this tool before modifying the architecture to add the vector architecture. We updated the `pisa.def` file. Prior to recompiling the tool we have to modify and compile the binary utilities (notably the files `./include/opcode/mips.h` and `./opcodes/ss-opc.c`). But we never managed to successfully recompile the binary utilities not even to get the modified PISA architecture running. Given this big impediment and the complexity of the PISA architecture which generated large differences in simulation statistics with the commercial MIPSsimTM, we decided to concentrate on another architecture toolset.

The ArchC simulator

The ArchC tool is an architecture description language which is developed by the Computer Systems Laboratory of the Institute of Computing of the University of Campinas (www.archc.org). With this tool we built an architectural instruction simulator which is composed of:

- A language description illustrating the target architecture including the memory hierarchy and the instruction set architecture.
- A simulator generator which uses the above description language to generate a Makefile which is then used to build a **SystemC** model.

Apart from being neatly constructed and well documented [[ArchC \(2004\)](#)], ArchC is based on a widely used commercial tool **SystemC**. It allows us to build quite simple architectures but it is sufficient for our needs. Moreover, the simulation software builder is based on the original GCC tool available at www.gnu.org. Hence it is easier to modify the instruction set.

Several architectural descriptions can be downloaded from www.archc.org, two of which are particularly interesting for us, being compatible with the 32-bit MIPS-I instruction set architecture:

- A functional simulator called MIPS-I which already implements most of the 32-bit scalar MIPS instructions
- A Cycle-Accurate simulator called the R3000 which is a pipelined description of the MIPS-I

4.2.2 Building the functional VeMICry

Installing the ArchC tools

Before installing the ArchC tool, the SystemC tool has to be installed from www.systemc.org. Then the ArchC tool can be downloaded and installed as explained in the first chapter of [[ArchC \(2004\)](#)]. The GNU compiler has to be installed as detailed in [[ArchC \(2003\)](#)], especially the part where GCC has to be re-targeted for ArchC.

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

Building the functional model

For the compiler, we modified the GCC tools to add our vector instructions inside the assembler. Concerning the architecture simulator, the backbone of the VeMICry model is composed of the definition files of the MIPS-I model which we have upgraded to add our vector instructions. In the initial model:

- We have 8 vector registers.
- Each vector is composed of 8 elements of 32-bits each (depth of 8)
- We assume that we have also 8 functional units.
- We assume that each instruction is executed in 1 clock-cycle (only a functional model).

The VeMICry is described in the following files:

- **vemicry.ac**: To define the hardware resources available, the hardware architecture of our processor
- **vemicry-isa.ac**: To define the instruction decoding
- **vemicry-isa.cpp**: To define the instruction execution (and hence the architecture of the VeMICry).

Once the architecture has been defined in these three files, the `acsim` tool is used to generate the makefile `Makefile.archc`. The latter is run to build the SystemC simulator model called `vemicry.x`. The generated simulator is called to run an input object file which is in fact our test cryptographic software compiled with GCC. The simulator generates a series of basic statistics like the sequence of instructions executed (`vemicry.dasm`), a trace of the Program Counter (`vemicry.trace`) and the occurrences of each instruction along with the number of cycle-counts (`vemicry.stats`).

4.2.3 Simulating the AES

Scalar AES: The functional simulation of the scalar AES code revealed that the number of instruction cycles to be 3283 for the encryption part and 519 cycles for the key schedule. This more in line with the 3264 cycles of the MIPSsim tool than the figure previously given by the SimpleScalar simulator.

Vectorised AES: The vector instructions are used to optimize the SHIFTRROWS, MIXCOLUMNS, ADDRROUNDKEY and SUBBYTE operations. The KEY_SCHEDULE is implemented as a separate routine. We validated the results generated by our vector AES encryption code. Simulations show that encrypting 16 bytes (for a 128-bit AES) takes 160 instruction cycles (see code in Appendix B.1). In addition to this the KEY_SCHEDULE took 246 instruction cycles. These figures represent a large gain in performance when compared to the same algorithms run on the scalar MIPS. There is a significant gain in performance not only because we are able to work on the all four columns of each AES block in parallel but

also because we have dedicated instructions tailored for some of the AES basic operations.

More performance gain is achieved when we encrypt larger data files. We ran simulations where we encrypted 32 bytes with one same key, i.e. we ran the `KEY_SCHEDULE` once and the encryption codes was modified to work on 8 words of each vector register. Encrypting 32 bytes took 182 instruction cycles. This illustrates a major advantage of our architecture: depending on the depth of vector registers, we are able to encrypt large data tables with little performance penalty: for example the masking (and un-masking) of the data or the key can be done at little cost (with the `VXOR` operation) at the beginning (and end) of the algorithm. Moreover, the look-up tables can be dynamically randomized to thwart classical DPA-like attacks.

Another big advantage with our approach is that robust software countermeasures (like those described in [Akkar & Giraud (2001)]) can be implemented to compensate for any side-channel information leakage.

4.2.4 Modular multiplication in binary fields

We implemented Montgomery's modular multiplication in \mathbb{F}_{2^m} as described in Figure 4.4.

```

Input   :  $a(x), b(x), f(x), M$  and  $N_0(x)$ 
Output :  $c(x) = a(x).b(x).x^{-32M} \bmod f(x)$ 

```

```

 $c(x) \leftarrow 0$ 
for  $j = 0$  to  $M - 1$  do
   $c(x) \leftarrow c(x) + A_j(x) \cdot b(x)$ 
   $M(x) \leftarrow C_0(x) \cdot N_0(x) \bmod x^{32}$ 
   $c(x) \leftarrow c(x) + M(x) \cdot f(x)$ 
   $c(x) \leftarrow c(x)/x^{32}$ 
endfor
return  $c(x)$ 

```

Figure 4.4: Montgomery Modular Multiplication on a 32-bit architecture

Scalar Montgomery multiplication in $GF(2^m)$

The scalar code corresponding to the modular Montgomery's Multiplication in F_{2^m} was compiled and simulated on the functional VeMICry architecture. Calculating the constant $N_0(x)$ takes 984 instruction cycles and the multiplication takes 22331 instruction cycles.

Vector modular multiplication for ECC

Vectorised code for modular multiplication is given in Appendix B.2. The calculation of N_0 takes 22 instruction cycles and the main part of the modular multiplication takes 97 instruction cycles.

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

Note that our test values are taken from the field $GF(2^{191})$, which means that the data values have a maximum length of 192 bits. If each vector register has 8 elements, we could work on up to 256-bits ECC with the same code, which would be far from what would be required for the next 20 years or so [Lenstra & Verheul (2001)].

In the above example, we perform a reduction by 32 bits each time. However, one could envisage performing a reduction by 64 bits as this would mean that we would have half as many loops. In the algorithm depicted in Figure 4.4, each word is on 64 bits, which means that the calculated N_0 is also on 64 bits and also the we shift by 64 bits at the end. We only perform half the number of loops. While doing so, we found out that the calculation of N_0 took 72 instruction cycles and the modular multiplication itself took 84 instruction cycles. Given that the calculation of N_0 depends only on the irreducible polynomial (modulus), it can be calculated only once at the beginning of the signature algorithm. To compare performance results, we can focus only on the multiplication algorithm. Hence the performance gain when doing a 64-bit reduction is of the order of 13% compared to the same algorithm implemented with a reduction by 32 bits. Even if we have only half as many loops, this advantage is largely counter-balanced by the fact that we have a rigid 32-bit architecture (for the vector registers) and the ‘cost’ to emulate a 64-bit one.

4.2.5 Modular multiplication for RSA

For RSA Rivest *et al.* (1978), we work with the modular multiplication in prime fields. We work on much larger data fields which can range between 1024 to 2048 bits or even 4096 bits. This is where we will see the real impact of our design choices (like the depth of each vector register) on the size and performance of the generated code size.

Modular Multiplication is implemented based on Montgomery’s method [Montgomery (1985)]. Efficient implementations of the latter algorithm are given in [Çetin Koç *et al.* (1996); Tenca & Çetin K. Koç (1999)]. We first implemented the CIOS (*Coarsely Integrated Operand Scanning*) method as described in [Çetin Koç *et al.* (1996)]¹. To calculate $R = A \times B \bmod N$ we use Montgomery’s method which yields $R' = A \times B \times r^{-1} \bmod N$ where N is long precision modulus of length l bits and r can be chosen such that $r = 2^l$. If each l -bit data Y can be decomposed into a linear combination of 32-bit integers denoted by Y_i such that (with $M = \lceil \frac{l}{32} \rceil$):

$$Y = Y_{M-1} \cdot 2^{32(M-1)} + \dots + Y_1 \cdot 2^{32} + Y_0 \quad (4.1)$$

We then have the algorithm in Figure 4.5 for the RSA’s modular multiplication. J_0 is pre-calculated as the multiplicative inverse of N_0 modulo 2^{32} .

The algorithm is implemented in Assembly language using the vector instructions given in Appendix A. The resulting code is given in Appendix B.3. On the functional simulator, the code takes 4095 instruction cycles. In this first version of the VeMICry, each vector register has 8 elements. Hence to run a 1024-bit modular multiplication we

¹Note that this was the approach implemented for the ECC modular multiplication in the previous section.

Input : A, B, N, M and J_0 Output : $R' = A.B.2^{-32M} \bmod N$
<ol style="list-style-type: none"> 1. $R' \leftarrow 0$ 2. for $j = 0$ to $M - 1$ do 3. $R' \leftarrow R' + A_j \cdot B$ 4. $J \leftarrow R'_0 \cdot J_0 \bmod 2^{32}$ 5. $R' \leftarrow R' + J \cdot N$ 6. $R' \leftarrow R' / 2^{32}$ 7. endfor 8. return R'

Figure 4.5: Modular multiplication for RSA with the CIOS method

have to implement an inner loop working on chunks of 256 bits (each vector register being composed of 8 words of 32 bits each) to calculate $R' \leftarrow R' + A_j \cdot B$ or $R' \leftarrow R' + J \cdot N$ while taking care to propagate the carries¹.

Improving the inner loop

We also investigated the FIOS (*Finely Integrated Operand Scanning*) approach [Çetin Koç *et al.* (1996)]. The instruction cycle count is reduced to 3296. This 19.5% gain in performance is achieved at the expense of one additional vector register. These results seem to be in contradiction to those shown in [Çetin Koç *et al.* (1996)] where the CIOS method outperforms the FIOS one by around 7.6%. This is because in the latter paper, even if there are fewer loops in the FIOS method, the potential gain is counterbalanced by the higher number of memory reads and writes. In our vector architecture, this increase in memory accesses has less impact because of the architecture of our vector register file.

Input : A, B, N, M and r Output : $R' = A.B.2^{-32M} \bmod N$
<ol style="list-style-type: none"> 1. $R' \leftarrow 0$ 2. for $j = 0$ to $M - 1$ do 3. $J \leftarrow (R'_0 + B_0 \cdot A_j) \cdot r \bmod 2^{32}$ 4. $R' \leftarrow R' + A_j \cdot B + J \cdot N$ 5. $R' \leftarrow R' / 2^{32}$ 6. endfor 7. return R'

Figure 4.6: Modular Multiplication for RSA with FIOS method

¹One important observation is that when calculating $D = C + A \times B$ where C and B are on k words each and A is one word long, in the worst case, D will be on $k + 1$ words.

4.3 Quantitative Analysis

In this section, we perform a quantitative analysis of our vector cryptographic processor using the functional simulator built using ArchC. We look at the performance of the modular multiplication between two long precision numbers, which is the building bloc of the RSA encryption scheme. Other algorithms like AES or ECC can provide a very high level of security with relatively short keys (usually between 160 and 256 bit-long) and only RSA is the interesting situation to consider for large numbers (of the order of thousands of bits) [Lenstra & Verheul (2001)]. The performance of the modular multiplication (based on Montgomery’s algorithm as explained earlier) is studied for different configurations of the VeMICry, i.e. by changing the depth p of each vector register and the number of lanes r (in other words the number of processing units working in parallel).

4.3.1 Tools set-up & methodology

The simulator for the functional VeMICry has been built in such a way that the depth of the vector registers can be redefined through the variable `vr_b_p` in the file `vemicry.h`. The simulator is then recompiled as explained in Section 4.2.2.

Simulating the variation in the number of lanes is a little more tedious because we have only built a ‘functional’ simulator. To emulate the lanes, we will suppose that for an architecture of depth p having r lanes, each of the vector instructions will have a latency of $\lceil \frac{p}{r} \rceil$ clock cycles. From there we can redefine the latency of the vector instructions to simulate this behavior. ArchC allows us to define a minimum and maximum limit for the number of clock cycles for each instruction using the object `cycle_range(min,max)`. During simulation, the simulator feeds two accumulators where one is increased by the *min* value and the other by the *max* value each time an instruction is executed. The content of these two accumulators are then presented as the min and max number of clock cycles taken by the program in the generated `vemicry.stats` file. Note that when, for a given instruction, the `cycle_range` is not defined, the `min` and `max` values are taken to be equal to 1 for that instruction.

To quantify the number of cycles taken by the `Mongo` routine (given in Appendix B.3), we execute the test program once with the routine to collect the number of cycles returned in the file `vemicry.stats` before executing a second time without the routine and note the number of cycles. The difference between the two numbers of cycles gives the one corresponding to the `Mongo` routine.

The measurements are taken for some characteristic values of data length, register depth and number of lanes. The collected data are displayed on graphs. The curves are plotted using MATLABTM by fitting the best curve that passes through the measured values: the best curve was determined using MATLABTM’s *cubic spline interpolant*. The resulting spline is a cubic polynomial that passes through all the measured points and provides a continuous and smooth function. Note that our multiplication routine supports any key length.

4.3.2 Varying key size

We first varied the size of the key length used and we measured the number of cycles taken to execute a modular multiplication (*This also allowed us to verify that the implemented vector code can work for different key sizes*). We performed measurements for key sizes of 512, 1024, 2048 and 4096 bits. In theory we expect the number of clock cycles to be multiplied by 4 every time we double the size of the key. This is because, if we refer back to the algorithm described in Figure 4.6, we not only double the size of the outer loop starting at line 2 but we also double the size of the inner loop required to calculate line 4, hence the factor of $2 \times 2 = 4$.

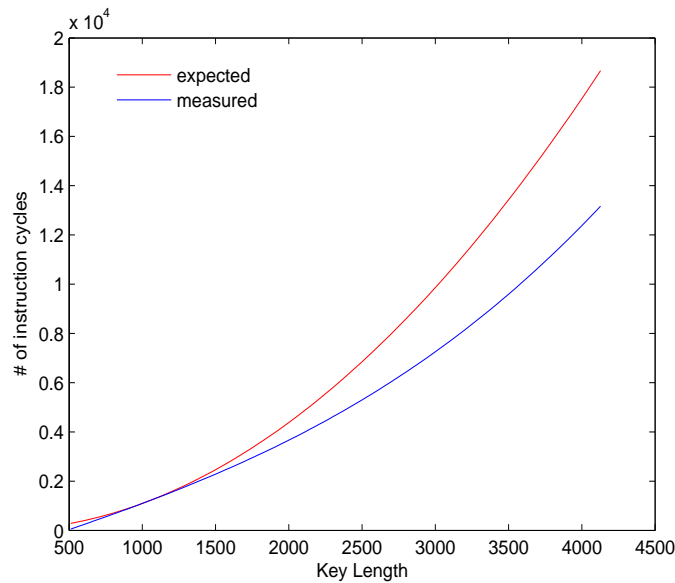


Figure 4.7: Number of cycles *versus* key size (Vector Depth= 32)

In Figure 4.7, the red curve represents what is expected and the blue one represents what is actually implemented. We can see that the performance penalty decreases as the size of the key increases.

4.3.3 Changing depth of vector registers

We repeated the above experiment but this time for different values of the depth p . In our functional model the latter can be modified by changing the variable `vrp_p` before recompiling the sources and generating the simulator. The assembly code was adequately modified.

The vectorization effect on varying key sizes

From the graphs shown in Figure 4.8 we can note that as p decreases, the measured variation in performance gets closer and closer to the theoretical result. Up to the case where $p = 1$ where in fact the practical results match the theoretical ones. The case of $p = 1$ is the limit where our vector architecture approaches a scalar one. This supports the fact

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

that with the vector architecture, we can work on larger key sizes with a performance penalty which is less than the expected one. This can be viewed as a relative gain in efficiency. This behaviour could be explained by the fact we have a register-to-register vector architecture where large data words are loaded at once, thus reducing the accesses to external memory.

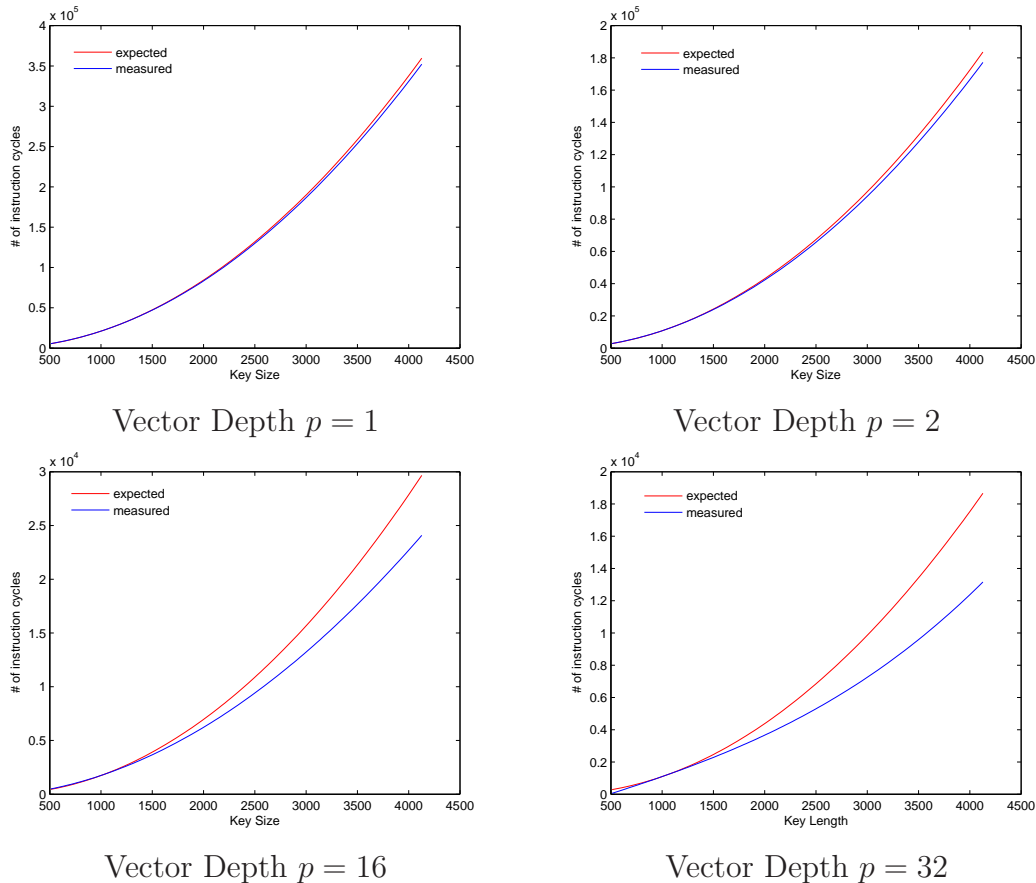
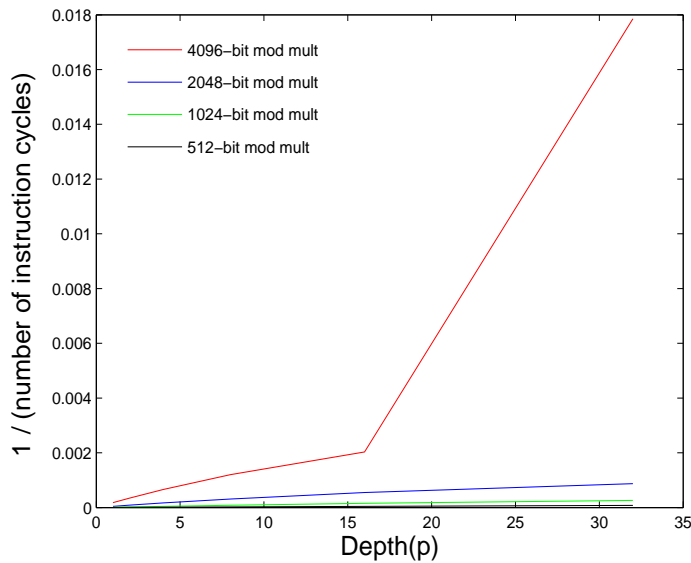


Figure 4.8: Number of cycles *versus* key size for different vector depths p

Profiling the rate of performance change with increasing depth

The other interesting aspect with the collected figures is to see how, for a given key size, the performance of modular multiplication routine is affected by changing p . Figure 4.9 shows that, for a given data size, the number of cycles is inversely proportional to the depth p .

However we can also note that the rate at which the number of cycles changes decreases with p . To have a closer look at this phenomenon, we had a look at how the rate of decrease in the number of cycles varies by doubling p . We obtain the curve in Figure 4.10 where we see that, for any data length, there is a characteristic drop in the rate of gain in performance beyond the point where $p = 16$.

Figure 4.9: Performance *versus* Vector Depth

The above observation is particularly interesting for calibrating our vector architecture because then we can bear in mind that the gain in performance in going from $p = 16$ to $p = 32$ (which is actually about 37%) would be lower than the gain in performance when going from $p = 8$ to $p = 16$ (about 43%). This could be counter-balanced with other consequences of doubling the size of the vector register (like die size, power consumption, etc).

4.3.4 Varying the number of lanes

The other aspect we looked at was the effect of varying the number of lanes. Since we are only working on a functional simulator, we did not actually implement different number of lanes (i.e. have different numbers of VPUs). Instead we simulate the variation of the number of lanes r by varying the latency of the vector instructions as depicted in Section 4.3.1. By latency, we actually refer to the instruction issue rate. From Figure 4.2, we can be made to wait $\lceil \frac{p}{r} \rceil$ cycles before issuing the next instruction, where p is the vector length or depth and r is the number of lanes. Only the VEXTRACT, MTVCR and MFVCR instructions can be considered to take one cycle in any case as they only transfer one word from one register to another.

The curves in Figure 4.11 show how the number of cycles for a 1024-bit modular multiplication varies when increasing the number of lanes. For each value of p , we increase r from 1 to p , doubling the value of r every time. We then ‘interpolated’ among the measured points to obtain the ‘trend’.

We see that the number of cycles taken is more or less inversely proportional to r and that as r gets larger this proportionality factor decreases. This is illustrated in Figure 4.12: the gain in performance decreases more or less linearly.

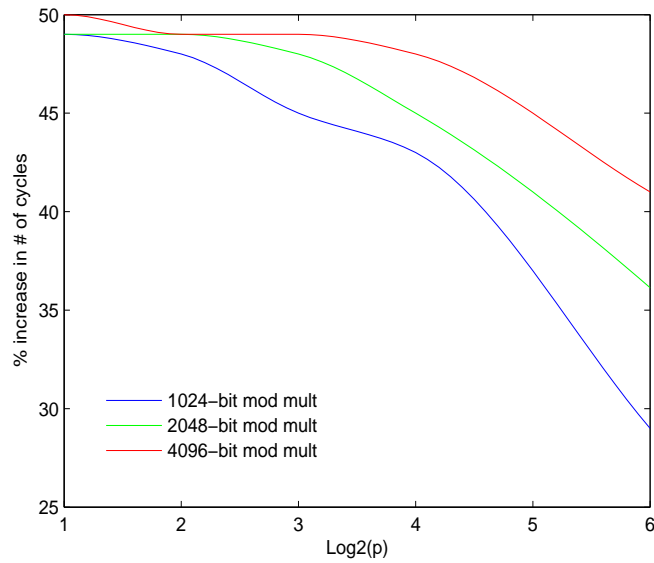


Figure 4.10: Rate of change of performance *versus* $\log_2(\text{Vector Depth})$

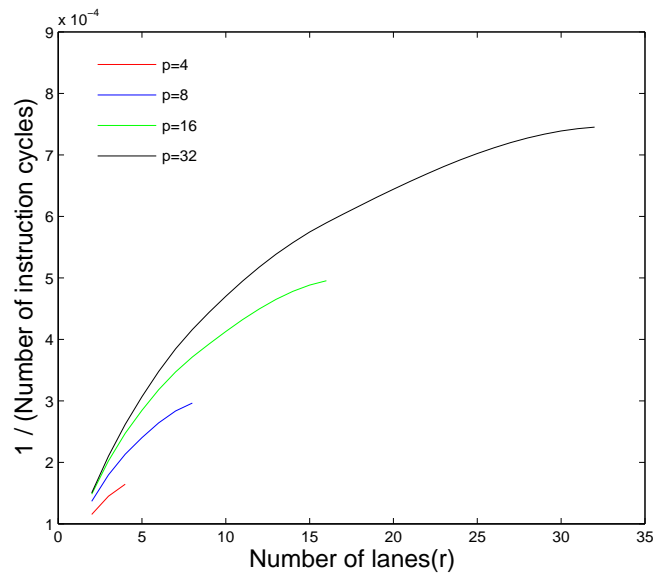


Figure 4.11: Performance *versus* number of lanes for different vector depths

4.4 Summary and Conclusions

The definition of the VeMICry architecture has allowed us to build a simulator using the ArchC tool. First of all, we verified that the vector instructions defined in Appendix A allowed us to execute algorithms like AES or RSA's and ECC's modular multiplications in an efficient and performant way.

The vector implementation of the AES takes 246 cycles for key schedule and 160 instruction cycles for the encryption part. With our vector architecture we can encrypt twice as much data in only 182 cycles. For the Montgomery's Modular Multiplication

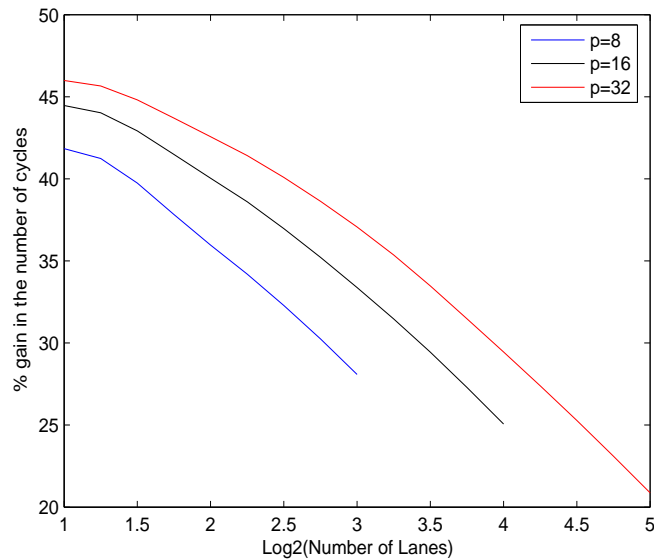


Figure 4.12: Rate of change in performance *versus* $\log_2 r$

over binary fields for Elliptic Curves' Cryptography, the vector code for the main modular multiplication loop takes 95 instruction cycles. With the *default model* ($p = r = 8$) we can work on up to 256-bit ECC with these same performance figures.

We also provided the vector implementation of a 1024-bit modular multiplication for RSA based on Montgomery's modular multiplication. The functional simulator shows that a 1024-bit modular multiplication can take 4095 instruction cycles. We studied two ways of improving our approach of the Montgomery's method: the first one is by performing 64-bit reductions instead of 32 bits and on 192-bit values we had a performance gain of 13%. The second improvement was used in cases where we work on values larger than 256 bits. We anticipate the reduction factor and perform only one inner loop (FIOS method). For a 1024-bit modular multiplication, we have a performance gain of 19.5% (3296 cycles) for a 1024-bit modular multiplication. We performed a quantitative analysis for RSA's modular multiplication on our VeMICry architecture. We found out that:

1. As key size gets bigger, the rate of increase of cycles is smaller than the theoretical values. Actually the rate of increase of instruction cycles decreases as the data sizes get bigger.
2. The above difference between the theoretical and experimental behaviours gets more important as p increases.
3. For a given length of data, increasing p decreases the number of instruction cycles logarithmically. From $p = 1$ to $p = 16$ the rate of loss in performance decreases more or less linearly but beyond $p = 16$ there seems to be a more important loss.
4. Increasing the number of lanes decreases the number of instruction cycles logarithmically. For a given p , the more lanes we have the smaller is the relative gain in performance.

4. VEMICRY ARCHITECTURE & FUNCTIONAL SIMULATION

During the functional simulations performed on the modular multiplications, we saw that the codes are executed in constant time (measurements are made in terms of number of instruction cycles), which is a *sine-qua-non* condition for an SPA¹-proof implementation. Moreover, having such a co-designed hardware-software approach is efficient for flexible secure implementations as software countermeasures can be implemented with little performance loss as opposed to a fully hardware implementation where it is sometimes impossible to have a fully secure implementation or where the cost of added security can be prohibitive.

¹Simple Power Analysis

Chapter 5

Cycle Accurate Verilog Model of VeMICry

This chapter describes the implementation of the synthesisable Verilog model of the VeMICry architecture. This implementation of the VeMICry architecture follows the description given in Chapter 4 and is composed of:

- a scalar RISC processor implementing an instruction set which is compatible to a MIPS-I instruction set.
- a vector co-processor to execute RISC-like vector instructions that are tailored to have secure software implementations of modular multiplication in fields of characteristic 2.

This implementation does not integrate all of the vector instructions described in Appendix A: the aim here was to study how pipelining and architectural hazards would be handled on a concrete implementation of the VeMICry. The other aim is to have an idea of how area and estimated power varies as a function of the different parameters of the vector co-processor.

5.1 Verilog Implementation of Scalar Processor

The scalar processor is a 5-stage pipelined 32-bit scalar MIPS-like processor. The memory architecture is a Harvard, i.e. separate memories for instructions and data. In the current implementation, the Instruction Memory and the Data Memory are implemented as simple single cycle memories, which on a ‘real’ processor could be instruction and data caches that can be accessed in one clock cycle. The memories are implemented as arrays of bytes meaning that even if we have a 32-bit architecture, the memories are byte addressable. Let us now look at the five stages of the pipeline:

- **Instruction Fetch Stage (IF):** The instructions are fetched from the Instruction Memory. The address of the fetched instruction is stored in the Program Counter (PC) register. The value of the PC is incremented by 4 and is stalled when there is a load stall or a jump stall or a vector pipe stall. Likewise, in case of a “taken branch” or a jump, the target address, which has been calculated in the EX stage,

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

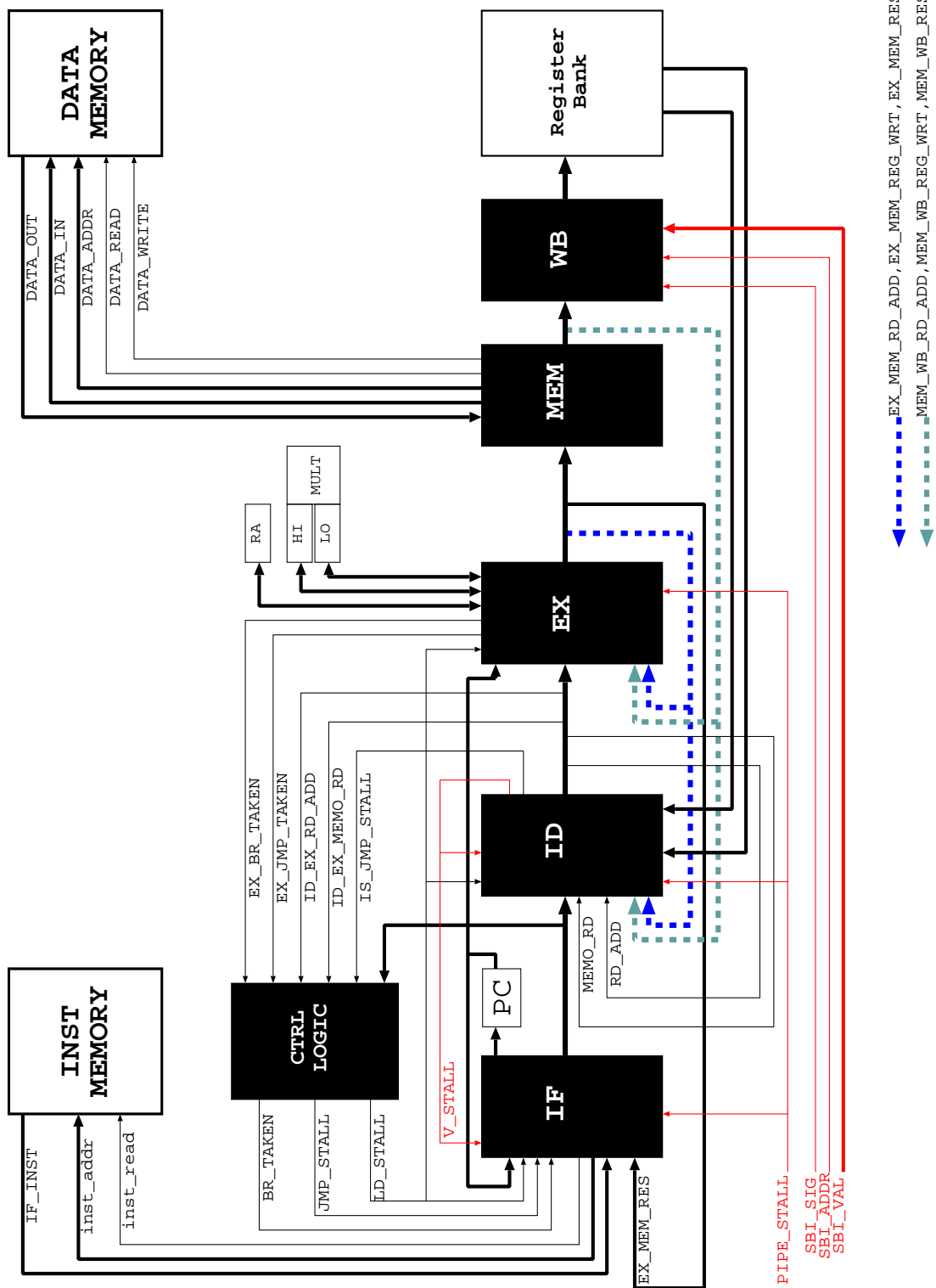


Figure 5.1: Architecture of the scalar MIPS

is written to PC. Note that this stage only accesses the Instruction Memory from which 32-bit instruction words are fetched.

- **Instruction Decode Stage (ID):** The fetched instructions are decoded and the “Jump” control signal is determined. The ID stage also decodes the vector instructions (see details in Section 5.2). Data fetch from the scalar register bank is performed at this stage. To take into account data hazards (see Section 5.1.2), the first level of register forwarding is implemented at this stage (from the outputs of the MEM and the EX stages, as illustrated in Figure 5.1).
- **Execute Stage (EX):** The execute stage performs the arithmetic and logic operations of the processor. A second level of register forwarding is also implemented in this stage on input registers RT and RS (from the outputs of the MEM and EX stages). The control signals “Branch Taken” and “Jump Taken” are also set at this stage and the corresponding new PC addresses are also determined at this stage and passed to the IF stage at the following clock cycle. Note that in the current implementation, the multiplier is directly implemented using Verilog’s multiplication operator: the results are written to two dedicated registers namely **HI** and **LO**.
- **Memory Access Stage(MEM):** The memory stage sets signals for either reading or writing data from or to the Data Memory. At this stage only the Data Memory can be accessed. At any given clock cycle, only one instruction reaches that stage, which means that we never have the case where there are conflicting accesses (for example trying to read and write data during the same clock cycle) to the Data Memory. At this stage, the scalar processor will also fetch data from the vector co-processor if a **VEXTRACT** instruction is in the pipe.
- **Register Write Back Stage (WB):** At this stage, any result from the EX or the MEM stage is written to the register bank. The register bank consists of 32 registers of 32-bits each. The **zero** register is read only and always returns zero.

5.1.1 Scalar Instruction Set

The instruction set of the scalar processor is a subset of the 32-bit MIPS-I instruction set [MIPS (1996)]. The instructions can be classified into three types: the R-Type, the I-type and the J-Type as illustrated in Figure 5.2 with the following fields:

- **OP** : Operation code
- **RS** : Source register
- **RT** : Target register
- **RD** : Destination register
- **SHMT** : Immediate shift value
- **FUNC** : Function code for $OP = 0$
- **IM16** : Immediate value (16 bits)

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

- **ADDR26** : Absolute address (26 bits)
- **[x]** : Size ‘x bits’ of field

R-Type Scalar Instructions

OP[6]	RS[5]	RT[5]	RD[5]	SHMT[5]	FUNC[5]
-------	-------	-------	-------	---------	---------

I-Type Scalar Instructions

OP[6]	RS[5]	RT[5]	IM[16]
-------	-------	-------	--------

J-Type Scalar Instructions

OP[6]	ADDR[26]
-------	----------

Figure 5.2: Scalar Instructions’ Structures

The list of scalar instructions supported by the current design of the scalar processor is given in the table below. Note that instructions that are “not implemented” are simply decoded by the instruction decoder but it is a NOP that is actually executed. Note also that in the current implementation, instructions like ADD, ADDI, SUB, MULT, SLT and SLTI actually only work on **unsigned** numbers, even if by definition they are supposed to work on signed numbers.

ADD	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s + R_t$ on signed values
ADDI	R_t, R_s, im_{16}	I-Type	$R_t \leftarrow R_s + im_{16}$ on signed values
ADDIU	R_t, R_s, im_{16}	I-Type	$R_t \leftarrow R_s + im_{16}$ on unsigned values
ADDU	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s + R_t$ on unsigned values
AND	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s \& R_t$ (logical AND)
ANDI	R_t, R_s, im_{16}	I-Type	$R_t \leftarrow R_s \& im_{16}$ (logical AND)
BEQ	R_t, R_s, im_{16}	I-Type	if $R_s = R_t$ branch, $PC \leftarrow PC + (im_{16} \ll 2)$
BGEZ	R_s, im_{16}	I-Type	if $R_s \geq 0$ branch, $PC \leftarrow PC + (im_{16} \ll 2)$

5.1 Verilog Implementation of Scalar Processor

BGEZAL	R_s, im_{16}	I-Type	if $R_s \geq 0$ branch, $RA \leftarrow PC, PC \leftarrow PC + (im_{16} \ll 2)$
BGTZ	R_s, im_{16}	I-Type	if $R_s > 0$ branch, $PC \leftarrow PC + (im_{16} \ll 2)$
BLEZ	R_s, im_{16}	I-Type	if $R_s \leq 0$ branch, $PC \leftarrow PC + (im_{16} \ll 2)$
BLTZ	R_s, im_{16}	I-Type	if $R_s < 0$ branch, $PC \leftarrow PC + (im_{16} \ll 2)$
BLTZAL	R_s, im_{16}	I-Type	if $R_s < 0$ branch, $RA \leftarrow PC, PC \leftarrow PC + (im_{16} \ll 2)$
BNE	R_t, R_s, im_{16}	I-Type	if $R_s \neq R_t$ branch, $PC \leftarrow PC + (im_{16} \ll 2)$
BREAK		R-Type	not implemented
DIV	R_d, R_s, R_t	R-Type	not implemented
DIVU	R_d, R_s, R_t	R-Type	not implemented
J	$ADDR_{26}$	J-Type	$PC \leftarrow (ADDR_{26} \ll 2)$
JAL	$ADDR_{26}$	J-Type	$RA \leftarrow PC, PC \leftarrow (ADDR_{26} \ll 2)$
JR	R_s	R-Type	$PC \leftarrow R_s$, jump to address in R_s
JALR	R_d, R_s	R-Type	not implemented
LB	$R_t, (im_{16})R_s$	I-Type	$R_t \leftarrow (0_{24} \parallel MEM[R_s + im_{16}])$ (8 bits)
LBU	$R_t, (im_{16})R_s$	I-Type	$R_t \leftarrow (0_{24} \parallel MEM[R_s + im_{16}])$ (8 bits)
LH	$R_t, (im_{16})R_s$	I-Type	$R_t \leftarrow (0_{16} \parallel MEM[R_s + im_{16}])$ (16 bits)
LHU	$R_t, (im_{16})R_s$	I-Type	$R_t \leftarrow (0_{16} \parallel MEM[R_s + im_{16}])$ (16 bits)
LUI	R_t, im_{16}	I-Type	$R_t \leftarrow (im_{16} \parallel 0_{16})$
LW	$R_t, (im_{16})R_s$	I-Type	$R_t \leftarrow MEM[R_s + im_{16}]$ (32 bits)
LWL	$R_t, (im_{16})R_s$	I-Type	not implemented

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

LWR	$R_t, (im_{16})R_s$	I-Type	not implemented
MFHI	R_d	R-Type	$R_d \leftarrow HI$
MFLO	R_d	R-Type	$R_d \leftarrow LO$
MOVN	R_d, R_s, R_t	R-Type	if $R_t \neq 0$ then $R_d \leftarrow R_s$ else NOP
MOVZ	R_d, R_s, R_t	R-Type	if $R_t = 0$ then $R_d \leftarrow R_s$ else NOP
MTHI	R_s	R-Type	$HI \leftarrow R_s$
MTLO	R_s	R-Type	$LO \leftarrow R_s$
MULT	R_s, R_t	R-Type	$(HI, LO) \leftarrow R_s \times R_t$ on signed values
MULTP	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s \times R_t$ in GF(2)
MULTU	R_s, R_t	R-Type	$(HI, LO) \leftarrow R_s \times R_t$ on unsigned values
NOP		R-Type	No operation
NOR	R_d, R_s, R_t	R-Type	$R_d \leftarrow \sim (R_s R_t)$
OR	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s R_t$
ORI	R_t, R_s, im_{16}	I-Type	$R_t \leftarrow R_s im_{16}$
SB	$R_t, (im_{16})R_s$	I-Type	$MEM[R_s + im_{16}] \leftarrow R_t$ (8 bits)
SH	$R_t, (im_{16})R_s$	I-Type	$MEM[R_s + im_{16}] \leftarrow R_t$ (16 bits)
SLL	$R_d, R_s, shmt$	R-Type	$R_d \leftarrow R_s \ll shmt$ bits (to the left)
SLLV	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s \ll R_t$ bits (to the left)
SLT	R_d, R_s, R_t	R-Type	if $R_s < R_t$ (signed) $R_d \leftarrow 1$ else $R_d \leftarrow 0$
SLTI	R_t, R_s, imm_{16}	I-Type	if $R_s < imm_{16}$ (signed) $R_t \leftarrow 1$ else $R_t \leftarrow 0$

5.1 Verilog Implementation of Scalar Processor

SLTIU	R_s, R_t, im_{16}	I-Type	not implemented
SLTU	R_d, R_s, R_t	R-Type	if $R_s < R_t$ (unsigned) $R_d \leftarrow 1$ else $R_d \leftarrow 0$
SRA	$R_d, R_s, shmt$	R-Type	not implemented
SRAV	R_d, R_s, R_t	R-Type	not implemented
SRL	$R_d, R_s, shmt$	R-Type	$R_d \leftarrow R_s \gg shmt$ bits to the right
SRLV	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s \gg R_t$ bits (to the right)
SUB	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s - R_t$ on signed values
SUBU	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s - R_t$ on unsigned values
SW	$R_t, (im_{16})R_s$	I-Type	$MEM[R_s + im_{16}] \leftarrow R_t$ (32 bits)
SWL	$R_t, (im_{16})R_s$	I-Type	not implemented
SWR	$R_t, (im_{16})R_s$	I-Type	not implemented
SYSCALL		R-Type	not implemented
XOR	R_d, R_s, R_t	R-Type	$R_d \leftarrow R_s \oplus R_t$
XORI	R_t, R_s, im_{16}	I-Type	$R_t \leftarrow R_s \oplus im_{16}$

Compiler for scalar code

The test programs used to debug and test the scalar design were mostly written in Assembly language. The compiler used is the GNU GCC tool, revision 3.3.1 available from [GNU (2005)], that can target MIPS architectures. The encoding and register/memory addressing used for the above instructions is fully compatible with the MIPS one defined in the GCC tools.

5.1.2 Implementation details

The scalar processor was designed from scratch. It is expected to have a behavior similar to publicly known features of the MIPS architecture. However differences may exist, namely some simplifications have been made (due to the targeted applications) and some sophisticated mechanisms have been omitted. For example, the register bypass mechanisms or the multiplier architecture may be different.

In the following subsections, we detail some of the architectural design choices made, most of which have an impact on the software written, in other words anyone writing a

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

program (or eventually a dedicated compiler) for that processor should have the following processor behavior in mind. We also detail how data hazards are taken into account.

Processor start-up

First note that the processor is **big endian**, i.e. within a 32-bit data the most significant byte is at the lowest address.

In order to start the processor, an asynchronous reset must be applied on the **reset** input of the processor. This input is active High. Upon reset, the PC is set to zero and the first instruction is automatically fetched at that address on the next rising clock edge. This means that the first fetched instruction is at address zero of the Instruction Memory.

Upon reset, the control signals are set to zero in order to make sure that the pipeline is not stalled and the program execution is not stopped. Thus the asynchronous reset signal has been propagated to the clocked blocks in between each combinational block within each pipeline stage.

Pipeline hazards

Pipeline Hazards are fully explained in [Hennessy & Patterson (2003)] and the design of the scalar processor takes into account these hazards. Given the architecture of the scalar processor we are concerned with three types of hazards: branch hazards, data hazards and structural hazards.

Branch Hazards: Branch hazards occur during the execution of a branch or jump instruction as the pipeline is disrupted by the fetch of an instruction which is not necessarily the following one. The main effect of such hazards is to reduce the performance of the processor. To limit the ‘cost’ of having a branch that is ‘taken’ we chose to first make sure that

- the instruction immediately after a branch or jump instruction is always executed. This is anyway a classic MIPS behaviour.
- when a branch or a jump instruction is decoded in the ID stage, at the next rising clock edge, a NOP instruction is injected at the IF stage.

The above features are illustrated in Figure 5.3 when the following sequence of instructions is executed:

```
BR Skip
INST1
INST2
INST3
INST4
INST5
...
Skip: INST6
```

INST7

...

Note that the branch instructions are relative branches and the resulting jump address is coded as a signed ‘immediate’ 16-bit values.

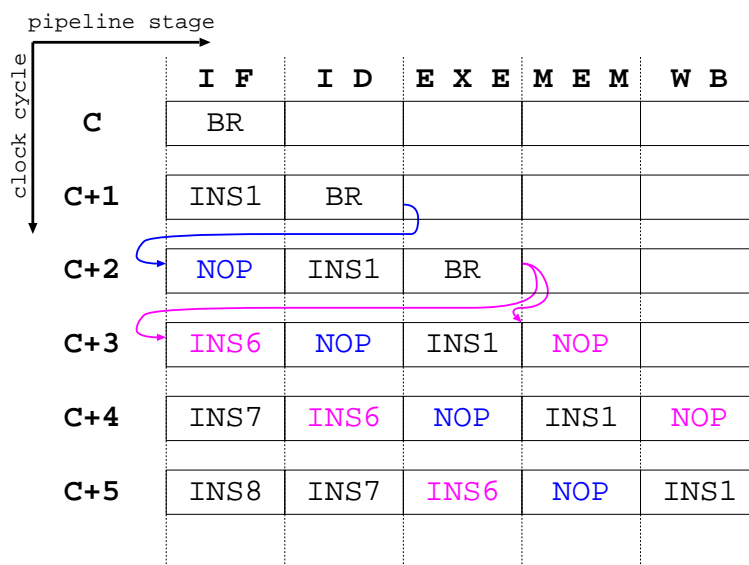


Figure 5.3: Execution of a taken branch instruction

Data hazards: Data hazards occur when an instruction I has one of its operands which is the result of the instruction I-1 or instruction I-2, result which is not yet ready at the moment the instruction I performs its data fetch. In our architecture, data is fetched at the ID stage and results are available after the EX or the MEM stage. There are two main types of data hazards.

- *Write After Read (WAR) hazards:* These occur when an instruction I attempts to write data while an instruction I-1 is trying to read it. This hazard is not relevant to our architecture because the instructions are executed in sequential order. Regarding the register file, such a hazard cannot occur because data fetches from the register file are done during the ID stage and data is written to the register file during the WB stage which comes after the ID stage. For memory-accessed data, we have a Data Memory interface that assumes that any data is accessed within one clock cycle. Moreover all memory accesses are done by the MEM stage which ensures that, at a given clock cycle, we exclusively have a memory read or a memory write.
- *Read After Write (RAW) hazards:* These are a bigger concern to our architecture. RAW hazards occur when an instruction I tries to read data that has just been modified by a previous instruction and that the new value has not yet been written at the targeted register or memory location. In our architecture, this is particularly relevant to data accessed from the register file.

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

We will distinguish between three levels of RAW hazards.

- A *Level 0* RAW hazard occurs when, at a given clock cycle C ¹, a given pipeline stage needs to perform a calculation on data which it has just calculated at the clock cycle $C-1$ (and for which the write-back occurs at a later pipeline stage). In our architecture this would happen at the EXE stage for which a register bypass mechanism has been implemented from the output of the EXE stage to the input of the same EXE stage. Note that such a register bypass is not implemented for memory-accessed data because in our architecture, data is read from or written to the Data Memory only during the MEM stage.
- A *Level 1* RAW hazard occurs when, at a given clock cycle C , a pipeline stage needs to fetch data that was calculated by a following pipeline stage at cycle $C-1$. To account for such hazards without any performance penalty register bypass mechanisms have been implemented
 - from the output of the EXE stage to the input of the ID stage (for cases where a register value is updated following an arithmetic or logic operation and then this register is read two instructions later).
 - from the output of the MEM stage to the input of the EXE stage (for cases where an instruction tries to load data from memory into a register which is then the source register of the next instruction for an arithmetic or logic operation).
 - from the output of the MEM stage to the input of ID stage (for cases where an instruction tries to load data from memory into a register from which data needs to be fetched from two instructions later).

Note that in the case of registers HI and LO², if instruction I-1 is one that modifies them, then instruction I must not be one that attempts to read or use them.

- A *Level 2* RAW hazard occurs when, at a given clock cycle C , a pipeline stage fetches a data that will be calculated at cycle $C+1$ by a following pipeline stage. Typically this would occur when the instruction I-1 is a LOAD to register R_i and the instruction I is some arithmetic operation on that register R_i . To account for such hazards, a load stall is injected in the pipeline in order to wait for the correct result to be available on the output of the MEM stage (see Figure 5.5).

The Level 0 and Level 1 RAW hazards are illustrated in Figure 5.4 and a load hazard is illustrated in Figure 5.5³.

¹i.e. at the rising edge of clock cycle C

²described when the EX stage was described

³Note that in simulations of the (post-synthesis) netlist of the scalar processor, the load stall signal does not seem to propagate properly. It is hence strongly recommended to avoid data dependencies in which the preceding instruction is a LOAD

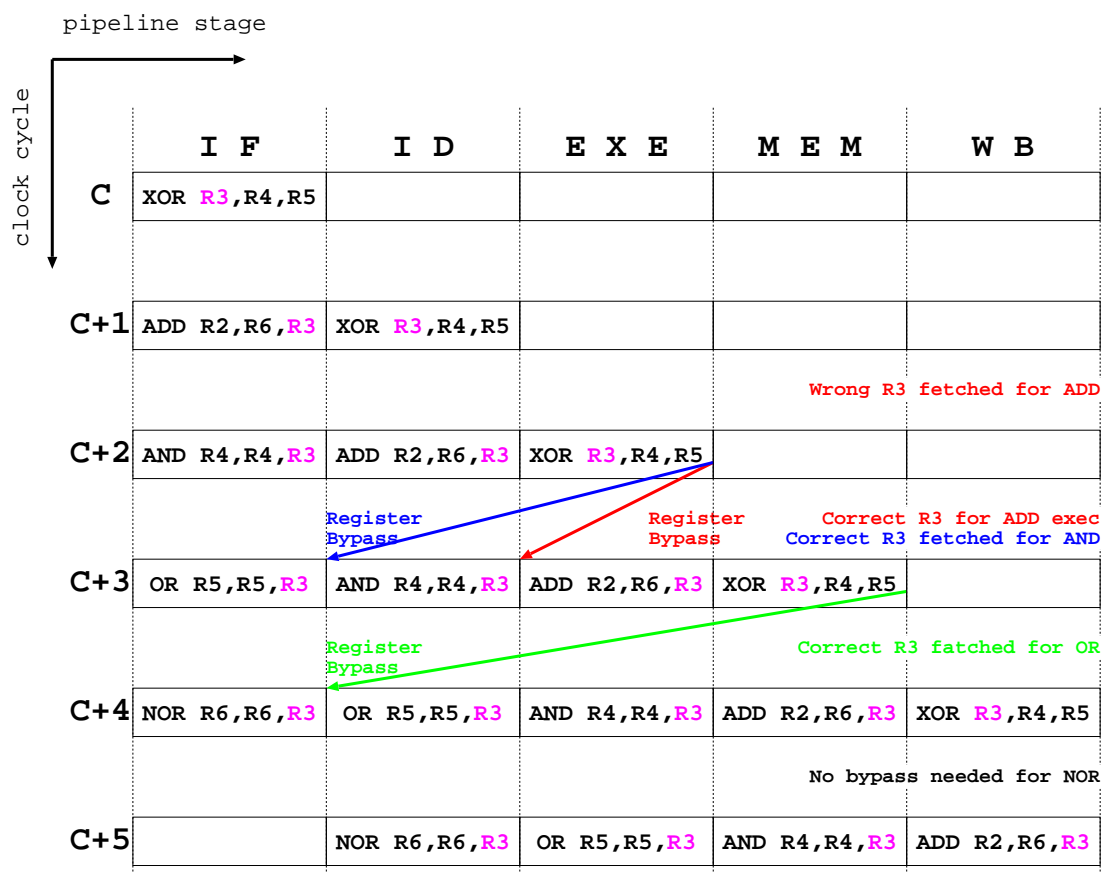


Figure 5.4: RAW Hazards & Register Bypasses

Structural hazards: Structural hazards occur when resources are shared by different pipeline stages. This is not the case in our design because each pipeline stage has its own resources. We also have separate Instruction and Data Memories.

5.1.3 Performance and area figures

The model is a cycle accurate one. On this architecture, a 192-bit modular multiplication routine in a binary field takes 25133 clock cycles. The same operation on a commercial MIPSsimTM simulator takes 22331 clock cycles, showing a discrepancy of 12% of our model. This difference may arise from the fact that the architecture simulated by the MIPSsimTM is not exactly the same as the one implemented here. Moreover, the MIPSsimTM is only a software simulator of the hardware (and not the actual hardware architecture itself) and hence it can be thought to have some behavioural (and hence performance) differences with the real core itself.

Synthesised scalar processor

The scalar processor was synthesized in a TSMC 90nm technology using Design CompilerTM from Synopsis and further measurements were done on the resulting netlist.

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

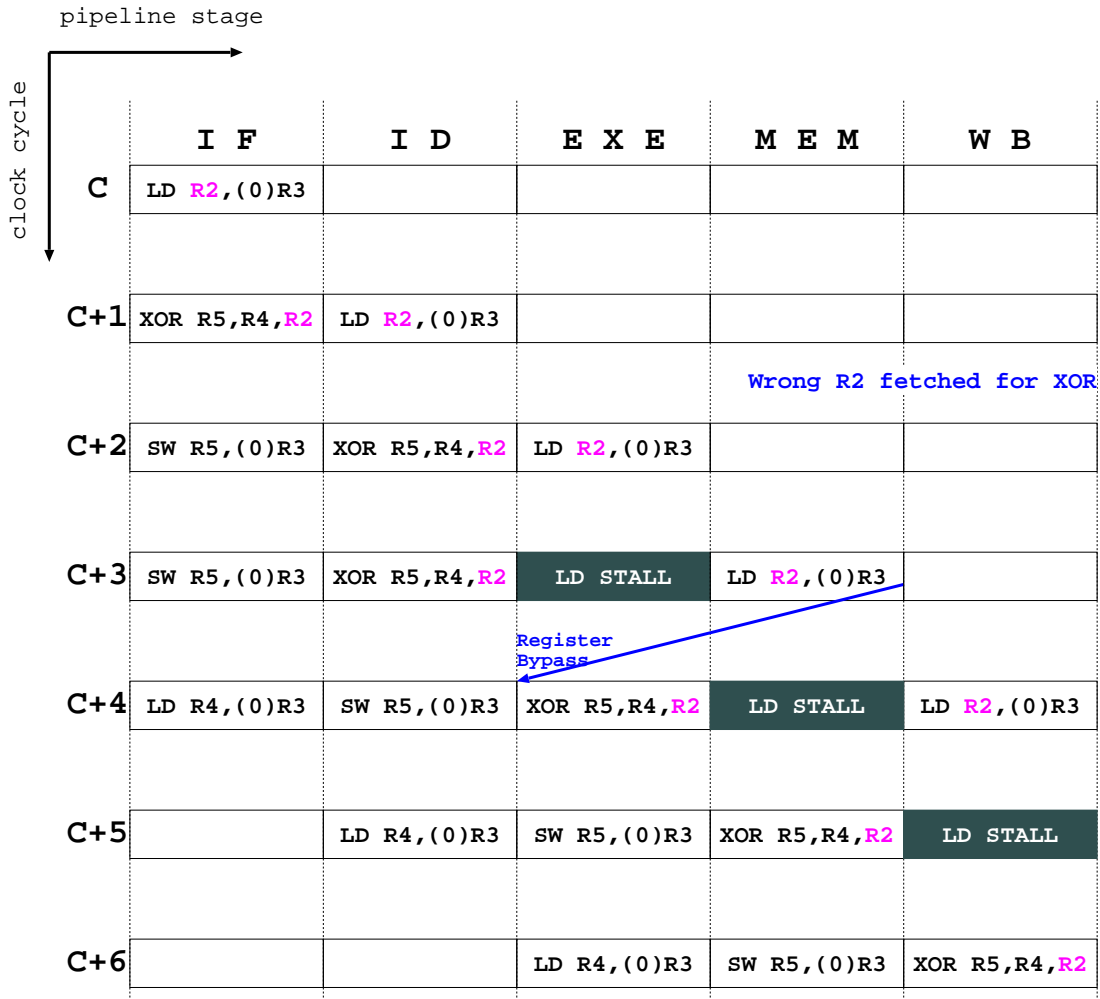


Figure 5.5: LOAD stall on scalar processor

First note that there are different versions of the synthesized scalar part: the ID stage of the scalar part has been fitted with a small state machine that stalls the instruction fetch pipeline during $\frac{v_p}{v_r}$ cycles when a vector instruction is being executed (see Section 5.2.1). This fraction is hardwired in the scalar processor, so different netlists had to be produced for values of $\frac{v_p}{v_r} \in \{1, 2, 4, 8, 16, 32, 64\}$.

The resulting circuit area is around $0.08545mm^2$. Timing reports provided by Design Compiler™ showed that we could expect the processor to run at frequencies as high as 129MHz. During the execution of the code in Appendix C, the peak power measured by PrimePower™ (see Section 6.6) was around $5.3mW$ and the mean power was $350\mu W$ (excluding the memories).

5.2 Verilog Model of the Vector Co-processor

Just like the scalar part, the vector co-processor is big endian. The architecture of the vector co-processor is determined by the way the vector register files are distributed across

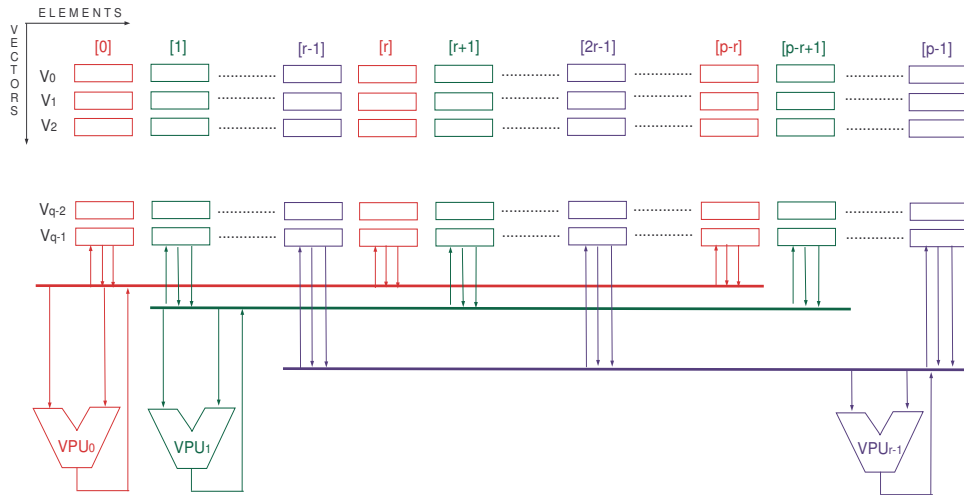


Figure 5.6: Vector Register File

the Vector Processing Units (VPUs). The general architecture is shown in Figure 5.6 where we have the following parameters:

- m : The size of each element of the vector elements. Currently $m = 32$.
- v_q : The number of such vector registers.
- v_p : The number of elements in each vector register. This will be called the *depth* of each vector register.
- v_r : The number of lanes into which the vector registers are organized. Each lane consists of a Vector Processing Unit (VPU). In an ideal situation, we would have as many VPUs as there are lanes but in our scalable design v_r is independent from v_p .

The architecture of the vector co-processor is such that the vector registers are directly entangled into the design block defining a ‘vector lane’. The first limitation of the design is that v_p has to be a multiple of v_r and given that the counter of the internal state-machine of the vector co-processor is coded on 8 bits $\frac{v_p}{v_r}$ must be less than 256.

The vector co-processor is a pipelined design as shown in Figure 5.8. The Instruction Fetch and the Instruction Decode are handled by the scalar processor. Upon detection of a vector instruction, the ID stage of the scalar sends the proper control signals to the vector co-processor which then executes the decoded vector instruction. The execution of such vector instructions is decomposed into the following stages:

- **Data Fetch (VDF)** stage where each VPU fetches the two (depending on the instruction) elements from the source vector registers. If a scalar register is involved, the value is fetched from the latter scalar register during the ID stage. Note that

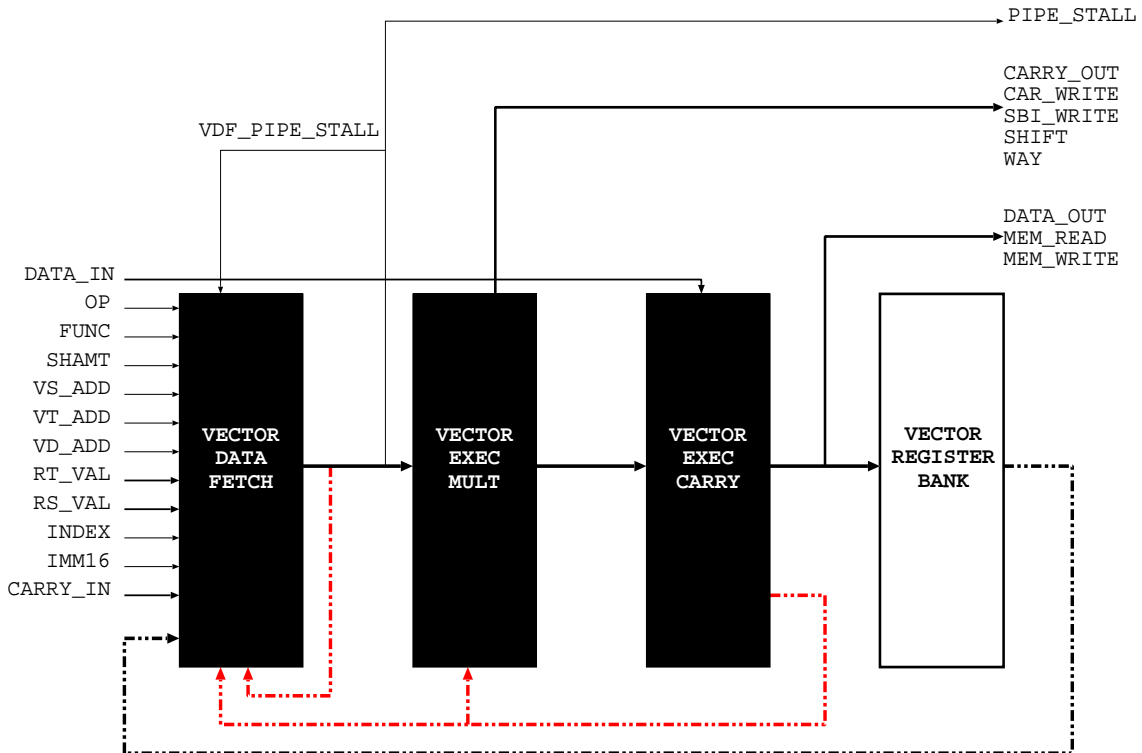


Figure 5.8: Architecture of a Vector Lane

5.2.1 Vector instruction execution

Consider the general case where we have v_r VPUs where $v_r \leq v_p$ (could be specially true for embedded processors). This means each VPU will have to be iterated $\left\lceil \frac{v_p}{v_r} \right\rceil$ times in order to apply the required operation on all v_p elements of the targeted vector registers as shown in Figure 5.9. Hence the next vector instruction will only be issued $\left\lceil \frac{v_p}{v_r} \right\rceil$ clock cycles later in the case where each “iteration” takes 1 clock-cycle.

Note that in our design, the vector registers are directly distributed across the lanes. Since v_p and v_r are parameters that are set at compile time, each vector lane is thus compiled with a register bank comprised of v_q vector registers which each in turn consists of $\frac{v_p}{v_r}$ 32-bit registers. With respect to the ‘software’ which, on the whole, sees v_q vector registers of v_p elements each, the distribution is done following the scheme from Figure 5.6. From a hardware point of view, at each i^{th} stage (comprised of VDF_i , $VEXM_i$, $VEXC_i$ and VWB_i with $0 \leq i < \frac{v_p}{v_r}$) as shown in Figure 5.9, within each vector lane, the corresponding vector operation is performed on the i^{th} element of the vector register *within the lane*. From a software point of view, this vector operation is actually being performed on element $j + (i \times v_r)$ of the targeted vector register where j is the lane number ($0 \leq j < v_r$).

This is two dimensional pipelining: one in space where each instruction executed is spread over the resources of the four stages (VDF, VEXM, VEXC and VWB) and one in time where each instruction is virtually decomposed of $\frac{v_p}{v_r}$ executions working on different

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

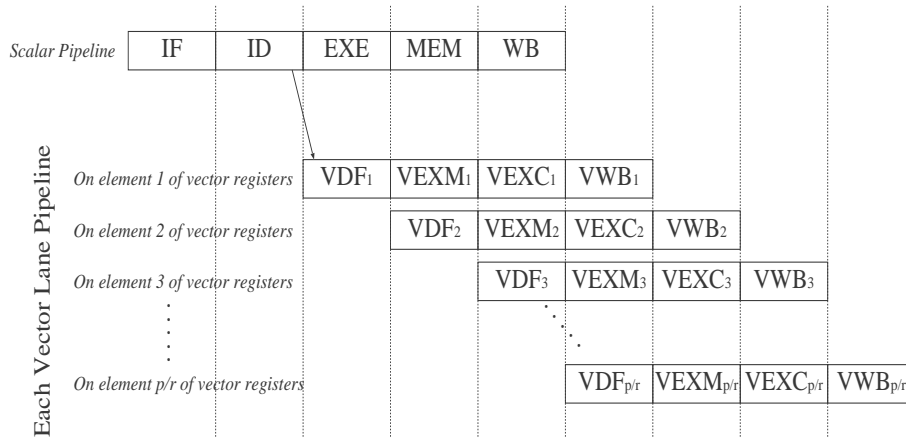


Figure 5.9: Timing relationship between scalar & vector executions

indices of the vector registers. The time dimension could also be considered as a way of handling *structural pipeline hazards* [Hennessy & Patterson (2003)]. To handle the “second dimension” of the pipeline, a small state machine has been implemented in the ID stage of the scalar processor. Upon detection of a vector instruction, an 8-bit counter is set to $\frac{v_p}{v_r} - 1$ and an index variable is initialized to zero. At each subsequent clock cycle (and until the counter reaches 0):

- a `V_STALL` signal is generated stalling the instruction fetch of the scalar processor, in other words no new instruction is fetched.
- the index variable is incremented to tell the vector co-processor which elements of the vector registers need to be worked on.
- the counter is decremented.

Note that with such a state machine, when $v_r = v_p$ the counter remains at zero and no stalling signal for the scalar processor is generated. Another design choice could have been to continue fetching and executing scalar instructions but on one side this would have generated additional control signals and hazard management units and on the other side, as we will see, our use cases comprise of test codes where there are mainly consecutive vector instructions.

5.2.2 Implemented vector instructions

We identified three classes of vector instructions as given in Definitions 4.1, 4.2 and 4.3. The structure of the vector instructions is similar to that of the scalar ones illustrated in Figure 5.2. There are only R-Type and I-Type vector instructions. The test programs are implemented in Assembly language which is compiled with the GNU GCC tools which have been modified to compile the vector instructions¹. Table 5.1 lists all the vector instructions identified during the theoretical study. However, only a subset of these vector

¹Note that to do so, we have replaced the ‘floating point instructions’ of the MIPS by our vector instructions, as a result of which the vector register addresses always have to be even numbers. This is taken into account by the “hardware” at the VDF stage.

instructions are today operational on the implemented vector co-processor as we implemented only those required as a proof-of-concept.

All of the vector instructions operate on all v_p elements of the vector registers, i.e. we don't limit the operations on only v_l with $v_l \leq v_p$ elements as suggested in the earlier theoretical studies performed for that design. This is because it makes the design much simpler and because there was no need for it in our use cases. Typically for the `VLOAD` and the `VSTORE` instructions, data are read from and written to the Data Memory in packets of v_r words at a time until all v_p data elements have been accessed.

5.2.3 Vector pipeline hazards

The vector co-processor pipeline is actually handled within each vector lane itself, i.e. each lane has its own VDF, VEXM, VEXC and VWB stages. When designing each lane, data hazards (as defined in Section 5.1.2) have been considered. Note that in this section, we only consider hazards among vector instructions themselves: the interaction between scalar and vector instructions is detailed in Section 5.3.

RAW data hazards in vector lanes

In order to account for RAW hazards without performance penalty, the following register bypass mechanisms have been implemented as illustrated in Figure 5.8.

- A data register bypass from the output of the VEXC stage to the input of the VDF stage.
- A data register bypass from the output of the VEXC stage to the input of the VEXM stage.

For the scalar processor, data hazards were mainly detected based on the address of the targeted registers. For the vector lanes, a data hazard occurs when the targeted registers have the same address and the same index within the vector registers.

The above data register bypass mechanism does not account for cases when we have data hazards for two consecutive PIVI instructions when $v_r = v_p$. A pipeline stall is generated (as illustrated in Figure 5.10). This stall signal holds the execution of the scalar IF and ID stages and the vector VDF and VEXM stages. This stall only occurs when $v_r = v_p$. For other case where $v_r < v_p$, the pipeline is already stalled to account for the structural hazard incurred by the smaller number of VPUs. In other words, when $v_r < v_p$, RAW data hazards have no performance penalty on the software. The above scenarios are illustrated in Figures 5.10 and 5.11 for the following code.

```
VSPMULT V2, V1, R6
VSPMULT V3, V2, R3
...
```

In Figure 5.10, we have the case where $v_p = v_r = 4$. We have 4 VPUs working in parallel. The result of the first `VSPMULT` is available at cycle $C + 2$ at the output of the VEXC stage and in order for the calculation of the second `VSPMULT` to begin (at the VEXM stage), a

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

VADDU	V_d, V_s, V_t	R-Type	not implemented
VBCROTR	V_t, V_s, im_{16}	I-Type	not implemented
VBYTELD	R_t, V_s, im_{16}	I-Type	not implemented
VEXTRACT	R_t, V_s, im_{16}	I-Type	if $im_{16} = 0$ $R_t \leftarrow CAR$ else $R_t \leftarrow V_s[im_{16}]$
VFVCR	R_d, V_s, R_t	R-Type	not implemented
VLOAD	V_t, R_s, im_{16}	I-Type	stores in V_t the v_p 32-bit words from address R_s
VMPMUL	V_d, V_s, V_t	R-Type	not implemented
VSADDU	V_d, V_s, R_t	R-Type	not implemented
VSAMULT	V_d, V_s, R_t	R-Type	not implemented
VSMOVE	V_t, R_s, im_{16}	I-Type	if $im_{16} = 0$ $CAR \leftarrow R_s$, if $im_{16} < v_p$ $V_t[0..im_{16} - 1] \leftarrow R_s$, else $V_t[0..v_p - 1] \leftarrow R_s$
VSPMULT	V_d, V_s, R_t	R-Type	$(CAR \parallel V_d[v_p - 1] \parallel V_d[v_p - 2] \parallel \dots V_d[0]) \leftarrow ((V_s[v_p - 1] \parallel V_s[v_p - 2] \parallel \dots V_s[0]) \times R_t) \oplus CAR$ in binary fields
VSTORE	R_t, V_s, im_{16}	I-Type	stores the v_p 32-bit words of V_s in MEM from address R_t
VTRANSP	V_t, V_s, im_{16}	I-Type	not implemented
VTVCR	V_d, R_s, R_t	R-Type	not implemented
VWSHL	V_t, V_s, im_{16}	I-Type	$(V_t[v_p - 1] \parallel V_t[v_p - 2] \parallel \dots V_t[0]) \leftarrow CAR \leftarrow V_s[v_p - 1], (V_s[v_p - 2] \parallel V_s[v_p - 3] \parallel \dots V_s[1] \parallel 0_{32})$
VWSHR	V_t, V_s, im_{16}	I-Type	not implemented
VXOR	V_d, V_s, V_t	R-Type	$V_d \leftarrow V_s \oplus V_t$

Table 5.1: Implemented Vector Instructions

5.2 Verilog Model of the Vector Co-processor

stall is required to wait for the correct result of V2 to be available. In Figure 5.11, we have the case where $v_p = 4$ and $v_r = 2$. In this case, by the time the first ‘half’ of the second VSPMULT reaches the VEXM stage, the corresponding result of the first VSPMULT is already available at the output of the VEXC stage. So no pipeline stall is needed.

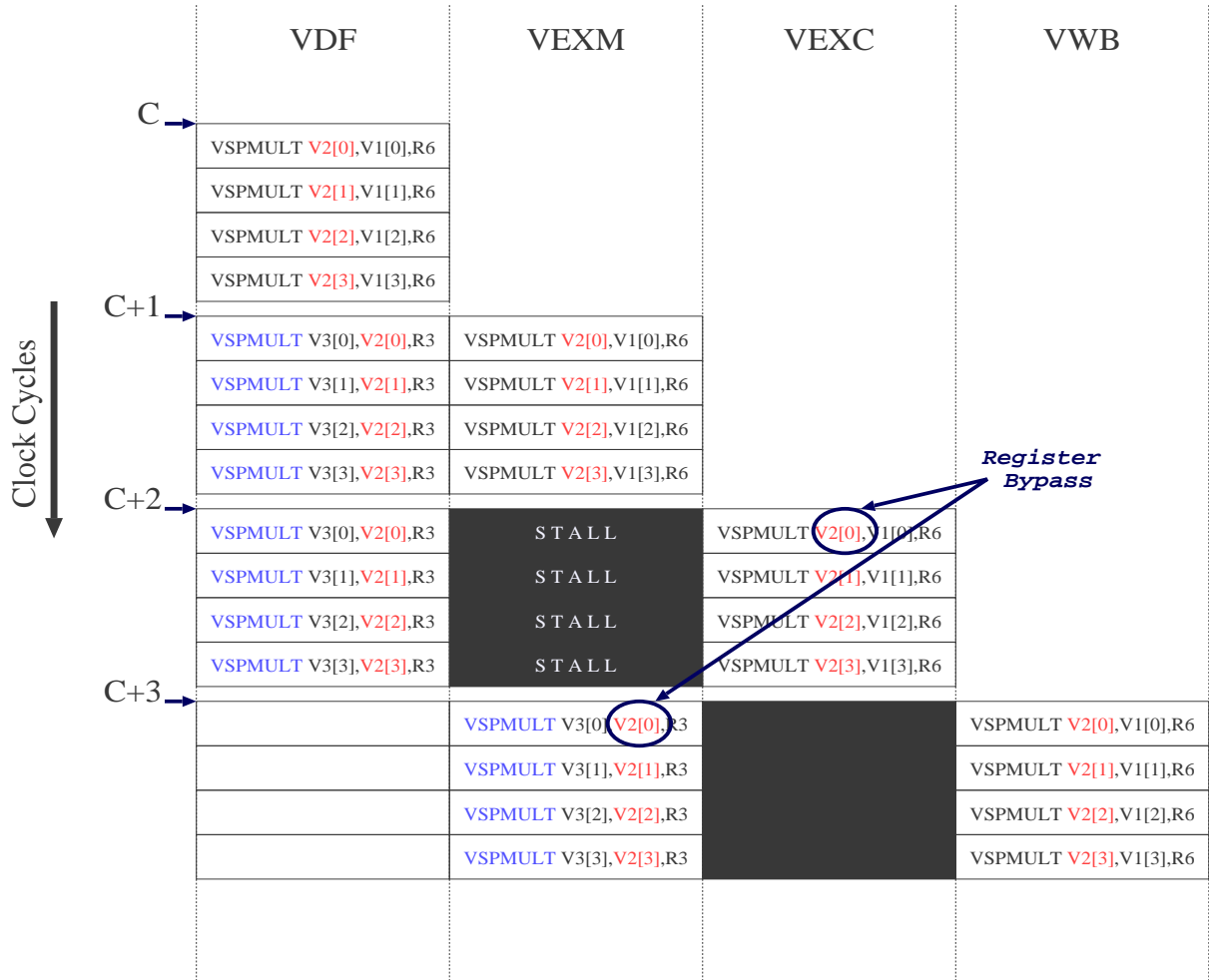


Figure 5.10: Pipeline stall for RAW data hazard when $v_r = v_p$

Structural hazards in vector lanes

In addition to the structural hazard that occurs when we have $v_r < v_p$ and that we have to stall the IF stage (as described in Section 5.2.1), a structural hazard may occur during vector loads and stores. This occurs because the same data and bus addresses (connected to the external Data Memory) are used during these operations and conflicts occur specially in cases when $v_r < v_p$. This hazard is not corrected by the hardware and as a result we recommend that a VLOAD and a VSTORE instruction must never follow each other, at least one instruction must be added in between.

Another point to note is that in the vector co-processor all the internal registers are updated at the end of the VWB stage except the CAR register which is updated one clock

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

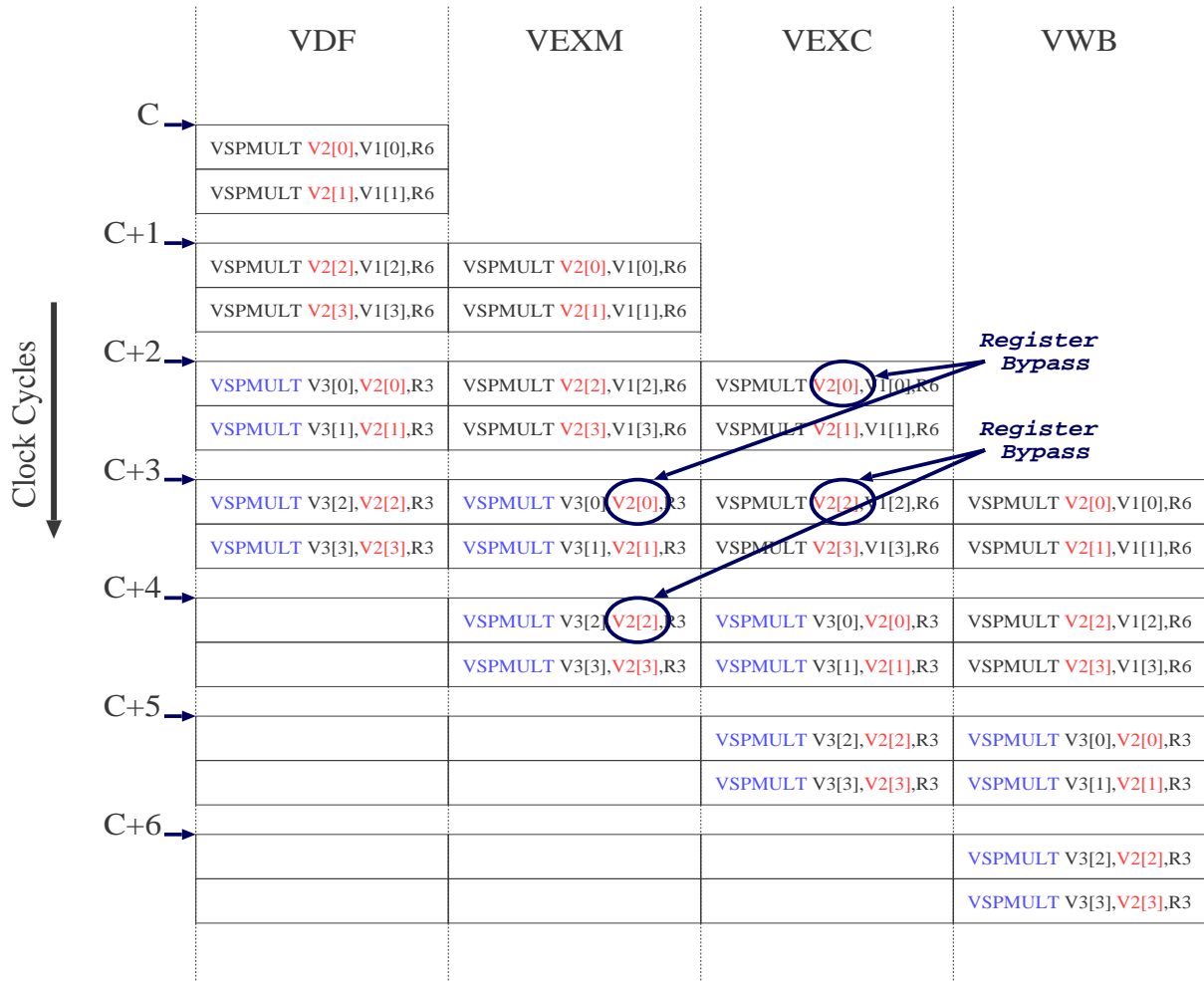


Figure 5.11: RAW data hazard when $v_r < v_p$

cycle ahead.

5.3 VeMICry system

The final design joins the scalar processor and the vector co-processor as illustrated in Figure 5.12. As already mentioned, we have two separate memories, one for the instructions and one for the data. The Instruction Memory is only read by the scalar processor's IF while the Data Memory is accessed by both computing entities. In Sections 5.1 and 5.2, we have looked at how, for each separate unit, we deal with eventual conflicting accesses to the Data Memory. Here, once both units are connected to the Data Memory, we added access control logic (a switch) to guarantee exclusive accesses of one or the other to the Data Memory. In our design, both units cannot be made to access the Data Memory at the same time because the scalar and vector units cannot work in parallel. Even if there is a scalar memory instruction in the pipe when a vector memory instruction is fetched, by the time the latter reaches its VEXC stage where it accesses the memory, the former will already have been completed.

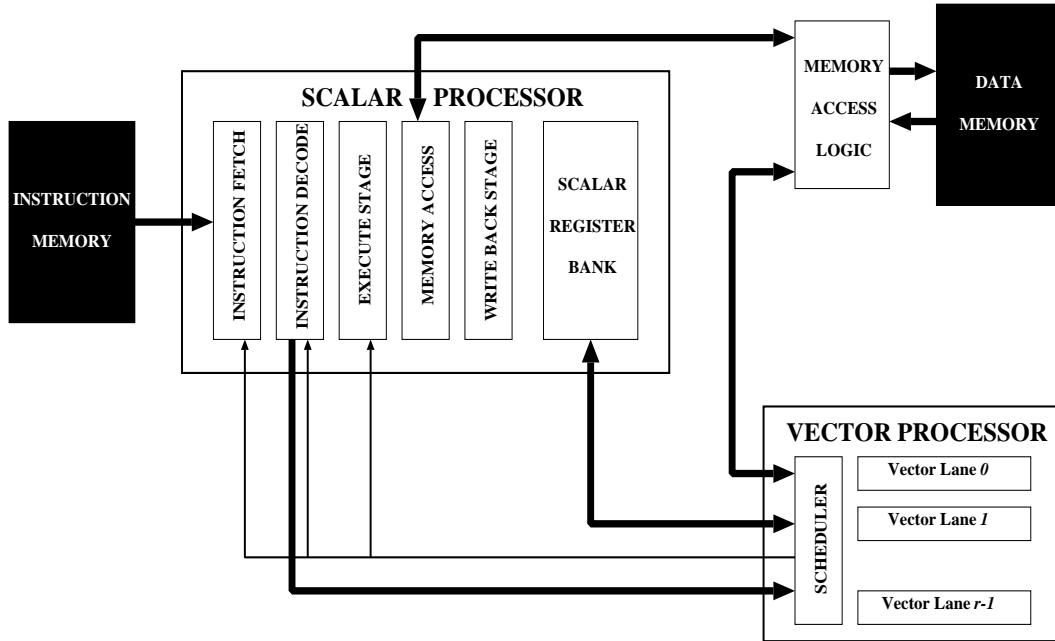


Figure 5.12: Final Architecture with scalar & vector parts

The management of data hazards between the scalar processor and the vector co-processor is mainly undertaken in software by following the rules. This choice has been made because we wanted to limit the number of control signals and control logic in the scalar processor. We want to be able to run the scalar processor on its own and this is why we limited the ‘vector-related’ logic inside the scalar part to one additional IF stall signal and a state-machine to manage the ‘iterations’ of the vector co-processor.

5.3.1 Rules for executing scalar and vector code

When executing a program that involves both scalar and vector instructions, problems might occur when we have data dependencies between successive (or neighbouring) instructions. This originates from the fact the scalar pipeline and the vector lane pipelines are independent from each other and that the latter has one more stage than the former (the ‘execute’ stage of the vector lanes is composed of two stages, VEXM and VEXC). Since most of our test codes are coded directly in Assembly language it was quite straightforward and easy to test and incorporate these rules. These rules would have to be taken into account when designing a dedicated C compiler for this processor. Suppose we have two consecutive instructions `INST1` and `INST2` where one of the inputs of `INST2` is the output of `INST1`:

1. In general, if `INST1` is a scalar instruction and `INST2` is a vector instruction, insert an independent¹ instruction in between. This can be done by either re-organizing the code or by inserting a `NOP`.
2. If the `INST1` is a scalar `LD` and `INST2` is a vector instruction, insert at least two independent instructions in between.

¹One that does not use the ‘conflicting’ data

5. CYCLE ACCURATE VERILOG MODEL OF VEMICRY

3. In general, if INST1 is a vector instruction and INST2 is a scalar one, insert at least three independent instructions in between. Given the set of vector instructions in Section 5.2.2, the only case where this can happen is for the when INST1 is a VEXTRACT.
4. If INST1 is a VEXTRACT and INST2 is a vector instruction (**even with no data dependency**), insert a NOP in between.

These rules are quite easy to follow in software and would have added unnecessary complexity to the hardware.

5.4 Summary

In this chapter, we have presented a Verilog cycle accurate synthesizable model of our vector co-processor. We first implemented a cycle accurate scalar processor based on the MIPS-I instruction set. This scalar processor has a five stage pipeline designed in such a way as to have an issue rate of 1 instruction per clock cycle. To achieve this, pipeline hazards are resolved. The design offers register bypass mechanisms to remove data hazards. Structural hazards are avoided by having exclusive resources for each pipeline stage, e.g. having separate Instruction and Data Memories. For Branch hazards, a “*not taken*” branch prediction is implemented whereby the instruction following a branch is always executed. The scalar part has been synthesized in TSMC’s 90nm technology. The resulting circuit has a area of 0.085mm² and could reach clock frequencies as high as 129MHz. The scalar processor is also used to drive the vector co-processor. The former fetches and decodes the vector instructions and sends the control signals to the latter. The vector co-processor has been designed to be scalable in the sense that the vector register depth v_p and the number of lanes v_r can be configured at compilation/synthesis time. The vector co-processor is a proof-of-concept and this is why the implemented vector instructions are for the time being limited to those needed to implement modular multiplication in binary fields (for Elliptic Curve Cryptography). It is composed of 4 pipeline stages where data hazards have been taken into account through register bypasses. Structural hazards may occur on the vector part when $v_r < v_p$, in which case the instruction fetch and decode are stalled. For the overall system (scalar + vector parts), the memory models used for the Instruction and Data memories are simple RAM memories, which, in a real system, could be a first model for caches. To test the design(s), software test code is assembled using the same compiler used for testing the functional simulator. Scalar and vector commands can be executed from the same stream of program given the ‘rules’ detailed in this chapter are followed in order to account for synchronisation issues between the scalar and the vector parts. Detailed performance, area and power studies are given in the following chapter.

Chapter 6

Analysis of Verilog VeMICry

With the design described in the previous chapter, we have a synthesisable, cycle accurate model of VeMICry for performing long precision modular multiplication in binary fields (for Elliptic Curve Cryptography). This chapter focuses on the performance, power and size of the vector part of the VeMICry processor (the analysis of the scalar part was rather straight forward - see Section 5.1.3). Note that even if the current implementation only supports instructions for Elliptic Curve modular multiplication in binary fields (where data lengths are of the order of 160-256 bits), we perform our quantitative analysis on data lengths of 256-2048 bits which are more relevant to RSA that requires modular multiplication in prime fields. By doing so, a wider spectrum of data lengths is covered. We can do so since for each vector instruction working in binary fields, we also defined an “equivalent” function working in prime fields (as shown in Appendix A). And given the pipelining used, these “equivalent” pairs of instructions are expected to take the same number of clock-cycles for the same depth of vector registers (p), the same number of lanes (r) and the same data sizes. A possible difference would be that, in the case of the instructions working in prime fields, due to the carry propagation, the area, power and critical path for each individual lane can be larger. But the vector co-processor’s area and power consumption can be expected to scale in the same way for instructions working in binary fields.

The vector part was synthesized in TSMC 90nm technology using Design Compiler™ from Synopsis. The constraints were set to target a maximum area optimization. Different configurations of the vector co-processor were synthesized for different values of vector register depths (p) and vector lanes (r) for a fixed number of vector registers (q) of 4. The resulting areas measured of course depended on the values of p and r . In Figure 6.1, we see that, for a given value of p the area of the resulting circuit varies linearly with the number of lanes used. This result confirms the modularity of our design and that the “SCHEDULER” shown in Figure 5.12 is not very much modified by the number of lanes present.

The Synopsis tool also provided timing analysis figures for the minimum clock period for the vector processor and hence the maximum clock frequency. From these data (illustrated in Figure 6.2), we can first see that the maximum frequencies are of the order of 220MHz to 290MHz. This variation is obtained by varying the number of lanes.

6. ANALYSIS OF VERILOG VEMICRY

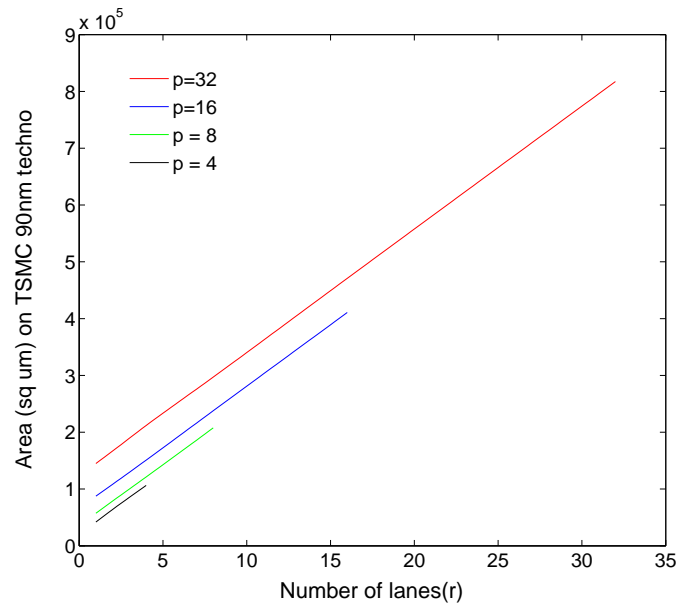


Figure 6.1: Area of vector co-processor *versus* number of lanes

Typically, increasing the number of lanes would increase the timing of the critical path to the registers shared among these lanes. However, as illustrated by the curves in Figure 6.2, for a fixed value of p , a peak is reached when we have four lanes working in parallel.

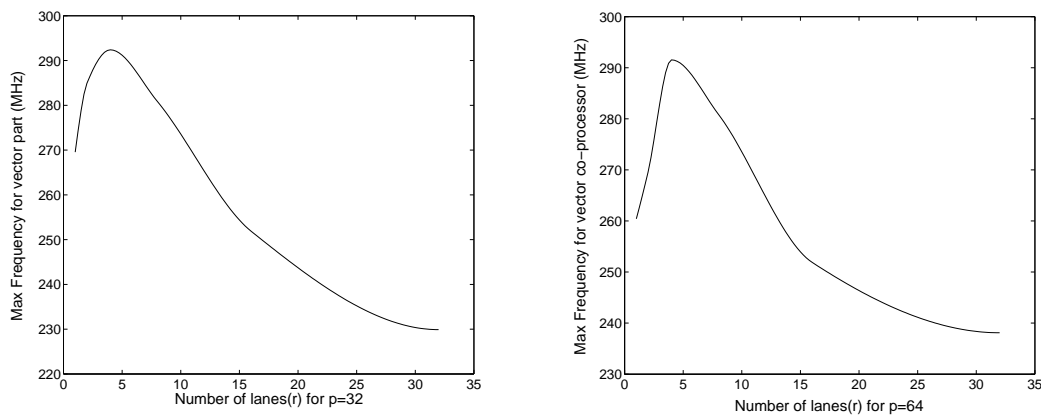


Figure 6.2: Max frequencies *versus* number of lanes

To analyze the performance of the vector part (in terms of clock cycles), we vary the size of the data on which a modular multiplication based on Montgomery's [Montgomery (1985)] FIOS [Çetin Koç *et al.* (1996)] method is performed for sizes ranging from 256 bits to 2048 bits. The test code used is presented in Appendix C.

6.1 Functional *versus* Cycle-accurate Simulation

Input : A, B, N, M and r
Output : $R' = A.B.2^{-32M} \bmod N$

1. $R' \leftarrow 0$
2. **for** $j = 0$ **to** $M - 1$ **do**
3. $J \leftarrow (R'_0 + B_0 \cdot A_j) \cdot r \bmod 2^{32}$
4. $R' \leftarrow R' + A_j \cdot B + J \cdot N$
5. $R' \leftarrow R' / 2^{32}$
6. **endfor**
7. **return** R'

Figure 6.3: FIOS Method for Montgomery’s Modular Multiplication

6.1 Functional *versus* Cycle-accurate Simulation

In Figure 6.4, we show how the number of cycles varies for varying data sizes for a targeted architecture where $p = r = 32$. With this figure we can see that, for a given hardware configuration (i.e. same values of p and r), the timing behaviours of the functional simulator and that of the cycle-accurate simulator are quite similar, with a multiplicative factor in between. This shows that the functional simulator based on the ArchC tool is useful in determining proportional performance improvements for different architectural parameters.

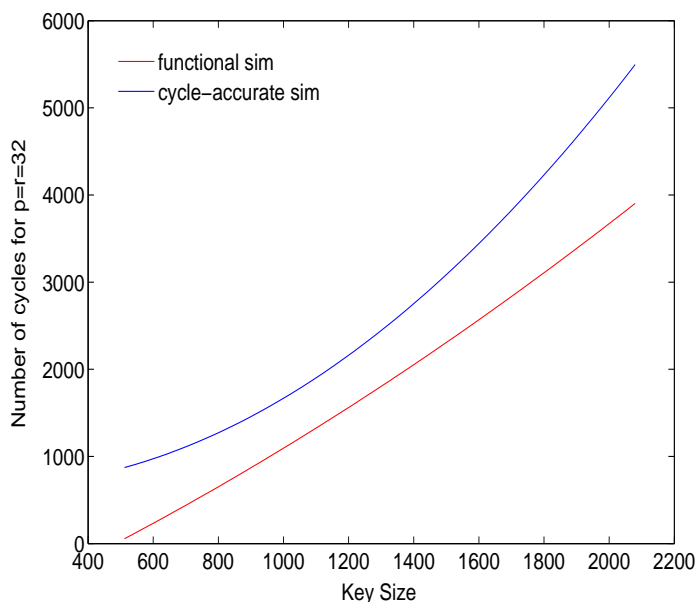


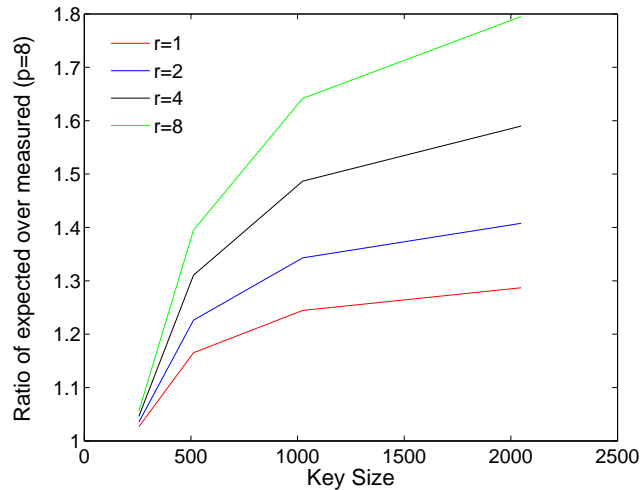
Figure 6.4: Comparing functional and cycle accurate simulators

6.2 Comparing Expected and Measured Results

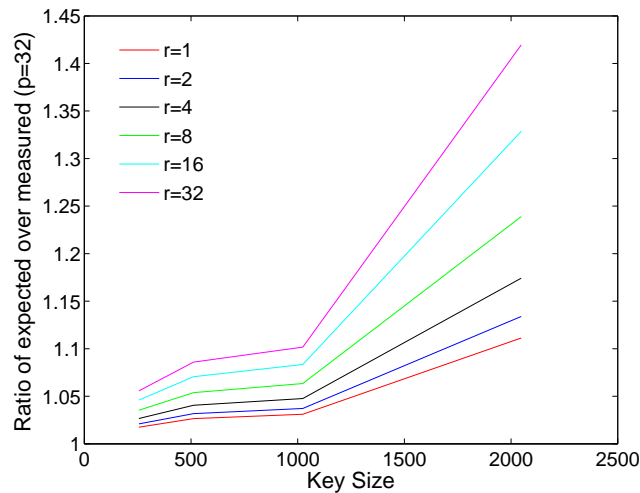
From the equations in Figure 6.3, we can derive the following theoretical rules:

6. ANALYSIS OF VERILOG VEMICRY

1. For $M \leq p$, doubling the data size would double the number of clock cycles taken by the modular multiplication routine (twice the number of loops).
2. For $M > p$, doubling the data size would multiply the number of clock cycles by 4 (twice the number of ‘for’ loops within which the line 4 is also a ‘for’ loop which is doubled).



$p = 8$



$p = 32$

Figure 6.5: Ratio of theoretical over measured variations in performance

However, as we can see from the curves in Figure 6.5, the measured performance figures are 1.5 to 2 times faster than the expected trend. The curves show the ratio of expected number of cycles over the number of measured clock cycles for a constant p and for different number of lanes r . This shows that, for the same hardware configuration (i.e. the same value of p and the same value of r), as we increase the size of the input data, the performance penalty generated by this increase in size of the data is less important

than one could expect theoretically. One of the reasons for this is that with our *register-to-register* vector register, the software is liable to make fewer memory loads and stores than expected.

6.3 Performance *versus* Vector Register Depth

For the case where $p = r$, the number of cycles taken by the modular multiplication is inversely proportional to the depth p . This trend was already seen on the functional simulator. Figure 6.6 illustrates this trend (where we have a near-straight line) when working on data of 2048 bits.

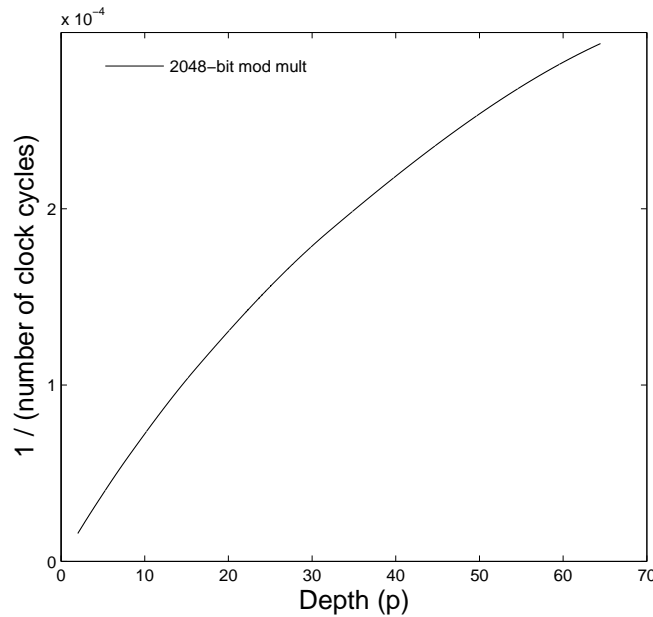


Figure 6.6: Performance *versus* depth on 2048-bit data

6.4 Performance *versus* Number of Lanes

Figure 6.7 illustrates a typical case where the performance varies by increasing the number of lanes. The illustrated example is for the case where $p = 32$ for 1024-bit data. Like the results obtained on the functional simulator, there seems to be some kind of inflection point for values of $r \lesssim \frac{p}{2}$ giving the hint that there may exist a critical point beyond which, for a given p , it is no longer interesting to increase r .

6.5 Area \times Performance *versus* Number of Lanes

The aim of having a scalable design is to find the best performance, power and area trade-off. The first way to perform such an analysis would be, for a given p , see how the

6. ANALYSIS OF VERILOG VEMICRY

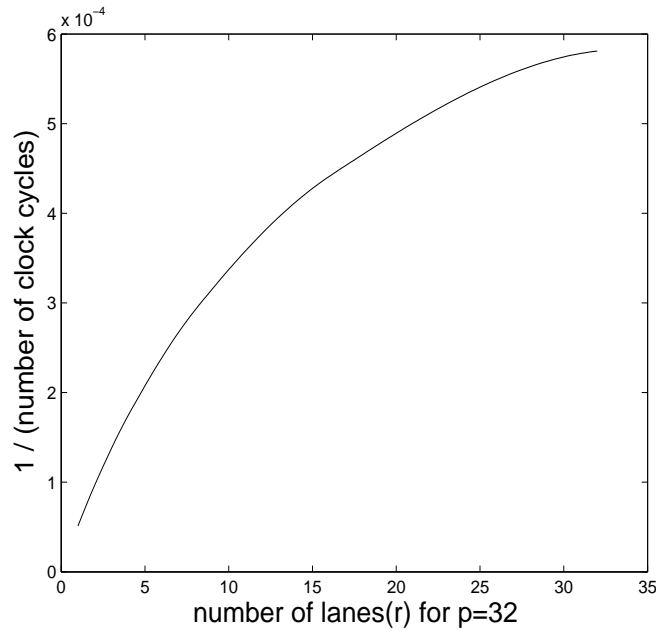


Figure 6.7: Performance *versus* number of lanes on 1024-bit data ($p = 32$)

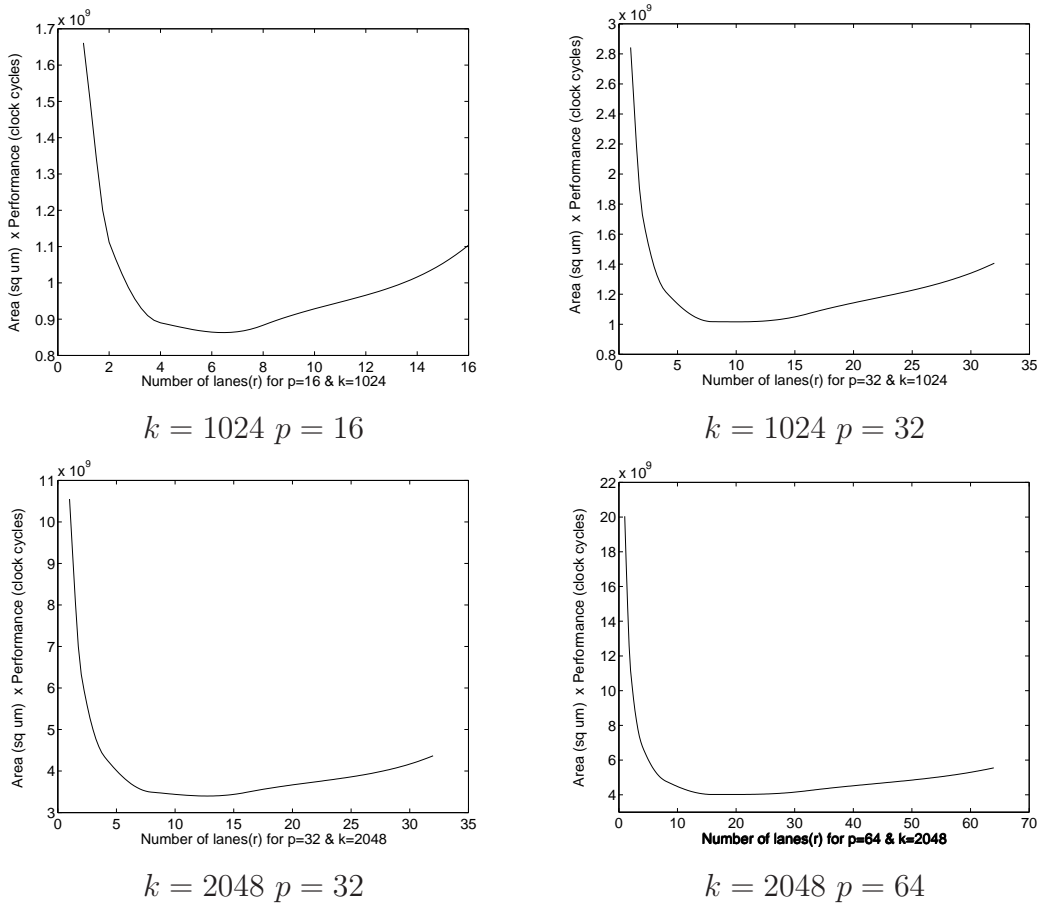
product $\text{Area} \times \text{Performance}$ varies by changing the number of lanes. Actually, we would target a smaller area and a small number of clock cycles. The curves in Figure 6.8 show that there is a minimum for $r \approx \frac{p}{4}$ for any data length.

6.6 Power Characteristics of VeMICry

Using Design CompilerTM, every time a version of the vector processor is synthesized (that is for different values of p and r), in addition to the netlist, the corresponding SDF is generated. An SDF is a Synthesis Design File into which timing and switching information are stored for each net of the targeted module. Then at simulation, the SDF file is used by PrimePowerTM to annotate the module whose power characteristics are measured (here it the `vemicry_module`). This provides a first order approximation for the power characteristics of the vector architecture (99.9% of nets and 98.9% of the leaf cells were annotated) during the simulation of the vector processor's netlist. The power measurements were done for calculations on the same set of 1024-bit data¹. Ideally we would perform the same manipulation using different sets of the 1024-bit data and perform an average but here our aim was to compare different architectures rather than obtain average power figures. We studied two aspects of the power profiles obtained:

- the **max power** which is the maximum instantaneous power reached by the module during the simulation. This characteristic is important because the maximum power allowed is a critical value for embedded processors.

¹Note that working with other data lengths only changed the duration of the calculation

Figure 6.8: Area \times Performance *versus* number of lanes

- the **mean power** which can then be correlated to heat dissipation of the circuit. In our experiments, for each power profile, the histogram of the different power values are plotted. From such an histogram, we ignore that values corresponding to the *static power*. The rest (the more significant *dynamic power*) followed some kind of normal distribution. The mean power is then determined as the value for which the maximum occurs for this distribution.

We looked at the relationship between vector register size and the power consumption. In Figure 6.9, measurements were taken for the case where we had only one lane ($r = 1$). There is a linear dependency with a gradient of about 1.25 mW per “register structure”. Here the “register structure” encompasses the 32-bit register and control logics associated with each register.

We also looked at the influence on the power of having more lanes (for a constant p). There is a linear dependency between the mean power and the number of lanes. From the data collected, this dependency is of the order of 4.3 mW/lane, bearing in mind that each lane has $\frac{p}{r}$ registers. Note that there is a similar linear relationship for the max power consumed.

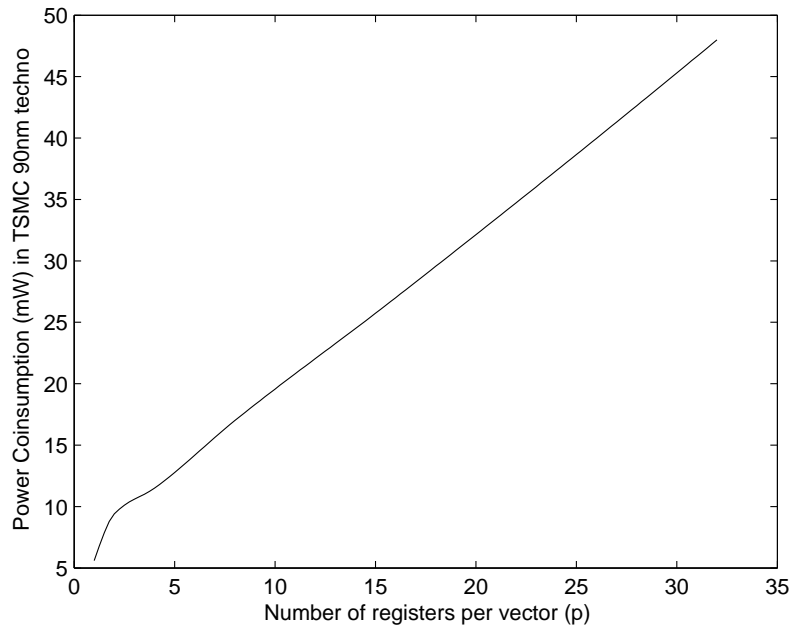


Figure 6.9: Mean power *versus* depth p for one lane

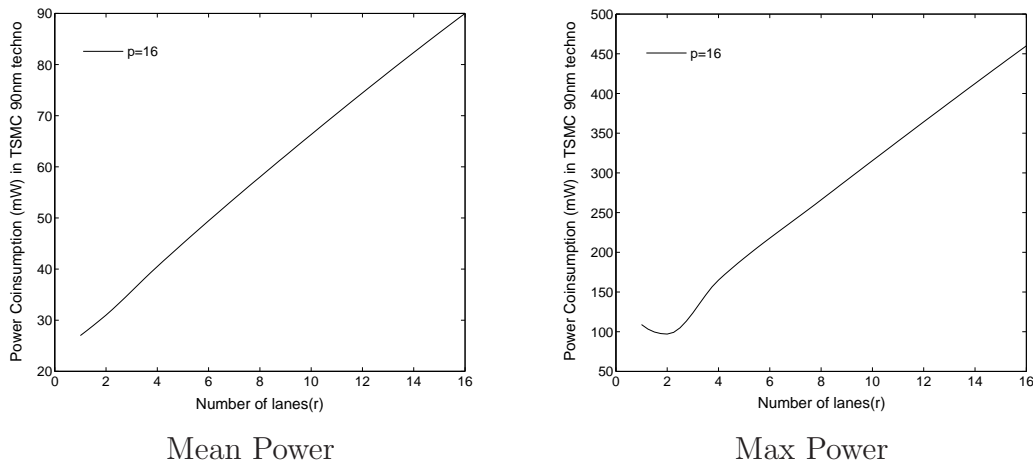
Finally we also looked at how the product of mean power \times Performance varies as a function of the number of lanes for a constant depth. As illustrated in Figure 6.11, there is a minimum for the case when $r \approx \frac{p}{4}$, which is not surprising given that the same conclusion was drawn for the area.

6.7 Security issues

In the design of the vector processor, there was no attempt to introduce any hardware countermeasures against side-channel or fault injection attacks. The “general purpose” nature of the vector co-processor allows a lot of flexibility in terms of implementation of software countermeasures like described in [Akkar (2004); Clavier & Joye (2001); Zambreno *et al.* (2004)]. In the following discussion, we focus on attacks based on power analysis, notably Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [Messerges *et al.* (1999)].

An SPA attack on algorithms like RSA or ECC consists in observing the current measured during an exponentiation algorithm and in trying to distinguish between a modular multiplication operation and a square operation. If such a distinction is possible, the attacker can infer information about the bits of the secret exponent used during a *signature*. In practice, the difference between a multiplication and a square can either be in terms of the:

- the timing taken by each of these operations.

Figure 6.10: Power *versus* number of lanes for $p = 16$

- the difference in amplitude or ‘shape’ of the power profile for each of these operations.

For the second feature, the difference could originate from the type of instruction(s) used or would depend, as it is on most architectures in practice, on the value or Hamming Weight of the data used. The latter aspect would also open the way to DPA attacks. In order to have a first idea about how such a vector processor would leak information, we looked more closely at how the simulated power profiles varied for some characteristic values for the two operands of the modular multiplication.

The code in Appendix C calculates $R = A \times B \times 2^{-k} \bmod F$ by parsing the words of B and multiplying each of these words to A . We hence wanted to investigate how differential power could be used to infer information about A or B . To do so, we did three power simulations on 512-bit values, for $p = 8$ and $r = 1$, where A had some random ‘constant’ value and B would take the following values given in hexadecimal.

```

B1 = 0x000000010000000100000001...000000010000000100000001
B2 = 0x0FFFFFFF000000010FFFFFFF...000000010FFFFFFF00000001
B3 = 0x000000010FFFFFFF00000001...0FFFFFFF000000010FFFFFFF

```

Note that for $B2$ and $B3$, we have alternate high Hamming Weight and low Hamming Weight 32-bit values. Since we are on a 32-bit machine, we wanted to see whether the ‘structure’ of the two values could be revealed by power analysis. We subtracted the curve for $B2$ from that of $B1$ and the curve for $B3$ from that of $B1$ to obtain the two curves illustrated in Figure 6.12. In the latter figure, the lower curve corresponds to “ $B2 - B1$ ” and the upper curve to “ $B3 - B1$ ”. Such an illustration shows that for each lane, the value or Hamming Weight of one of the operands does have a significant influence on the power signature.

6. ANALYSIS OF VERILOG VEMICRY

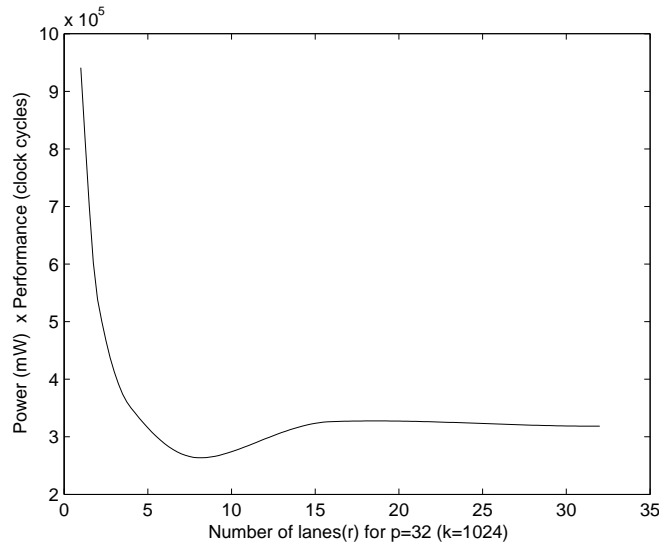


Figure 6.11: Performance \times Power *versus* number of lanes ($p = 32$)

In order to investigate the effect of parallelism on the power signature, it made more sense to vary this time the value of A because from our code it is the words of A that are distributed across the lanes. We studied two cases on 256-bit values for some constant ‘random’ value of B . In Case 1 $\{p = 8, r = 4\}$, we performed power simulations for $A1$ and $A2$ below and subtracted the second from the first. In Case 2 the experiment was repeated for $\{p = 8, r = 8\}$.

```
A1 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000001000000010000000100000001
A2 = 0x000000001000000010000000100000001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

In Case 1, since we have twice as many elements in each vector register as there are lanes, each ‘main loop’ (where the words of B are multiplied to A) is executed in two stages on each half of A_i . Hence for the cases of $A1$ and $A2$, because of the difference in Hamming Weight or value between the two halves, we expect the difference between the power curves for $A2$ and $A1$ to give significant signatures. This is confirmed in the upper curve of Figure 6.13. In Case 2, since we can work on all the elements of the vector registers (and hence of all the elements of A) in parallel, we would expect to have no significant difference between the two curves. This difference is shown in the lower curve of Figure 6.13. The trace in the lower curve takes less time than that of the upper curve, which is normal because there are twice as many lanes in parallel. For each ‘main loop’, there is at some point some significant difference but less spread over time within the loop. This illustrates that the parallelization has an effect on the power signature. But this also illustrates that there are other design artifacts that influence the power signature like, for example, the leakage during data fetches. Another explanation could come from the observation that, no matter the value of $r > 1$, the first lane (i.e. Lane 0) seems to consume slightly more power and have a longer critical path than the other lanes. Moreover, such power curves are to be considered with great caution because they

are gross approximations in terms of power and also because the scalar processor's power consumption is not taken into account.

The above manipulations show that, as expected, the vector co-processor does leak data dependent information. There are hints that show that, with the vectorization affect, this data dependant power signature is less spread over time. The leakage also depends on the software used and the way the data are manipulated, which could be considered as a positive thing because it provides a lot a flexibility in order to implement adequate countermeasures to counteract such attacks. For example, *message blinding* is a technique whereby the input message of a PK algorithm is randomized in software to decorrelate the input data from the power consumption measured: for RSA, the input message m is replaced by m' such that $m' = m + tn$ where n is the modulus and t is a small random value (could be on 32 bits) [Akkar (2004)]. The scalability of the design of the vector co-processor offers the flexibility to easily implement such a countermeasure. Say we want to design a vector co-processor optimised for 1024-bit RSA. The natural choice would be to have vector registers that can hold 1024 bits (i.e. a depth of $p = 32$). But if the software is going to implement *message blinding* we would have to work on 1056 bits and would rather have a vector register depth of $p = 33$ or $p = 34$ and r to be a corresponding factor. Our architecture was successfully tested for these parameters (for example for $p = 33$ and $r = 3$).

6. ANALYSIS OF VERILOG VEMICRY

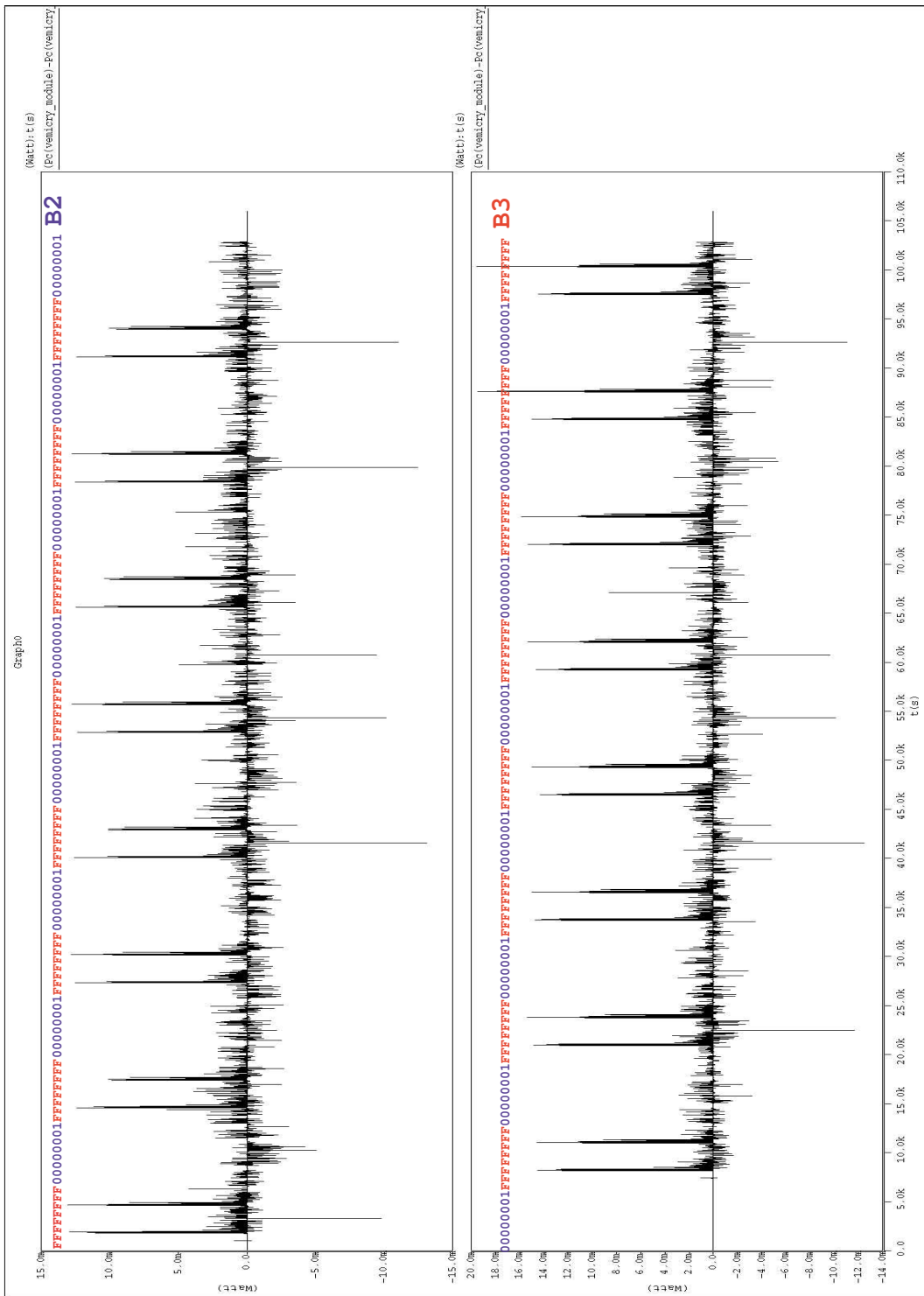


Figure 6.12: Difference Power for varying Hamming weights of B on 512 bits

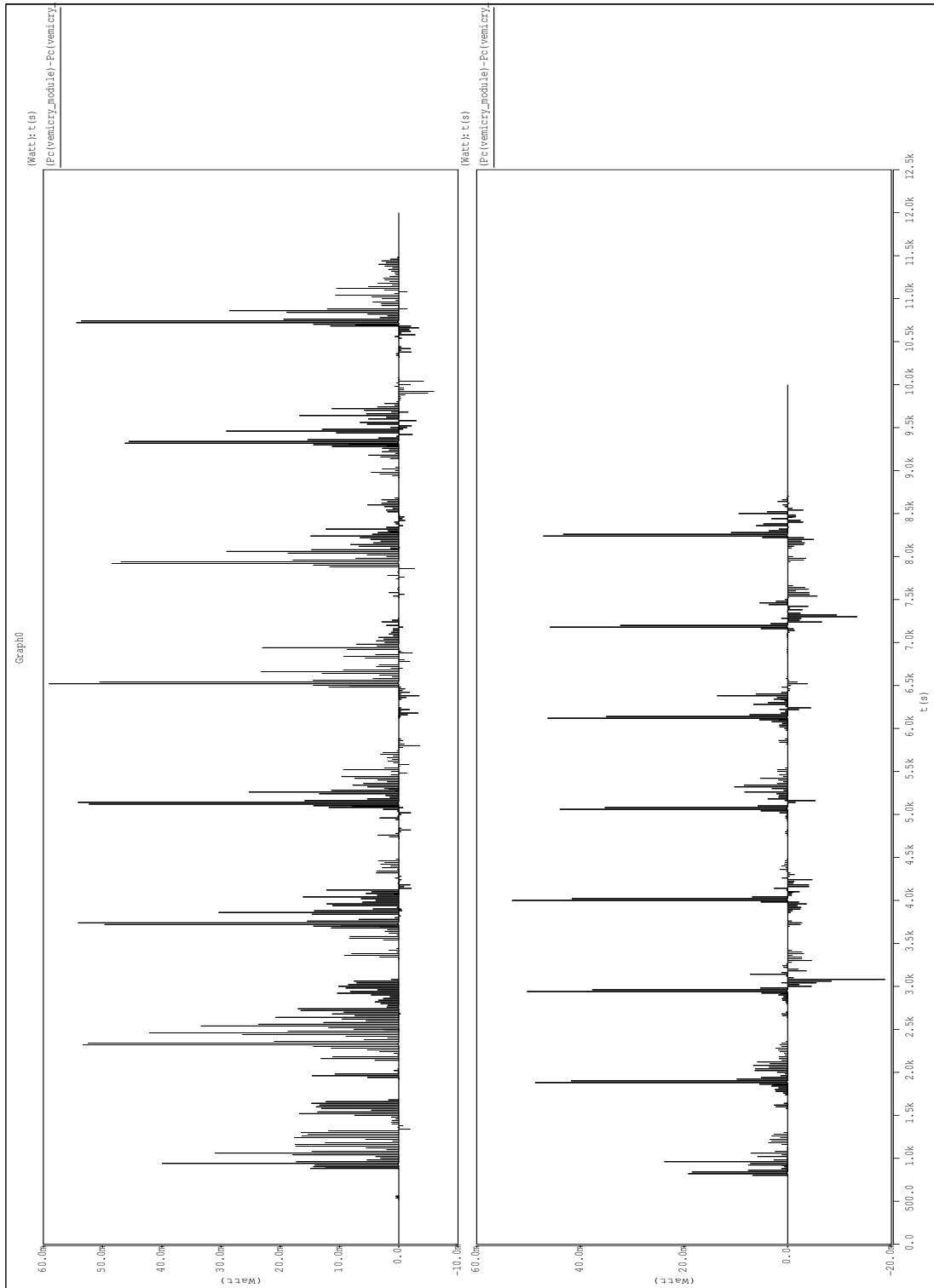


Figure 6.13: Power Traces for different Hamming Weight data on 256 bits

Chapter 7

Conclusion

In this research work, a new approach for the design of cryptographic accelerators is studied. Current solutions either involve the use of scalar general purpose processors which have limited compute power or include the implementation of dedicated hardware which lack flexibility in terms of algorithms implemented or the level of security. The main motivation is, therefore, to propose an architecture which can achieve high performance while providing the flexibility of software implementations of cryptographic algorithms and countermeasures against fault or side-channel attacks. The software could be upgraded either to adapt to new application requirements or to insert countermeasures against new attacks. Moreover, since I also identified the need to satisfy different applications with different constraints of performance, power or area on the chips being used, I wanted to give to the hardware designer the flexibility of resizing his chip (area and power) as a function of the targeted market requirements. As a result, by redefining the resources available in such a chip, the same architectural approach is suited to embedded devices (like smart-cards or mobile phones) as well to laptops, desktop computers or HSMs.

During the past decades, from the CRAY-I [Russell (1978)] up to the MMX graphics extensions [Peleg & Weiser (1996)], vector processing has been used to provide high performance through data parallelism. Vector processing is power efficient thanks to the simple instruction scheduling and low circuit complexity through little control logic overhead [Espasa *et al.* (1998)]. Compared to instruction parallel architectures where statistically nearly half of the power is spent in instruction decoding and scheduling [Folegnani & González (2001)], in vector processors most of the additional transistors end up in data storage and computing logic. Given the efficiency of vector processing, my research goal was to determine whether cryptography could be efficiently executed in a vector form. I have demonstrated that algorithms like AES and Montgomery-based modular multiplications (both in binary or prime fields) can be expressed in a vector form given a vector instruction set which I defined (Appendix A). The architecture of such a vector co-processor is determined by the number of lanes r , the depth of each vector register p and above all by how the elements of these vector registers are distributed across the lanes. With such a definition, scalability is achieved by changing the values of p and r . The instruction set that was defined has been decomposed into three groups (the GIVI, PIVI and MAVI) where each group has a characteristic decomposition across the pipeline stages of the vector co-processor.

7. CONCLUSION

An instruction set simulator, built using the ArchC [ArchC (2004)] tool, was built in order to test and validate cryptographic code implemented using the vector instructions that I identified. Such a functional simulator provides a first order approximation of the performance to be expected from such a vector machine assuming that the design has an instruction issue rate of 1 instruction per clock cycle. Initial results showed that the AES encryption (excluding the key schedule) can be implemented on the VeMICry (with $p = r = 8$) in 160 cycles which is a large gain in performance when compared to the 3283 cycles measured on a scalar processor for the same degree of software flexibility. For a modular multiplication on 192 bits on the same vector architecture, the cycle count is 95 which, again, compares favorably with the 3200 cycles for a purely scalar approach (and again for an equivalent degree of freedom in terms of software implementation). I also looked at how performance would be affected by varying the vector co-processor's parameters p , r and the size of the data. First, it could be seen that the impact on performance of working on larger data sizes is less important than theoretically expected and that this difference between the expected and measured variations gets larger as p gets bigger. Moreover I also saw that increasing r increases performance logarithmically for a given p .

A cycle-accurate synthesizable Verilog model of VeMICry was then implemented. The design is made up of a scalar 5-stage pipelined processor (MIPS-I compatible) and a vector 4-stage pipelined co-processor. In this model I focus mainly on the vector instructions for Public Key cryptography because on one side algorithms like AES are much faster in hardware (unless security is the primary concern, in which case a software approach would be more sensible) and on the other side PK algorithms are more interesting for a vector architecture because they involve large data sets and hence would require more data parallelism. The current design only implements the instructions necessary for binary field modular multiplication. Given the pipeline decomposition, the vector instructions for prime field modular multiplications would be taking the same number of clock cycles as for binary field multiplications.

The Verilog model of VeMICry is synthesized in TSMC's 90nm technology. The scalar part has an area of $0.085mm^2$ and an average power of $350\mu W$ during the execution of one of the vector modular multiplications. The theoretical maximum clock frequency for the scalar processor is 129MHz. For the vector part, the performance, area and power consumption figures depend on the values of p and r . The impact on performance of varying p and r , for different sizes of data, is the same as that already observed on the functional simulator. However, if we add the area and power factor, I clearly demonstrate that the best area/power/performance trade-off is reached for $r = \frac{p}{4}$. In order to appreciate how this approach compares to other cryptographic processors, I compiled Table 7.1. The latter compares the performance of different configurations of VeMICry with other commercially available hardware platforms for a 1024-bit modular multiplication. From this table it can be seen that, in terms of performance, the VeMICry architecture provides performance figures that are similar to what could be identified as the *best-in-class* solutions. Moreover we can note that the VeMICry, through its scalability, can address a vast panorama of markets. However, the Table also reveals that for some configurations

of the VeMICry (for $p = 8$ for example) the vector co-processor does not offer better performances than processors like the MIPS4KscTM or the SC200TM. This can be explained by the fact that such architectures have more powerful instructions like multiply and add instructions which can be executed in 1-2 clock-cycles on 32-bit words. Moreover, to better compare these commercial architectures with the VeMICry, one would need detailed area and power figures for each of them.

Finally, in terms of security, a quick look at power simulation curves of the VeMICry show that, as expected, there are data dependent power signatures, but there are hints that show that this signature depends on the structure of the software and the degree of parallelization. Software countermeasures would be easy to add.

TARGET	CO-PROCESSOR	SOURCE	CLOCK-CYCLES
Infineon SLE88	Crypto@1408	Infineon (2003)	1245
VeMICry	Vector (p=32, r=32)		1721
Infineon SLE44	“Sedlak” accelerator	Naccache & M’Raihi (1996)	2050
VeMICry	Vector (p=32, r=16)		2267
STM ST19	MAP accelerator	Handschuh & Paillier (2000)	2473
VeMICry	Vector (p=16, r=16)		2687
Infineon SLE66	ACE accelerator	Handschuh & Paillier (2000)	2864
VeMICry	Vector (p=32, r= 8)		3423
Power PC	Velocity Engine	Crandall & Klivington (1999)	3600
VeMICry	Vector (p=16, r= 8)		3714
VeMICry	Vector (p= 8, r= 8)		4619
VeMICry	Vector (p=16, r= 4)		5896
Philips P83	FAME co-processor	Naccache & M’Raihi (1996)	6510
VeMICry	Vector (p= 8, r= 4)		6608
MIPS4Ksc	none	MIPS (2005)	7812
VeMICry	Vector (p= 8, r= 2)		10842
ARM SC200	none	ARM (2002)	12760
TMS320C6201	DSP	Itoh <i>et al.</i> (1999)	14000

Table 7.1: Performance Comparison for 1024-bit modular multiplication

Possible Future Work

With the current Verilog implementation of the VeMICry, I have reached the main goals of this research work which were to study whether it was possible to design a vector crypto-accelerator, to show how it could be done and to illustrate how its scalability can help to make design choices based on power, area, performance to fit application constraints.

7. CONCLUSION

In terms of design of the VeMICry, a follow-up to this project would be to implement all of the vector instructions that have been defined in Appendix A and not decoded by the current Verilog model and then perform exhaustive tests both on the scalar and vector parts and in particular test all the possible configurations of interactions and synchronization between the scalar and the vector part. It would be interesting to see in what sense the area and power figures change once all the instructions have been implemented. All this could be done with the aim of designing a real chip and hence performing side-channel information leakage characterization.

In terms of research, now that the concept of vector processing for cryptography has been demonstrated, it would be interesting to investigate about how other cryptographic algorithms could be vectorised and investigate whether additional vector instructions would be beneficial. Another field of study could be to implement secure versions of the cryptographic algorithms and investigate the impact on performance of adding software countermeasures.

Appendix A

Vector Instructions

The VeMICry processor is composed of two families of instructions: the *scalar* instructions which correspond to the conventional MIPS-I instruction set and the *vector* instructions tailored to suit cryptographic requirements.

Suppose we have a vector processor having q vector registers. Each vector register is a vector of p words of 32 bits each. We also have a Vector Condition Register (VCR) which contains p bits and which is used for conditional vector instructions to show if the condition is applied to each of the individual words of the vector. Moreover, we have a second ‘scalar’ register called the Carry Register (CAR) which, for some instructions, ‘carry bits/words’ are written back. In the definitions below, most of the accesses to the vector registers are done with a stride of 1. The notion of stride is explained in [Flynn (1995)] and in Annexe A of [Hennessy & Patterson (2003)]. A stride of ‘1’ means that the words that are consecutively stored in the vector register are fetched by parsing the specified memory with a step of 1 word unit (here 4 bytes).

A. VECTOR INSTRUCTIONS

	V_i	i^{th} vector register
	R_j	j^{th} scalar register
	n	16-bit immediate value
VADDU	V_d, V_s, V_t	performs the unsigned addition between the i^{th} elements of V_s and V_t , writing the result as the i^{th} element of $V_d \forall 0 \leq i < p$. The carry is propagated and added to the $i + 1^{st}$ element of V_d . The carry from the addition of the corresponding p^{th} words is added to the content of CAR.
VBCROTR	V_d, V_s, n	The Vector-Bit-Conditional-Rotate-Right operates on each i^{th} word of V_s . If $VCR[i]$ is 1, then $V_s[i]$ is rotated by n bits to the right and the result is written to $V_d[i]$. If $VCR[i]$ is 0, then $V_s[i]$ is copied to $V_d[i]$ without transformation.
VBYTELD	V_d, R_s, n	each word of V_d is treated as four bytes. Each byte is an offset which is added to the address stored in R_s and the byte stored at that address is read from the VPU's corresponding memory. The read byte is written to the same location as that of its original corresponding byte. This process is executed for n words of V_d .
VEEXTRACT	R_d, V_s, n	copies the value of the $V_s[n - 1]$ into R_d . If $n = 0$, then it is CAR which is written to scalar register.
VLOAD	V_d, R_s, n	loads in V_d the p consecutive 32-bit words from memory starting from address stored in R_s with a stride of 1. n is not used.
VMPMUL	V_d, V_s	The Vector Modular Polynomial Multiplication treats each i^{th} word of V_s as four bytes: each byte is a polynomial in $GF(2^8)$ which is multiplied by x modular the polynomial represented in the 9 least significant bits in scalar register VCR. The result is written to V_d .
VSADDU	V_d, V_s, R_t	Vector-Scalar-Addition does the unsigned arithmetic addition of value in R_t to every i^{th} word of V_s and writes the result to V_d . The carry is not propagated but is instead written as the i^{th} bit of the register CAR.
VSAMULT	V_d, V_s, R_t	Vector-Scalar-Arithmetic-Multiplication: multiplies R_t by $V_s[p] V_s[p - 1] \dots V_s[0]$ with carry propagation and result is written to V_d . The most significant carry bits are written to register CAR.

VTRANSP	V_d, V_s, n	copies vector in V_s to register V_d . If n is zero, there is a direct copy without transposition. If n is non-zero, V_s is viewed as a $4 \times p$ matrix which is transposed and written to vector register V_d with a stride of n .
VSMOVE	V_d, R_s, n	copies the value in register R_s to the first n words of V_d . If n is zero, then R_k is copied to CAR. If $n \geq p$ then R_s is copied to every word of V_d .
VSPMULT	V_d, V_s, R_t	Vector-Scalar-Polynomial-Multiplication: does the polynomial multiplication of R_t by $V_s[p] V_s[p-1] \dots V_s[0]$ and the result is written to V_d . The previous value of CAR is XORed to the result in $V_d[0]$. The most significant $p + 1^{st}$ word is written to the register CAR.
VSTORE	R_d, V_s, n	stores the p consecutive 32-bit words from register V_s to memory starting from address stored in R_d with a stride of 1. n is not used.
VWSHL	V_d, V_s, n	Vector-Word-Shift-Left shifts the contents of vector V_s by 1 position to the left inserting zeros to the right. The resulting vector is written to V_d and the outgoing word to CAR. n is not used.
VWSHR	V_d, V_s, n	Vector-Word-Shift-Right shifts the contents of vector V_s by 1 word position to the right inserting the data stored in CAR to the left. The resulting vector is written to V_d . n is not used.
VXOR	V_d, V_s, V_t	XORs corresponding words between V_s and V_t and stores the result in V_d .
MTVCR	R_s	Copies R_s to VCR.
MFVCR	R_d	Copies the value in VCR to the scalar register R_d .

A. VECTOR INSTRUCTIONS

Appendix B

Vector Codes

B.1 AES vector code

```
/* Begin ShiftRows */
vload  $f0, $4, (4*2)
vtransp $f2, $f0, (4*2)
addiu  $11, $0, 0x00EE
mtvcr  $11
vbcrotr $f4, $f2, 24
addiu  $11, $0, 0x00CC
mtvcr  $11
vbcrotr $f2, $f4, 24
addiu  $11, $0, 0x0088
mtvcr  $11
vbcrotr $f4, $f2, 24
vtransp $f0, $f4, (4*2)

/* Begin MixColumns */
addiu  $11, $0, 0xFFFF
mtvcr  $11
vbcrotr $f2, $f0, 8
vbcrotr $f4, $f0, 16
vbcrotr $f6, $f0, 24
vxor   $f8, $f0, $f6
addiu  $11, $0, 0x011B
mtvcr  $11
vmpmul $f10, $f8
vxor   $f0, $f10, $f2
vxor   $f8, $f0, $f4
vxor   $f0, $f8, $f6

/* Add Round Key */
vload  $f2, $5, 4
vwshl  $f4, $f2, 4
vload  $f4, $5, 4
vxor   $f2, $f0, $f4
```

B. VECTOR CODES

```
vstore $4, $f2, (4*2)
```

B.2 ECC vector code

```
.global CalcN0
.ent    CalcN0
```

CalcN0:

```
addi    $9, $0, 1
vsmove  $f0, $4, 1
move    $8, $4
```

loop_calcn:

```
move    $2, $8
vspmult $f0, $f0, $4
vextract $8, $f0, 1
bne     $8, $9, loop_calcn
nop
j       $31
nop
.end    CalcN0
```

```
.global MultBinPoly
.ent    MultBinPoly
```

MultBinPoly:

```
lw      $24, 16($29)      # loading size of data
lw      $2, 20($29)       # loading the N0
vload   $v0, $5, 5       # v0 <= b(x) on 6 words
vload   $v1, $6, 5       # v1 <= f(x) on 6 words
vsmove  $v3, $0, 8       # v3 cleared; (v3 == c(x))
addiu   $15, $0, 0       # initialization for loop
sll     $24, $24, 2      # $24 <= 4M
```

LoopBin:

```
add     $8, $15, $4      # addr. of j-th word of a(x)
lw      $8, 0($8)        # j-th word of a(x)
vspmult $v5, $v0, $8     # v5 <= a[j] * v0; (v0 == b(x))
vxor    $v3, $v5, $v3    # v3 <= v5 + v3
vextract $9, $v3, 1      # $9 <= C_0
vsmove  $v2, $9, 1       # v2[0] <= $9 ($9 == C_0)
vspmult $v4, $v2, $2     # v4 <= N0 * v2
vextract $9, $v4, 1      # $9 <= M(x)
vspmult $v5, $v1, $9     # v5 <= v4[0] * v1; (v1 == f(x))
vxor    $v3, $v3, $v5    # v3 <= v3 + v5
vwshr   $v3, $v3, 1      # v3 shifted right by 1 word
addi    $15, $15, 4      # Increase index by 4
bne     $15, $24, LoopBin
```

```

nop
vstore    $7, $v3, 5
j         $31
nop
.end      MultBinPoly

```

B.3 RSA vector code

```

.global Mongo
.ent      Mongo
Mongo:
    lw     $24, 16($29)      # loading value of M
    lw     $2, 20($29)      # loading the value of r
    sll   $24, $24, 2       # $24 <= 4*M
    vsmove $v0, $0, 0       # v0 all cleared
    addiu  $15, $0, 0

LoopClear:
    add    $8, $7, $15      # [1] $8 <= @R + offset
    vstore $8, $v0, 7       # [1] at addr. $8,clear data
    addi   $15, $15, 128    # [1] $15 <= $15 + (p*4)
    bne   $15, $24, LoopClear
    nop
    add    $8, $7, $15      # [1] Clearing one more word in R
    sw     $0, 0($8)
    addiu  $15, $0, 0       # [2] Initializing j to 0

LoopB:
    add    $8, $5, $15      # [3] $8 <= addr of Aj
    lw     $8, 0($8)        # [3] $8 <= Aj
    vsmove $v3, $0, 0       # [3] Clearing v3 (carry)
    addiu  $9, $0, 0        # [3] Initializing loop counter i

LoopMult:
    add    $10, $9, $4       # [3] $10 <= addr of Bi
    vload  $v0, $10, 31     # [3] v0 <= Bi (p=32)
    vsamult $v1, $v0, $8    # [3] v1 <= Aj * Bi
    vaddu  $v0, $v1, $v3    # [3] v0 <= v1 + v3 (v3 has carry)
    add    $10, $9, $7       # [3] $10 <= addr of Ri
    vload  $v1, $10, 31     # [3] v1 <= Ri (p=32)
    vaddu  $v0, $v1, $v0    # [3] v0 <= Ri + Aj * Bi
    vstore $10, $v0, 7      # [3] Ri written back to memory
    vextract $25, $v0, 0    # [3] $25 <= CAR (i.e. carry)
    vsmove $v3, $25, 1     # [3] v3 <= carry
    addi   $9, $9, 128      # [3] $9 <= $9 + (p*4)
    bne   $9, $24, LoopMult
    nop
    add    $10, $9, $7       # [3] R has one more word
    sw     $25, 0($10)      # [3] Storing one more word of R

```

B. VECTOR CODES

```
lw      $8, 0($7)          # [4] $8 <= Ro
mult    $8, $2             # [4] (HI,LO) <= r * Ro
mflo    $8                 # [4] J <= (r * Ro) mod 2^32
vsmove  $v3, $0, 0        # [5] Clearing v3 (carry)
addiu   $9, $0, 0         # [5] Initializing loop counter i

LoopReduc:
add     $10, $9, $6       # [5] $10 <= addr of Ni
vload   $v0, $10, 7       # [5] v0 <= Ni (p=32)
vsamult $v1, $v0, $8      # [5] v1 <= J * Ni
vaddu   $v0, $v1, $v3     # [5] v0 <= v1 + v3 (v3 has carry)
add     $10, $9, $7       # [5] $10 <= addr of Ri
vload   $v1, $10, 7       # [5] v1 <= Ri (p=32)
vaddu   $v0, $v1, $v0     # [5] v0 <= Ri + J * Ni
addi    $10, $10, -4      # [6] implicit right shift
vstore  $10, $v0, 7       # [5] Ri written to memory
vextract $25, $v0, 0      # [5] $25 <= CAR (carry)
vsmove  $v3, $25, 1      # [5] v3 <= carry
addi    $9, $9, 128       # [5] $9 <= $9 + (p*4)
bne     $9, $24, LoopReduc
nop

add     $10, $9, $7       # [5] R has one more word
lw      $11, 0($10)       # [5] $11 <= R(M-1)
addu    $25, $25, $11     # [5] $25 <= carry + R[M-1]
addi    $10, $10, -4
sw      $25, 0($10)       # [5] R[M-1] in memory
addi    $15, $15, 4       # [2] j = j+1
bne     $15, $24, LoopB   # [7] End of loop
nop
j       $31
nop
.end    Mongo
```

Appendix C

VeMICry Test Code

```
@008
8f b8 00 00 // lw $24,0($29) (loading value of M)
8F a2 00 04 // lw $2,4($29) (loading precalculated H0)
00 18 c0 80 // sll $24, $24, 2
68 00 01 00 // vsmove $f0,$0,64
27 a4 00 08 // addiu $4, $29, 8 (A)
00 98 28 21 // addu $5, $4, $24 (B)
00 b8 30 21 // addu $6, $5, $24 (F)
00 d8 50 21 // addu $10, $6, $24 (Res minus 1 word)
25 47 00 04 // addiu $7, $10, 4 (Res)
24 0f 00 00 // addiu $15, $0, 0
<LoopClear>
00 ef 40 20 // add $8, $7, $15 <LoopClear>
21 ef 00 20 // addi $15, $15, 32 (p*4)
f4 08 00 20 // vstore $8, $f0, 32
15 f8 ff fc // bne $15, $24, LoopClear
00 00 00 00 // nop

24 0f 00 00 // addiu $15, $0, 0
<LoopB>
00 af 40 20 // add $8, $5, $15 <LoopB>
8d 08 00 00 // lw $8, 0($8) (Loading Bi)
8c 89 00 00 // lw $9, 0($4) (Loading A0)
8c ec 00 00 // lw $12, 0($7) (Loading R0)
68 00 00 00 // vsmove $f0, $0, 0
01 09 00 0E // multp $8, $9
00 00 50 12 // mflo $10
01 4c 50 26 // xor $10, $10, $12
24 09 00 00 // addiu $9, $0, 0
01 42 00 0e // multp $10, $2
24 19 00 00 // addiu $25, $0, 0
24 0b 00 00 // addiu $11, $0, 0
00 00 70 12 // mflo $14
68 06 01 00 // vsmove $f6, $0, 256 (p)
00 00 00 00
```

C. VEMICRY TEST CODE

```
<LoopMult>
01 24 50 20    // add  $10, $9, $4 <LoopMult>
00 00 00 00
f9 40 00 20    // vload  $f0, $10, 32
00 00 00 00
fc 08 10 08    // vspmult $f2, $f0, $8
69 66 00 01    // vsmove $f6, $11, 1
01 27 50 20    // add $10, $9, $7
fc 46 00 01    // vxor  $f0, $f2, $f6
f9 42 00 20    // vload $f2, $10, 32
01 26 60 20    // add $12, $9, $6
fc 40 20 01    // vxor $f4, $f2, $f0
6c 8b 00 00    // vextract $11, $f4, 0
00 00 00 00
68 00 00 00    // vsmove $f0, $0, 0
f9 80 00 20    // vload  $f0, $12, 32
6b 26 00 01    // vsmove $f6, $25, 1
fc 0e 10 08    // vspmult $f2, $f0, $14
00 00 00 00
fc 46 00 01    // vxor  $f0, $f2, $f6
6c 59 00 00    // vextract $25, $f2, 0
00 00 00 00
fc 80 00 01    // vxor $f0, $f4, $f0
21 4a ff fc    // addi  $10, $10, -4 (implicit right shift)
21 29 00 20    // addi $9, $9, 32 (p*4)
68 02 00 00    // vsmove $f2, $0, 0
f4 0a 00 20    // vstore $10, $f0, 32
15 38 ff e5    // bne  $9, $24, LoopMult
00 00 00 00    // nop

01 27 50 20    // add  $10, $9, $7
03 2b c8 26    // xor $25, $25, $11
21 4a ff fc    // addi $10, $10, -4
ad 59 00 00    // sw $25, 0($10)
21 ef 00 04    // addi  $15, $15, 4
15 f8 ff cf    // bne $15, $24, LoopB
00 00 00 00

00 00 00 00    // nop
00 00 00 00    // nop
00 00 00 00
00 00 00 00
00 00 00 0D
```

Appendix D

VeMICry Verilog

D.1 Scalar Processor Verilog

```
/** Instruction Set Definition in "instruction_set.v" **/  
  
/** TOP LEVEL MODULE **/  
  
module scamips (clock, reset, inst_read, inst_add , IF_INST,  
               debugw, typ, access, data_read, data_write, data_add,  
               data_in_mem, memdataout, rs_add, rt_add, rd_add, shamt, op, func, imm16,  
               vindex, rs_val, rt_val, pipe_stall, sbi_sig, sbi_add, sbi_val, eID_VEX_RT_ADD);  
  
    // Parameters  
    parameter period = 20 ;  
    parameter pWIDTH = 32;           // data path width  
    parameter pNUMREG = 32;         // number of scalar registers  
    parameter vq = 4;  
    parameter vp = 1;  
    parameter vr = 1;  
    parameter vp_over_vr_width = 8;  
    parameter vp_over_vr_minus1 = (vp/vr)-1;  
  
    wire [vp_over_vr_width-1:0] v_o_v_m1;  
  
    // Register Bank  
    reg [pWIDTH-1:0] SCAREG[0:pNUMREG-1];  
    reg [pWIDTH-1:0] NPC;  
    reg [pWIDTH-1:0] PC;  
    reg [pWIDTH-1:0] LO;  
    reg [pWIDTH-1:0] HI;  
  
    // IF Internal Registers  
    reg [pWIDTH-1:0] IF_EX_RES;  
  
    // IF-ID connections  
    reg [pWIDTH-1:0] IF_ID_INST;  
  
    // ID Internal Registers  
    reg [pWIDTH-1:0] ID_INST;  
    reg ID_JMP_STALL;  
    reg [pWIDTH-1:0] ID_RS_VAL;  
    reg [pWIDTH-1:0] ID_RT_VAL;  
    reg [pWIDTH-1:0] ID_VRS_VAL;  
    reg [pWIDTH-1:0] ID_VRT_VAL;  
    reg [4:0] ID_RS_ADD;  
    reg [4:0] ID_RT_ADD;  
    reg [4:0] ID_RD_ADD;  
    reg [4:0] ID_SHAMT;  
    reg [5:0] ID_OP;  
    reg [5:0] ID_FUNC;
```

D. VEMICRY VERILOG

```
reg [15:0]          ID_IMM16;
reg [25:0]          ID_ADDR;
reg                ID_MEMO_WRT;
reg                ID_MEMO_RD;
reg                ID_RT;
reg                ID_STALL;
reg [7:0]           vstall_index;
reg [7:0]           ID_VSTALL_INDEX;
reg                VSTALL;
wire               ID_VSTALL;
reg [vp_over_vr_width-1:0] ID_VDF_INDEX;
wire [vp_over_vr_width-1:0] ID_INDEX;
reg                id_is_vector;

// ID-EX connections
reg [pWIDTH-1:0]   ID_EX_RS_VAL;
reg [pWIDTH-1:0]   ID_EX_RT_VAL;
reg [4:0]           ID_EX_RS_ADD;
reg [4:0]           ID_EX_RT_ADD;
reg [4:0]           ID_EX_RD_ADD;
reg [4:0]           ID_EX_SHAMT;
reg [5:0]           ID_EX_OP;
reg [5:0]           ID_EX_FUNC;
reg [5:0]           ID_VEX_OP;
reg [5:0]           ID_VEX_FUNC;
reg [15:0]          ID_EX_IMM16;
reg [25:0]          ID_EX_ADDR;
reg                ID_EX_MEMO_WRT;
reg                ID_EX_MEMO_RD;
reg                ID_EX_REG_WRT;
reg [pWIDTH-1:0]   ID_VEX_RS_VAL;
reg [pWIDTH-1:0]   ID_VEX_RT_VAL;
reg [4:0]           ID_VEX_RS_ADD;
reg [4:0]           ID_VEX_RT_ADD;
reg [4:0]           ID_VEX_RD_ADD;
reg [4:0]           ID_VEX_SHAMT;
reg [15:0]          ID_VEX_IMM16;

// EX Internal Registers
wire [pWIDTH-1:0]  EX_RT_VAL;
wire [pWIDTH-1:0]  EX_RS_VAL;
wire               EX_REG_WRT;
reg               EX_BR_TAKEN;
wire               EX_JMP_TAKEN;
wire               EX_LD_STALL;
reg [(2*pWIDTH)-1:0] EX_MULT;
reg [pWIDTH-1:0]   EX_RD_VAL;
wire [pWIDTH-1:0]  EX_IMM16;
wire [4:0]          EX_RD_ADD;
wire [5:0]          EX_OP;
wire [5:0]          EX_FUNC;
wire               EX_MEMO_WRT;
wire               EX_MEMO_RD;
reg [3:0]           EX_ALU_OTP;
wire [15:0]         EX_CONST;
reg [1:0]           EX_ACCESS;
wire [4:0]          EX_SHAMT;
wire [4:0]          EX_RT_ADD;
wire [25:0]         EX_ADDR;

// EX-MEM connections
reg [pWIDTH-1:0]   EX_MEM_RES;
reg [4:0]           EX_MEM_RD_ADD;
reg               EX_MEM_REG_WRT;
reg               EX_MEM_MEMO_WRT;
reg               EX_MEM_MEMO_RD;
reg [5:0]          EX_MEM_OP;
reg [5:0]          EX_MEM_FUNC;
reg [pWIDTH-1:0]   EX_MEM_RT_VAL;
reg               EX_MEM_LD_STALL;
```

```

reg [1:0]          EX_MEM_ACCESS;

// MEM Internal Registers
reg [pWIDTH-1:0]  MEM_RES;
reg [1:0]         MEM_ACCESS;

// MEM-WB connections
reg [pWIDTH-1:0]  MEM_WB_RES;
reg               MEM_WB_REG_WRT;
reg [4:0]         MEM_WB_RD_ADD;
reg               MEM_WB_LD_STALL;
reg [pWIDTH-1:0]  MEM_RES_ADD;
reg               MEM_MEMO_RD;
reg               MEM_MEMO_WRT;

// Control Signals
wire              BR_TAKEN;
reg               JMP_STALL;
reg               LD_STALL;
wire              IS_LD_STALL;
wire              debug;

reg               pipe_stall_int;
reg [pWIDTH-1:0]  sbi_val_int;

// Module IOs
input             clock;
input             reset;
input [pWIDTH-1:0] IF_INST;
output reg [pWIDTH-1:0] inst_add;
output           inst_read;
input [pWIDTH-1:0] memdataout;
output           debugw;
output           typ;
output [1:0]     access;
output           data_read;
output           data_write;
output [pWIDTH-1:0] data_add;
output [pWIDTH-1:0] data_in_mem;
input           pipe_stall, sbi_sig;
input [4:0]     sbi_add;
input [pWIDTH-1:0] sbi_val;
output [4:0]    rs_add, rt_add, rd_add, shamt;
output [5:0]    op, func;
output [15:0]   imm16;
output [pWIDTH-1:0] rs_val, rt_val;
output [vp_over_vr_width-1:0] vindex;

assign v_o_v_m1 = vp_over_vr_minus1[vp_over_vr_width-1:0];

/** Control Signals */
assign IS_LD_STALL = ((!reset)&&(ID_EX_MEMO_RD && ((IF_ID_INST[25:21] == ID_EX_RD_ADD) ||
((IF_ID_INST[31:26] == 'nop) && (IF_ID_INST[20:16] == ID_EX_RD_ADD)))) &&
(IF_ID_INST[31:26] != 'j) && (IF_ID_INST[31:26] != 'jal)) ?
1'b1 : 1'b0;

assign BR_TAKEN = (reset) ? 1'b0 :
(EX_BR_TAKEN||EX_JMP_TAKEN);

always @(posedge clock or posedge reset)
  if (reset)
    pipe_stall_int <= 1'b0;
  else
    pipe_stall_int <= pipe_stall;

always @(posedge sbi_sig or posedge reset)
  if (reset)
    sbi_val_int <= 0;

```

D. VEMICRY VERILOG

```
    else
        sbi_val_int <= sbi_val;

always @(posedge clock or posedge reset)
    if (reset) begin
        JMP_STALL <= 1'b0;
        LD_STALL <= 1'b0;
    end
    else begin
        JMP_STALL <= ID_JMP_STALL;
        LD_STALL <= IS_LD_STALL;
    end

/** Instruction Fetch */

assign inst_read = (reset) ? 1'b0 :
    ((JMP_STALL) ? 1'b0 : 1'b1);

always @*
    if (reset)
        inst_add <= 0;
    else if (LD_STALL||VSTALL||pipe_stall_int)
        inst_add <= PC-4;
    else if (!JMP_STALL)
        if (BR_TAKEN)
            inst_add <= EX_MEM_RES;
        else
            inst_add <= PC;

always @*
    if (LD_STALL||VSTALL||pipe_stall_int)
        NPC <= PC;
    else if (!JMP_STALL)
        if (BR_TAKEN)
            NPC <= EX_MEM_RES+4;
        else
            NPC <= PC+4;

always @(posedge clock or posedge reset) begin
    if (reset) begin
        PC <= 0;
        IF_ID_INST <= 0;
    end
    else begin
        if (LD_STALL||(VSTALL==1)||(pipe_stall_int==1))
            PC <= PC;
        else if (JMP_STALL) PC <= NPC;
        else if (BR_TAKEN) PC <= EX_MEM_RES+4;
        else PC <= PC+4;
        IF_ID_INST <= IF_INST;
    end // else: !if(reset)
end // end always for IF

// To stop simulation //

assign debug = (IF_ID_INST=='h0D) ? 1'b1 : debug;

/** Instruction Decode */

always @*
    if (reset)
        ID_INST <= 0;
    else if (!(VSTALL||LD_STALL||pipe_stall_int))
        ID_INST <= IF_ID_INST;

always @*
    if (reset)
        id_is_vector <= 1'b0;
    else if (!(VSTALL||LD_STALL))
        id_is_vector <= (ID_INST[31:26]=='vr1) || (ID_INST[31:26]=='vload) ||
```

```

                (ID_INST[31:26]==‘vstore) ||
                (ID_INST[31:26]==‘vextract) || (ID_INST[31:26]==‘vwshl) ||
                (ID_INST[31:26]==‘vwshr) || (ID_INST[31:26]==‘vsmove);

always @*
  if (reset)
    vstall_index <= 8’b0;
  else if (VSTALL||LD_STALL) begin
    if (ID_VSTALL_INDEX!=0)
      vstall_index <= ID_VSTALL_INDEX-8’b1;
    end
  else if (id_is_vector)
    vstall_index <= v_o_v_m1;
  else
    vstall_index <= 8’b0;

always @*
  if (reset)
    ID_JMP_STALL <= 1’b0;
  else if (!(VSTALL||LD_STALL))
    ID_JMP_STALL <= (ID_INST[31:26] == ‘j) ||(ID_INST[31:26] == ‘jal) ||
      ((ID_INST[31:26] == ‘nop) && (ID_INST[5:0] == ‘jr)) ||
      ((ID_INST[31:26] == ‘nop) && (ID_INST[5:0] == ‘jalr)) ||
      (ID_INST[31:26] == ‘beq) || (ID_INST[31:26] == ‘bne) ||
      (ID_INST[31:26] == ‘blez) || (ID_INST[31:26] == ‘bltz) ||
      (ID_INST[31:26] == ‘bgtz);

always @*
  if (reset)
    {ID_OP,ID_RS_ADD,ID_RT_ADD,ID_IMM16} <= 0;
  else if (!(VSTALL||LD_STALL))
    {ID_OP,ID_RS_ADD,ID_RT_ADD,ID_IMM16} <= ID_INST;

always @*
  if (reset)
    {ID_SHAMT,ID_FUNC} <= 0;
  else if (!(VSTALL||LD_STALL))
    {ID_SHAMT,ID_FUNC} <= ID_INST[10:0];

assign ID_VSTALL = ((!reset)&&(vstall_index>0)) ? 1’b1 :
  1’b0;
assign ID_INDEX = v_o_v_m1 - vstall_index;

always @*
  if (!(VSTALL)||LD_STALL))
    ID_ADDR <= ID_INST[25:0];

always @*
  if (reset)
    ID_RT <= 1’b0;
  else if (!(VSTALL||LD_STALL))
    ID_RT <= (ID_OP == ‘bne) || (ID_OP == ‘beq) || (ID_OP == ‘lwl) || (ID_OP == ‘lwr);

always @*
  if (!(VSTALL||LD_STALL))
    ID_MEMO_RD <= (ID_OP == ‘lb) | (ID_OP == ‘lh) | (ID_OP == ‘lwl) | (ID_OP == ‘lw) |
      (ID_OP == ‘lbu) | (ID_OP == ‘lhu) | (ID_OP == ‘lwr);

always @*
  if (reset)
    ID_MEMO_WRT <= 1’b0;
  else if (!(VSTALL||LD_STALL))
    ID_MEMO_WRT <= (ID_OP == ‘sb) | (ID_OP == ‘sh) | (ID_OP == ‘swl) | (ID_OP == ‘sw) | (ID_OP == ‘swr);

always @*
  if (reset)
    ID_RD_ADD <= 5’b0;
  else if (!(VSTALL||LD_STALL))
    if ((ID_OP == ‘nop) || (ID_OP == ‘vr1))
      ID_RD_ADD <= ID_INST[15:11];

```

D. VEMICRY VERILOG

```
    else
        ID_RD_ADD <= ID_INST[20:16];

always @*
    if (reset)
        ID_RT_VAL <= 0;
    else if (!(VSTALL||LD_STALL) && ((ID_OP == 'nop) || (ID_OP == 'vri) || ID_RT || ID_MEMO_WRT ||
        (ID_OP == 'vstore)))
        if ((ID_RT_ADD == EX_MEM_RD_ADD) && (EX_MEM_REG_WRT == 1))
            ID_RT_VAL <= EX_MEM_RES;
        else if ((ID_RT_ADD == MEM_WB_RD_ADD) && (MEM_WB_REG_WRT == 1))
            ID_RT_VAL <= MEM_WB_RES;
        else
            ID_RT_VAL <= SCAREG[ID_INST[20:16]];

always @*
    if (reset)
        ID_RS_VAL <= 0;
    else if (!(VSTALL||LD_STALL))
        if ((ID_RS_ADD == EX_MEM_RD_ADD) && (EX_MEM_REG_WRT == 1))
            ID_RS_VAL <= EX_MEM_RES;
        else if ((ID_RS_ADD == MEM_WB_RD_ADD) && (MEM_WB_REG_WRT == 1))
            ID_RS_VAL <= MEM_WB_RES;
        else
            ID_RS_VAL <= SCAREG[ID_INST[25:21]];

always @*
    if (reset)
        ID_VRS_VAL <= 0;
    else if (!(VSTALL||LD_STALL))
        if (ID_OP=='vload')
            ID_VRS_VAL <= ID_RS_VAL;
        else if (ID_OP == 'vstore')
            ID_VRS_VAL <= ID_RT_VAL;
        else
            ID_VRS_VAL <= 32'h0;

always @*
    if (reset)
        ID_VRT_VAL <= 0;
    else if (!(VSTALL||LD_STALL))
        if ((ID_OP=='vri)&&(ID_FUNC=='vspmult_func))
            ID_VRT_VAL <= ID_RT_VAL;
        else if (ID_OP == 'vsmove')
            ID_VRT_VAL <= ID_RS_VAL;

always @(posedge clock or posedge reset) begin
    if (reset) begin
        ID_EX_OP      <= 0;
        ID_EX_FUNC    <= 0;
        ID_EX_RS_VAL  <= 0;
        ID_EX_RT_VAL  <= 0;
        ID_EX_RS_ADD  <= 0;
        ID_EX_RT_ADD  <= 0;
        ID_EX_RD_ADD  <= 0;
        ID_EX_SHAMT   <= 0;
        ID_EX_IMM16   <= 0;
        ID_EX_ADDR    <= 0;
        ID_EX_MEMO_RD <= 0;
        ID_EX_MEMO_WRT <= 0;
        ID_EX_REG_WRT <= 0;
        ID_VEX_OP     <= 0;
        ID_VEX_FUNC   <= 0;
        ID_VEX_RS_VAL <= 0;
        ID_VEX_RT_VAL <= 0;
        ID_VEX_RS_ADD <= 0;
        ID_VEX_RT_ADD <= 0;
        ID_VEX_RD_ADD <= 0;
        ID_VEX_SHAMT  <= 0;
        ID_VEX_IMM16  <= 0;
    end
end
```

```

    ID_VSTALL_INDEX <= 0;
    VSTALL <= 0;
    ID_VDF_INDEX <= 0;
end
else begin
    if (! id_is_vector) begin
        ID_EX_OP      <= ID_OP;
        ID_EX_FUNC    <= ID_FUNC;
        ID_EX_RS_VAL  <= ID_RS_VAL;
        ID_EX_RT_VAL  <= ID_RT_VAL;
        ID_EX_RS_ADD  <= ID_RS_ADD;
        ID_EX_RT_ADD  <= ID_RT_ADD;
        ID_EX_RD_ADD  <= ID_RD_ADD;
        ID_EX_SHAMT   <= ID_SHAMT;
        ID_EX_IMM16   <= ID_IMM16;
        ID_EX_ADDR    <= ID_ADDR;
        ID_EX_MEMO_RD <= ID_MEMO_RD;
        ID_EX_MEMO_WRT <= ID_MEMO_WRT;
        ID_EX_REG_WRT <= ID_JMP_STALL;
        ID_VEX_OP     <= 0;
        ID_VEX_FUNC   <= 0;
        ID_VEX_RS_VAL <= 0;
        ID_VEX_RT_VAL <= 0;
        ID_VEX_RS_ADD <= 0;
        ID_VEX_RT_ADD <= 0;
        ID_VEX_RD_ADD <= 0;
        ID_VEX_SHAMT  <= 0;
        ID_VEX_IMM16  <= 0;
    end
    else if (vstall_index==v_o_v_m1) begin
        ID_EX_OP      <= 0;
        ID_EX_FUNC    <= 0;
        ID_EX_RS_VAL  <= 0;
        ID_EX_RT_VAL  <= 0;
        ID_EX_RS_ADD  <= 0;
        ID_EX_RT_ADD  <= 0;
        ID_EX_RD_ADD  <= 0;
        ID_EX_SHAMT   <= 0;
        ID_EX_IMM16   <= 0;
        ID_EX_ADDR    <= 0;
        ID_EX_MEMO_RD <= 0;
        ID_EX_MEMO_WRT <= 0;
        ID_EX_REG_WRT <= 0;
        ID_VEX_OP     <= ID_OP;
        ID_VEX_FUNC   <= ID_FUNC;
        ID_VEX_RS_VAL <= ID_VRS_VAL;
        ID_VEX_RT_VAL <= ID_VRT_VAL;
        ID_VEX_RS_ADD <= ID_RS_ADD;
        ID_VEX_RT_ADD <= ID_RT_ADD ;
        ID_VEX_RD_ADD <= ID_RD_ADD;
        ID_VEX_SHAMT  <= ID_SHAMT;
        ID_VEX_IMM16  <= ID_IMM16;
    end // else: !if(! id_is_vector)
    ID_VSTALL_INDEX <= vstall_index;
    VSTALL <= ID_VSTALL;
    ID_VDF_INDEX    <= ID_INDEX;
end
end

/**/ Assigning output values for VeMICry interface ***/

assign rs_add = ID_VEX_RS_ADD >> 1;
assign rt_add = ID_VEX_RT_ADD >> 1;
assign rd_add = ID_VEX_RD_ADD;
assign shamt = ID_VEX_SHAMT;
assign op = ID_VEX_OP;
assign func = ID_VEX_FUNC;
assign imm16 = ID_VEX_IMM16;
assign vindex = ID_VDF_INDEX;
assign rs_val = ID_VEX_RS_VAL;

```

D. VEMICRY VERILOG

```
assign rt_val = ID_VEX_RT_VAL;

/**/ Execute stage ***/

assign EX_LD_STALL = LD_STALL;
assign EX_OP = ID_EX_OP;
assign EX_FUNC = ID_EX_FUNC;
assign EX_RD_ADD = ID_EX_RD_ADD;
assign EX_MEMO_RD = ID_EX_MEMO_RD;
assign EX_MEMO_WRT = ID_EX_MEMO_WRT;
assign EX_REG_WRT = !(ID_EX_REG_WRT) &&
                    (ID_EX_RD_ADD != 0) &&
                    !(ID_EX_MEMO_WRT);
assign EX_JMP_TAKEN = (!reset) ? (((ID_EX_OP == 'nop') && ((ID_EX_FUNC == 'jr') ||
                    (ID_EX_FUNC == 'jalr))) | (ID_EX_OP == 'j') |
                    (ID_EX_OP == 'jal')) : 1'b0;
assign EX_RS_VAL = ((EX_MEM_LD_STALL && (ID_EX_RS_ADD == MEM_WB_RD_ADD)) ||
                    ((ID_EX_OP != 'j') && (ID_EX_OP != 'jal') && (!EX_MEM_LD_STALL) &&
                    (ID_EX_RS_ADD == MEM_WB_RD_ADD) && (MEM_WB_REG_WRT))) ? MEM_WB_RES :
                    (((ID_EX_OP != 'j') && (ID_EX_OP != 'jal') && (!EX_MEM_LD_STALL) &&
                    (ID_EX_RS_ADD == EX_MEM_RD_ADD) && (EX_MEM_REG_WRT))) ? EX_MEM_RES : ID_EX_RS_VAL;
assign EX_RT_VAL = ((EX_MEM_LD_STALL && (ID_EX_OP == 'nop') && (ID_EX_RT_ADD == MEM_WB_RD_ADD)) ||
                    ((ID_EX_OP != 'j') && (ID_EX_OP != 'jal') && (!EX_MEM_LD_STALL) &&
                    (((ID_EX_OP == 'nop') || ID_EX_MEMO_WRT) && (ID_EX_RT_ADD == MEM_WB_RD_ADD) &&
                    MEM_WB_REG_WRT))) ? MEM_WB_RES :
                    ((ID_EX_OP != 'j') && (ID_EX_OP != 'jal') && (!EX_MEM_LD_STALL) &&
                    (((ID_EX_OP == 'nop') || ID_EX_MEMO_WRT) && (ID_EX_RT_ADD == EX_MEM_RD_ADD) &&
                    EX_MEM_REG_WRT)) ? EX_MEM_RES : ID_EX_RT_VAL;

assign EX_CONST = ((ID_EX_IMM16 & 'h8000) == 0) ? 16'h0000 :
                    16'hFFFF;
assign EX_IMM16 = {EX_CONST, ID_EX_IMM16};
assign EX_SHAMT = ID_EX_SHAMT;
assign EX_RT_ADD = ID_EX_RT_ADD;
assign EX_ADDR = ID_EX_ADDR;

always @(posedge clock or posedge reset)
if (reset) begin
    EX_BR_TAKEN <= 1'b0;
    EX_MEM_RES <= 0;
    EX_MEM_ACCESS <= 0;
    HI <= 0;
    LO <= 0;
    EX_MEM_LD_STALL <= 0;
    EX_MEM_OP <= 0;
    EX_MEM_FUNC <= 0;
    EX_MEM_RD_ADD <= 0;
    EX_MEM_MEMO_RD <= 0;
    EX_MEM_MEMO_WRT <= 0;
    EX_MEM_REG_WRT <= 0;
    EX_MEM_RT_VAL <= 0;
end
else begin
    if ((!LD_STALL) && (!pipe_stall_int)) begin
        if (EX_MEMO_RD || EX_MEMO_WRT) begin
            EX_MEM_RES <= EX_RS_VAL + EX_IMM16;
            EX_MEM_ACCESS <= ((EX_OP == 'lh) || (EX_OP == 'lhu) || (EX_OP == 'sh)) ? 1 :
                            ((EX_OP == 'lb) || (EX_OP == 'lbu) || (EX_OP == 'sb)) ? 2 :
                            0;
        end
    end
    else begin
        // Instructions Execution
        case (EX_OP)
            'nop: begin
                case (EX_FUNC)
                    'sll: EX_MEM_RES <= EX_RT_VAL << EX_SHAMT;
                    'srl: EX_MEM_RES <= EX_RT_VAL >> EX_SHAMT;
                    // 'sra: to be implemented
                    'sllv: EX_MEM_RES <= EX_RT_VAL << (EX_RS_VAL & 'h1F);
                    'srlv: EX_MEM_RES <= EX_RT_VAL >> (EX_RS_VAL & 'h1F);
                end
            end
        end
    end
end
```

```

//'srav: to be implemented
'jr:  EX_MEM_RES <= EX_RS_VAL;
'jalr: begin
    EX_MEM_RES <= EX_RS_VAL;
end
'movz: if (EX_RT_VAL == 0) begin
    EX_MEM_RES <= EX_RS_VAL;
end
'movn: if (EX_RT_VAL != 0) begin
    EX_MEM_RES <= EX_RS_VAL;
end
//'syscall:
//'break:
'mfhi:  EX_MEM_RES <= HI;
'mthi:  begin
    HI <= EX_RS_VAL;
end
'mflo:  EX_MEM_RES <= LO;
'mtlo:  begin
    LO <= EX_RS_VAL;
end
'mult:  {HI,LO} <= EX_RS_VAL * EX_RT_VAL; //sign?
'multu: {HI,LO} <= EX_RS_VAL * EX_RT_VAL;
'multp: {HI,LO} <= binmult(EX_RS_VAL , EX_RT_VAL);
//'div:
//'divu:
'add:   EX_MEM_RES <= EX_RS_VAL + EX_RT_VAL; //sign?
'addu:  EX_MEM_RES <= EX_RS_VAL + EX_RT_VAL;
'sub:   EX_MEM_RES <= EX_RS_VAL - EX_RT_VAL; //sign?
'subu:  EX_MEM_RES <= EX_RS_VAL - EX_RT_VAL;
'and:   EX_MEM_RES <= EX_RS_VAL & EX_RT_VAL;
'or:    EX_MEM_RES <= EX_RS_VAL | EX_RT_VAL;
'xor:   EX_MEM_RES <= EX_RS_VAL ^ EX_RT_VAL;
'nor:   EX_MEM_RES <= ~(EX_RS_VAL | EX_RT_VAL);
'slt:   EX_MEM_RES <= (EX_RS_VAL < EX_RT_VAL); //sign?
'sltu:  EX_MEM_RES <= (EX_RS_VAL < EX_RT_VAL);
default: ;
endcase // endcase for R-instructions
EX_BR_TAKEN <= 1'b0;
end // End R-type instructions
'bltz: begin
    case (EX_RT_ADD)
        'h00: begin // 'bltz
            EX_BR_TAKEN <= ((EX_RS_VAL & 'h80000000) != 0);
            EX_MEM_RES <= (((EX_RS_VAL & 'h80000000) != 0) & !EX_JMP_TAKEN) ?
                PC + (EX_IMM16 << 2) - 4 : EX_MEM_RES;
        end
        'h01: begin // 'bgez
            EX_BR_TAKEN <= ((EX_RS_VAL & 'h80000000) == 0);
            EX_MEM_RES <= (((EX_RS_VAL & 'h80000000) == 0) & !EX_JMP_TAKEN) ?
                PC + (EX_IMM16 << 2) - 4 : EX_MEM_RES;
        end
        'h10: if (EX_RS_VAL & 'h80000000) begin
            EX_BR_TAKEN <= 1;
            if (!EX_JMP_TAKEN)
                EX_MEM_RES <= PC + (EX_IMM16 << 2) - 4;
        end // 'bltzal
        'h11: if (!(EX_RS_VAL & 'h80000000)) begin
            EX_BR_TAKEN <= 1;
            if (!EX_JMP_TAKEN)
                EX_MEM_RES <= PC + (EX_IMM16 << 2) - 4;
        end // 'bgezal
    endcase // rt for branch instructions
end // case: 'bltz
'beq: begin
    EX_BR_TAKEN <= (EX_RT_VAL == EX_RS_VAL);
    EX_MEM_RES <= ((EX_RT_VAL == EX_RS_VAL) & !EX_JMP_TAKEN) ?
        PC + (EX_IMM16 << 2) - 4 : EX_MEM_RES;
end
'bne: begin

```

D. VEMICRY VERILOG

```
EX_BR_TAKEN <= (EX_RT_VAL != EX_RS_VAL);
EX_MEM_RES <= ((EX_RT_VAL != EX_RS_VAL) & !EX_JMP_TAKEN) ?
    PC + (EX_IMM16 << 2) - 4 : EX_MEM_RES;

end
`blez: begin
    EX_BR_TAKEN <= ((EX_RS_VAL == 0) || (EX_RS_VAL[pWIDTH-1]));
    EX_MEM_RES <= (((EX_RS_VAL == 0) || (EX_RS_VAL[pWIDTH-1])) & !EX_JMP_TAKEN) ?
        PC + (EX_IMM16 << 2) - 4 : EX_MEM_RES;
end
`bgtz: begin
    EX_BR_TAKEN <= ((EX_RS_VAL > 0) && !(EX_RS_VAL[pWIDTH-1]));
    EX_MEM_RES <= ((EX_RS_VAL > 0) && !(EX_RS_VAL[pWIDTH-1]) & !EX_JMP_TAKEN) ?
        PC + (EX_IMM16 << 2) - 4 : EX_MEM_RES;
end
`addi: begin
    EX_MEM_RES <= EX_RS_VAL + EX_IMM16;
    EX_BR_TAKEN <= 1'b0; end
`addiu: begin
    EX_MEM_RES <= EX_RS_VAL + EX_IMM16[15:0];
    EX_BR_TAKEN <= 1'b0; end
`slti: begin
    EX_MEM_RES <= (EX_RS_VAL < EX_IMM16);
    EX_BR_TAKEN <= 1'b0; end
`andi: begin
    EX_MEM_RES <= EX_RS_VAL & EX_IMM16[15:0];
    EX_BR_TAKEN <= 1'b0; end
`sltiu: EX_BR_TAKEN <= 1'b0; // to implement
`ori: begin
    EX_MEM_RES <= EX_RS_VAL | EX_IMM16[15:0];
    EX_BR_TAKEN <= 1'b0; end
`xori: begin
    EX_MEM_RES <= EX_RS_VAL ^ EX_IMM16[15:0];
    EX_BR_TAKEN <= 1'b0; end
`lui: begin
    EX_MEM_RES <= {EX_IMM16[15:0], 16'h0000};
    EX_BR_TAKEN <= 1'b0; end
`j: begin
    EX_MEM_RES <= (PC & 'hf0000000) | (EX_ADDR << 2);
    EX_BR_TAKEN <= 1'b0; end
`jal: begin
    EX_MEM_RES <= (PC & 'hf0000000) | (EX_ADDR << 2);
    EX_BR_TAKEN <= 1'b0;
end
default: EX_BR_TAKEN <= 1'b0;
endcase // endcase for main execution
end // if no load/store instruction
end // If for load stall
EX_MEM_LD_STALL <= EX_LD_STALL;
EX_MEM_OP <= EX_OP;
EX_MEM_FUNC <= EX_FUNC;
EX_MEM_RD_ADD <= EX_RD_ADD;
EX_MEM_MEMO_RD <= EX_MEMO_RD;
EX_MEM_MEMO_WRT <= EX_MEMO_WRT;
EX_MEM_ACCESS <= ((EX_OP == 'lh) || (EX_OP == 'lhu) || (EX_OP == 'sh)) ? 1:
    ((EX_OP == 'lb) || (EX_OP == 'lbu) || (EX_OP == 'sb)) ? 2:
    0;
if ((EX_OP == 'nop) && ((EX_FUNC == 'movz) && (EX_RT_VAL != 0)) ||
    ((EX_FUNC == 'movn) && (EX_RT_VAL == 0)) || (EX_FUNC == 'mthi) || (EX_FUNC == 'mtlo))
    EX_MEM_REG_WRT <= 0;
else
    EX_MEM_REG_WRT <= EX_REG_WRT;
    if ((EX_OP == 'nop) && (EX_FUNC == 'jalr) && (EX_RD_ADD != 0))
        EX_MEM_RT_VAL <= PC-4;
    else
        EX_MEM_RT_VAL <= EX_RT_VAL;
end // end always for EX stage

/** MEMory Stage */

always @*
```

```

    if (!EX_MEM_LD_STALL)
        MEM_MEMO_RD <= EX_MEM_MEMO_RD;

always @*
    if (!EX_MEM_LD_STALL)
        MEM_MEMO_WRT <= EX_MEM_MEMO_WRT;

always @*
    if (!EX_MEM_LD_STALL)
        MEM_ACCESS <= EX_MEM_ACCESS;

always @*
    if (!EX_MEM_LD_STALL) begin
        if (!(EX_MEM_MEMO_RD || EX_MEM_MEMO_WRT)) && (EX_MEM_OP == 'nop') &&
            (EX_MEM_FUNC == 'jalr') begin
                MEM_RES <= EX_MEM_RT_VAL;
            end
        else MEM_RES <= EX_MEM_RES;
    end

always @(posedge clock or posedge reset)
    if (reset) begin
        MEM_WB_REG_WRT <= 0;
        MEM_WB_RD_ADD <= 0;
        MEM_WB_RES <= 0;
        MEM_WB_LD_STALL <= 0;
    end
    else begin
        if (sbi_sig) begin
            MEM_WB_REG_WRT <= sbi_sig;
            MEM_WB_RD_ADD <= sbi_add;
            MEM_WB_RES <= sbi_val_int;
        end
        else begin
            MEM_WB_REG_WRT <= EX_MEM_REG_WRT;
            MEM_WB_RD_ADD <= EX_MEM_RD_ADD;
            MEM_WB_LD_STALL <= EX_MEM_LD_STALL;
            if (EX_MEM_MEMO_RD) begin
                MEM_WB_RES <= memdataout[pWIDTH-1:0];
            end
            else begin
                MEM_WB_RES <= MEM_RES[pWIDTH-1:0];
            end
        end
    end // else: !if(sbi_sig)
end // always @ (posedge clock)

/**/ Output signals to (vector) data memory /**/
assign debugw = debug;
assign typ = (MEM_MEMO_WRT || MEM_MEMO_RD);
assign access = MEM_ACCESS;
assign data_read = MEM_MEMO_RD;
assign data_write = MEM_MEMO_WRT;
assign data_add = MEM_RES;
assign data_in_mem = EX_MEM_RT_VAL;

/**/ Register Write stage /**/

always @(posedge clock or posedge reset)
    if (reset)
        for (i=0; i<pNUMREG; i=i+1)
            SCAREG[i]<=0;
    else begin
        if ((MEM_WB_REG_WRT == 1) && (MEM_WB_RD_ADD != 0) && (!MEM_WB_LD_STALL)) begin
            SCAREG[MEM_WB_RD_ADD] <= MEM_WB_RES;
        end
        else begin
            SCAREG[0] <= 32'h0;
        end
    end
    if ((ID_EX_OP == 'jal) || ((ID_EX_OP == 'jalr) && (ID_EX_RD_ADD == 0)) ||
        ((ID_EX_OP == 'bltz) && (((ID_EX_RT_ADD == 'h10) && (EX_RS_VAL & 'h80000000)) ||

```

D. VEMICRY VERILOG

```
        ((ID_EX_RT_ADD=='h11)&&(!(EX_RS_VAL & 'h80000000)))))) begin
            SCAREG['RA] <= PC - 4;
        end
    end
end // always @ (posedge clock)

function [(2*pWIDTH)-1:0] binmult;
    input[pWIDTH-1:0] op1;
    input[pWIDTH-1:0] op2;
    reg[7:0] loop;
    reg [(2*pWIDTH)-1:0] temp;
    reg [pWIDTH-1:0] optemp;

    begin
        temp = 0;
        for (loop=pWIDTH; loop > 0; loop=loop-1) begin
            optemp = (op2[loop-1] == 1)? op1:32'h00000000;
            temp = (temp << 1) ^ optemp;
        end
        binmult = temp;
    end // always @ (op1 or op2)
endfunction // binmult
endmodule // Module scamips
```

D.2 Vector Processor Verilog

```
/** Instruction Set Definition in file "instruction_set.v" ***/
/** TOP LEVEL MODULE ***/

module vemicro_module ( clock, reset, typ, data_read, data_write,
    data_add, data_in_mem, memdataout, ID_VDF_VS_ADD, ID_VDF_VT_ADD, ID_VDF_VD_ADD,
    ID_VDF_SHAMT, ID_VDF_OP, ID_VDF_FUNC, ID_VDF_IMM16, ID_VDF_INDEX, ID_EX_RS_VAL,
    ID_EX_RT_VAL, pipe_stall, sbi_sig, sbi_add, sbi_val);

    // Parameters
    parameter period = 20;
    parameter pWIDTH = 32;
    parameter pSIZEMEM = 2080;
    parameter vq = 4;
    parameter vp = 32;
    parameter vr = 32;
    parameter vp_over_vr_width = 8;
    parameter vp_over_vr_minus1 = (vp/vr)-1;

    reg [pWIDTH-1:0] CAR;
    reg [pWIDTH-1:0] SBI;

    // Vector Lanes IOs
    reg [vp_over_vr_width-1:0] VDF_VEXM_INDEX;
    reg [vp_over_vr_width-1:0] VEXM_VEXC_INDEX, VEXC_VWB_INDEX;

    reg [15:0] VDF_VEXM_IMM16;
    reg [15:0] VEXM_VEXC_IMM16;

    reg [pWIDTH-1:0] VEXM_VEXC_RS_VAL;

    reg [pWIDTH-1:0] VDF_VEXM_RS_VAL;

    wire [0:(pWIDTH*vr)-1] LANE_OUT_VAL;
    reg [4:0] VDF_VEXM_RD_ADD, VEXM_VEXC_RD_ADD, VEXC_VWB_RD_ADD;

    // Data Memory Control Signals
    reg [0:(pWIDTH*vr)-1] memaddress;
    wire [0:(pWIDTH*vr)-1] LANE_CARRIES;
    wire [0:(pWIDTH*vr)-1] LANE_CARRIES_32;
    reg [0:(pWIDTH*vr)-1] LANE_CARRIES_REG;
    wor SHIFT_CARRIES;
    wor WAY;
    reg debug;
```

```

wor                mem_write_wire;
wor                mem_read_wire;
wor                car_write_wire;
reg [pWIDTH-1:0]  sbi_reg;
wor                sbi_write_wire;

// Module IOs
input [4:0]        ID_VDF_VS_ADD;
input [4:0]        ID_VDF_VT_ADD;
input [4:0]        ID_VDF_VD_ADD;
input [4:0]        ID_VDF_SHAMT;
input [5:0]        ID_VDF_OP;
input [5:0]        ID_VDF_FUNC;
input [15:0]       ID_VDF_IMM16;
input [vp_over_vr_width-1:0] ID_VDF_INDEX;
input [pWIDTH-1:0] ID_EX_RS_VAL;
input [pWIDTH-1:0] ID_EX_RT_VAL;
input              clock;
input              reset;
input [0:(pWIDTH*vr)-1] memdataout;
output             typ;
output             data_read;
output             data_write;
output [0:(pWIDTH*vr)-1] data_add;
output [pWIDTH-1:0] data_in_mem;
output             pipe_stall;
output             sbi_sig;
output [4:0]       sbi_add;
output [pWIDTH-1:0] sbi_val;
reg                sbi_sync;

// Control Signals
reg [7:0]          ID_VSTALL_INDEX;
wire              PIPE_STALL;
wire              FLOAT_WIRE;

// VeMICry - Delaying Index Signal & OP signal and imm16 signals//
always @(posedge clock or posedge reset)
  if (reset) begin
    VDF_VEXM_INDEX   <= 0;
    VEXM_VEXC_INDEX <= 0;
    VEXC_VWB_INDEX  <= 0;
    VDF_VEXM_IMM16  <= 0;
    VEXM_VEXC_IMM16 <= 0;
    VDF_VEXM_RS_VAL <= 0;
    VEXM_VEXC_RS_VAL <= 0;
    VDF_VEXM_RD_ADD <= 0;
    VEXM_VEXC_RD_ADD <= 0;
    VEXC_VWB_RD_ADD <= 0;
  end
  else begin
    if (!PIPE_STALL) begin
      VDF_VEXM_INDEX   <= ID_VDF_INDEX;
      VEXM_VEXC_INDEX <= VDF_VEXM_INDEX;
      VEXC_VWB_INDEX  <= VEXM_VEXC_INDEX;
      VDF_VEXM_IMM16  <= ID_VDF_IMM16;
      VEXM_VEXC_IMM16 <= VDF_VEXM_IMM16;
      VDF_VEXM_RS_VAL <= ID_EX_RS_VAL;
      VEXM_VEXC_RS_VAL <= VDF_VEXM_RS_VAL;
      VDF_VEXM_RD_ADD <= ID_VDF_VD_ADD;
      VEXM_VEXC_RD_ADD <= VDF_VEXM_RD_ADD;
      VEXC_VWB_RD_ADD <= VEXM_VEXC_RD_ADD;
    end // if (!PIPE_STALL)
  end

assign typ = (mem_read_wire || mem_write_wire);
assign data_read = mem_read_wire;
assign data_write = mem_write_wire;
assign data_add = LANE_OUT_VAL;

```

D. VEMICRY VERILOG

```
assign data_in_mem = (VEXM_VEXC_RS_VAL+(VEXM_VEXC_INDEX*4*vr));
assign pipe_stall = PIPE_STALL;

// VeMICry - Generate Parallel Lanes //

genvar i;

generate
for (i=0; i < vr; i=i+1) begin : VLANES
if (i==0) begin
vector_lane #(pWIDTH, vr, vq, vp_over_vr_minus1, vp_over_vr_width, i)
vector_lane (clock, reset, ID_VDF_OP, ID_VDF_FUNC, ID_VDF_IMM16, ID_VDF_SHAMT,
ID_VDF_VS_ADD, ID_VDF_VT_ADD, (ID_VDF_VD_ADD >> 1),
(ID_EX_RS_VAL+((i+(vr*ID_VDF_INDEX))<<2)), ID_EX_RT_VAL,
ID_VDF_INDEX, memdataout[(pWIDTH*i):((pWIDTH*(i+1))-1)],
LANE_OUT_VAL[(pWIDTH*i):((pWIDTH*(i+1))-1)],
CAR, LANE_CARRIES[(pWIDTH*i):((pWIDTH*(i+1))-1)],
SHIFT_CARRIES, WAY, mem_read_wire,
mem_write_wire, car_write_wire, sbi_write_wire, PIPE_STALL);
end
else begin
vector_lane #(pWIDTH, vr, vq, vp_over_vr_minus1, vp_over_vr_width, i)
vector_lane (clock, reset, ID_VDF_OP, ID_VDF_FUNC, ID_VDF_IMM16, ID_VDF_SHAMT,
ID_VDF_VS_ADD, ID_VDF_VT_ADD, (ID_VDF_VD_ADD >> 1),
(ID_EX_RS_VAL+((i+(vr*ID_VDF_INDEX))<<2)), ID_EX_RT_VAL,
ID_VDF_INDEX, memdataout[(pWIDTH*i):((pWIDTH*(i+1))-1)],
LANE_OUT_VAL[(pWIDTH*i):((pWIDTH*(i+1))-1)],
LANE_CARRIES_REG[(pWIDTH*i):((pWIDTH*(i+1))-1)],
LANE_CARRIES[(pWIDTH*i):((pWIDTH*(i+1))-1)],
SHIFT_CARRIES, WAY, mem_read_wire,
mem_write_wire, car_write_wire, sbi_write_wire, FLOAT_WIRE);
end // else: !if(i==0)
end // block: VLANES
endgenerate

// VeMICry - CAR register management //

assign LANE_CARRIES_32 = LANE_CARRIES << 32;

always @*
if (car_write_wire == 1) begin
if (SHIFT_CARRIES == 1) begin
if (WAY == 0) begin // shift left
LANE_CARRIES_REG <= LANE_CARRIES >> 32;
end
else begin
LANE_CARRIES_REG[0:((pWIDTH*vr)-1)] <= LANE_CARRIES_32[0:((pWIDTH*vr)-1)];
LANE_CARRIES_REG[(pWIDTH*(vr-1)):((pWIDTH*vr)-1)] <= CAR;
end
end
else begin
LANE_CARRIES_REG <= LANE_CARRIES;
end // else: !if(SHIFT_CARRIES == 1)
end

// VeMICry - SBI registers management //

always @*
if ((ID_VDF_OP == 'vextract) && (ID_VDF_INDEX == 0))
sbi_reg <= 0;
else
if (sbi_write_wire)
sbi_reg <= sbi_reg | LANE_CARRIES[0:31];

always @(posedge clock or posedge reset)
if (reset) begin
CAR <= 0;
SBI <= 0;
sbi_sync <= 0;
end
```

```

else begin
  if (car_write_wire) begin
    CAR <= (WAY == 0)? LANE_CARRIES[(pWIDTH*(vr-1)):((pWIDTH*vr)-1)]:
      32'h0;
  end
  if (sbi_write_wire) begin
    if (VEXM_VEXC_IMM16 == 0)begin
      SBI <= CAR;
    end
    else begin
      SBI <= sbi_reg;
    end
    sbi_sync <= (VEXC_VWB_INDEX == vp_over_vr_minus1);
  end
  else begin
    sbi_sync <= 0;
  end // else: !if(sbi_write_wire)
end // always @ (posedge clock)

assign sbi_sig = sbi_sync;
assign sbi_val = SBI;
assign sbi_add = VEXC_VWB_RD_ADD; //SBI_RD_ADD;

endmodule // Top Level VeMICry

/** Vector Processor Lane Definition **//

module vector_lane (clock, reset, op, func, imm16, shamt, vs_add,
  vt_add, vd_add, sbi, scalar, index, data_in,
  data_out, carry_in, carry_out, shift, way,
  mem_read, mem_write, car_write, sbi_write, is_pipe_stall);

  // Parameters

  parameter          lawidth = 32;
  parameter          vr      = 1;
  parameter          vq      = 2;
  parameter          vpvr    = 2;
  parameter          vp_over_vr_width = 8;
  parameter          i_number = 0;

  // Vector Register File defined per lane
  reg [lawidth-1:0]  VECREG[0:vq-1][0:vpvr]; // vector register file

  // Module IOs
  input [vp_over_vr_width-1:0] index;
  input          clock;
  input          reset;
  input [5:0]    op;
  input [5:0]    func;
  input [4:0]    shamt;
  input [15:0]   imm16;
  input [4:0]    vs_add;
  input [4:0]    vt_add;
  input [4:0]    vd_add;
  input [lawidth-1:0] sbi;
  input [lawidth-1:0] scalar;
  input [lawidth-1:0] data_in;
  input [lawidth-1:0] carry_in;

  output          mem_read;
  output          mem_write;
  output          car_write;
  output          sbi_write;
  output          is_pipe_stall;
  output [lawidth-1:0] data_out;
  output          shift;
  output          way;
  output [lawidth-1:0] carry_out;

```

D. VEMICRY VERILOG

```
// VDF internal registers
reg [lwidth-1:0]   vdf_vt_val;
reg [lwidth-1:0]   vdf_vs_val;
reg [5:0]          vdf_op;
reg [5:0]          vdf_func;
reg [4:0]          vdf_vs_add;
reg [4:0]          vdf_vt_add;
//reg [4:0]         vdf_vd_add;
//reg [4:0]         vdf_shamt;
//reg [15:0]        vdf_imm16;
//reg [lwidth-1:0]  vdf_sbi;
reg [vp_over_vr_width-1:0] vdf_index;

// VDF-VEXM pipeline
reg [lwidth-1:0]   VDF_VEXM_VT_VAL;
reg [lwidth-1:0]   VDF_VEXM_VS_VAL;
reg [5:0]          VDF_VEXM_OP;
reg [5:0]          VDF_VEXM_FUNC;
reg [4:0]          VDF_VEXM_VS_ADD;
reg [4:0]          VDF_VEXM_VT_ADD;
reg [4:0]          VDF_VEXM_VD_ADD;
reg [4:0]          VDF_VEXM_SHAMT;
reg [15:0]         VDF_VEXM_IMM16;
reg [lwidth-1:0]   VDF_VEXM_SBI;
reg [lwidth-1:0]   VDF_VEXM_SCALAR;
reg [lwidth-1:0]   VDF_VEXM_CARRYIN;
reg [vp_over_vr_width-1:0] VDF_VEXM_INDEX;
reg                VDF_PIPE_STALL;

// VEXM Internal Registers
reg [lwidth-1:0]   vexm_vt_val;
reg [lwidth-1:0]   vexm_vs_val;
wire[5:0]          vexm_op;
reg [5:0]          vexm_func;
reg [4:0]          vexm_vs_add;
reg [4:0]          vexm_vt_add;
reg [4:0]          vexm_vd_add;
reg [4:0]          vexm_shamt;
reg [15:0]         vexm_imm16;
reg [(2*lwidth)-1:0] vexm_res_val;
reg [lwidth-1:0]   vexm_scalar;
reg [lwidth-1:0]   vexm_sbi;
wire               vexm_mem_read;
wire               vexm_mem_write;
wire               vexm_car_write;
wire               vexm_sbi_write;
reg [vp_over_vr_width-1:0] vexm_index;
wire               vexm_shift;
wire               vexm_way;

// VEXM-VEXC pipeline
reg [lwidth-1:0]   VEXM_VEXC_VT_VAL;
reg [lwidth-1:0]   VEXM_VEXC_VS_VAL;
reg [lwidth-1:0]   VEXM_VEXC_CARRY;
reg [5:0]          VEXM_VEXC_OP;
reg [5:0]          VEXM_VEXC_FUNC;
reg [4:0]          VEXM_VEXC_VS_ADD;
reg [4:0]          VEXM_VEXC_VT_ADD;
reg [4:0]          VEXM_VEXC_VD_ADD;
reg [4:0]          VEXM_VEXC_SHAMT;
reg [15:0]         VEXM_VEXC_IMM16;
reg                VEXM_VEXC_MEM_READ;
reg                VEXM_VEXC_MEM_WRITE;
reg                VEXM_VEXC_CAR_WRT;
reg                VEXM_VEXC_SBI_WRT;
reg [vp_over_vr_width-1:0] VEXM_VEXC_INDEX;
reg                VEXM_VEXC_SHIFT;
reg                VEXM_VEXC_WAY;

// VEXC Internal Registers
```

```

wire [lwidth-1:0]   vexc_vt_val;
wire [lwidth-1:0]   vexc_vs_val;
wire [5:0]          vexc_op;
wire [5:0]          vexc_func;
wire [4:0]          vexc_vs_add;
wire [4:0]          vexc_vt_add;
wire [4:0]          vexc_shamt;
wire [15:0]         vexc_imm16;
wire [vp_over_vr_width-1:0] vexc_index;
wire [4:0]          vexc_vd_add;
wire               vexc_reg_wrt;
reg [lwidth-1:0]    vexc_res_val;
wire [lwidth-1:0]   vexc_carryin;
wire               vexc_mem_read;
wire               vexc_mem_write;

// VEXC-VWB pipeline
reg               VEXC_VWB_MEM_WRT;
reg               VEXC_VWB_MEM_READ;
reg               VEXC_VWB_REG_WRT;
reg [4:0]         VEXC_VWB_VD_ADD;
reg [vp_over_vr_width-1:0] VEXC_VWB_INDEX;
reg [lwidth-1:0]   VEXC_VWB_RES;
reg [lwidth-1:0]   VEXC_VWB_DATAIN;

wire               pipe_stall;

// Vector Lane - Vector Data Fetch //

assign pipe_stall = (VDF_PIPE_STALL) ? 1'b0 :
  (((VDF_VEXM_VD_ADD == vs_add) && (VDF_VEXM_INDEX == index) &&
    (op != 'vload) && (op != 'vsmove) && (op != 0)) ||
    ((VDF_VEXM_VD_ADD == vt_add) && (VDF_VEXM_INDEX == index) &&
    (op == 'vr1) && (func == 'vxor_func))) &&
  (VDF_VEXM_OP != 0));

always @*
  if (!VDF_PIPE_STALL)
    vdf_op <= op;
always @*
  if (!VDF_PIPE_STALL)
    vdf_func <= func;
always @*
  if (!VDF_PIPE_STALL)
    vdf_vs_add <= vs_add;
always @*
  if (!VDF_PIPE_STALL)
    vdf_vt_add <= vt_add;
always @*
  if (!VDF_PIPE_STALL)
    vdf_index <= index;

always @*
  // Register Bypass for VS
  if (vdf_op != 0) begin
    if ((vdf_vs_add == VEXC_VWB_VD_ADD) && (vdf_index == VEXC_VWB_INDEX) &&
      (VEXC_VWB_REG_WRT)) begin
      if (VEXC_VWB_MEM_READ) begin
        vdf_vs_val <= VEXC_VWB_DATAIN;
      end
      else begin
        vdf_vs_val <= VEXC_VWB_RES;
      end
    end
    else begin
      vdf_vs_val <= VECREG[vdf_vs_add][vdf_index];
    end
  end // if (vdf_op != 0)

always @*

```

D. VEMICRY VERILOG

```
// Register Bypass for VT
if ((vdf_op == 'vr1) && (vdf_func == 'vxor_func)) begin
  if ((vdf_vt_add == VEXC_VWB_VD_ADD)
      && (vdf_index == VEXC_VWB_INDEX)
      && (VEXC_VWB_REG_WRT)) begin
    if (VEXC_VWB_MEM_READ) begin
      vdf_vt_val <= VEXC_VWB_DATAIN;
    end
    else begin
      vdf_vt_val <= VEXC_VWB_RES;
    end
  end
  end
else begin
  vdf_vt_val <= VECREG[vdf_vt_add][vdf_index];
end
end

always @(posedge clock or posedge reset)
if (reset) begin
  VDF_VEXM_VT_VAL <= 0;
  VDF_VEXM_VS_VAL <= 0;
  VDF_VEXM_OP <= 0;
  VDF_VEXM_FUNC <= 0;
  VDF_VEXM_VS_ADD <= 0;
  VDF_VEXM_VT_ADD <= 0;
  VDF_VEXM_VD_ADD <= 0;
  VDF_VEXM_SHAMT <= 0;
  VDF_VEXM_IMM16 <= 0;
  VDF_VEXM_SBI <= 0;
  VDF_VEXM_INDEX <= 0;
  VDF_VEXM_SCALAR <= 0;
  VDF_PIPE_STALL <= 0;
end
else begin
  if (!VDF_PIPE_STALL) begin
    VDF_VEXM_VT_VAL <= vdf_vt_val;
    VDF_VEXM_VS_VAL <= vdf_vs_val;
    VDF_VEXM_FUNC <= vdf_func;
    VDF_VEXM_VS_ADD <= vdf_vs_add;
    VDF_VEXM_VT_ADD <= vdf_vt_add;
    VDF_VEXM_INDEX <= vdf_index;
    VDF_VEXM_SCALAR <= scalar;
  end
  VDF_VEXM_OP <= vdf_op;
  VDF_PIPE_STALL <= pipe_stall;
  if (!VDF_PIPE_STALL)
    VDF_VEXM_VD_ADD <= vd_add;
  if (!VDF_PIPE_STALL)
    VDF_VEXM_SHAMT <= shamt;
  if (!VDF_PIPE_STALL)
    VDF_VEXM_IMM16 <= imm16;
  if (!VDF_PIPE_STALL)
    VDF_VEXM_SBI <= sbi;
end
end

assign is_pipe_stall = VDF_PIPE_STALL;

// Vector Lane - Vector EXecute Multiply //

assign vexm_op = (VDF_PIPE_STALL!=1) ? VDF_VEXM_OP :
  6'b0;
assign vexm_mem_read = (vexm_op == 'vload);
assign vexm_mem_write = (vexm_op == 'vstore);
assign vexm_car_write = (((vexm_op == 'vr1) && (VDF_VEXM_FUNC == 'vspmult_func)) ||
  (vexm_op == 'vwshl) || (vexm_op == 'vwshr) ||
  ((vexm_op == 'vsmove)&&(VDF_VEXM_IMM16==0)));
assign vexm_sbi_write = (vexm_op == 'vextract);
assign vexm_shift = (((vexm_op == 'vr1) && (VDF_VEXM_FUNC == 'vspmult_func)) ||
  (vexm_op == 'vwshl) || (vexm_op == 'vwshr));
assign vexm_way = (vexm_op == 'vwshr); // if zero, shift left else right
```

```

always @*
  if (((vexm_op != 'vload) && (vexm_op != 'vsmove)) && ((VDF_VEXM_VS_ADD == VEXC_VWB_VD_ADD) &&
    (VDF_VEXM_INDEX == VEXC_VWB_INDEX) && (VEXC_VWB_REG_WRT))) begin
    if (VEXC_VWB_MEM_READ) begin
      vexm_vs_val <= VEXC_VWB_DATAIN;
    end
    else begin
      vexm_vs_val <= VEXC_VWB_RES;
    end
  end
  else if (!VDF_PIPE_STALL)
    vexm_vs_val <= VDF_VEXM_VS_VAL;

always @(posedge clock or posedge reset)
  if (reset)
    {VEXM_VEXC_CARRY, VEXM_VEXC_VS_VAL} <= 0;
  else
    case (vexm_op)
      'vextract: VEXM_VEXC_CARRY <= (((VDF_VEXM_INDEX*vr)+i_number+1)==VDF_VEXM_IMM16)
        ? vexm_vs_val : 32'h0;
      'vsmove: {VEXM_VEXC_CARRY, VEXM_VEXC_VS_VAL} <= {VDF_VEXM_SCALAR, VDF_VEXM_SCALAR};
      'vload: VEXM_VEXC_VS_VAL <= VDF_VEXM_SBI;
      'vwshl: {VEXM_VEXC_CARRY, VEXM_VEXC_VS_VAL} <= {vexm_vs_val, 32'h0};
      'vwshr: {VEXM_VEXC_CARRY, VEXM_VEXC_VS_VAL} <= {vexm_vs_val, 32'h0};
      'vr1: begin
        case (VDF_VEXM_FUNC)
          'vspmult_func: {VEXM_VEXC_CARRY, VEXM_VEXC_VS_VAL} <= binmult(vexm_vs_val, VDF_VEXM_SCALAR);
          'vxor_func : VEXM_VEXC_VS_VAL <= vexm_vs_val;
        endcase
      end
      default: VEXM_VEXC_VS_VAL <= vexm_vs_val;
    endcase

always @(posedge clock or posedge reset)
  if (reset) begin
    VEXM_VEXC_VT_VAL <= 0;
    VEXM_VEXC_OP <= 0;
    VEXM_VEXC_FUNC <= 0;
    VEXM_VEXC_VS_ADD <= 0;
    VEXM_VEXC_VT_ADD <= 0;
    VEXM_VEXC_VD_ADD <= 0;
    VEXM_VEXC_SHAMT <= 0;
    VEXM_VEXC_IMM16 <= 0;
    VEXM_VEXC_MEM_READ <= 0;
    VEXM_VEXC_MEM_WRITE <= 0;
    VEXM_VEXC_INDEX <= 0;
    VEXM_VEXC_CAR_WRT <= 0;
    VEXM_VEXC_SBI_WRT <= 0;
    VEXM_VEXC_SHIFT <= 0;
    VEXM_VEXC_WAY <= 0;
  end
  else begin
    if ((vexm_op == 'vr1) && (VDF_VEXM_FUNC == 'vxor_func) && (VDF_VEXM_VT_ADD == VEXC_VWB_VD_ADD)
      && (VDF_VEXM_INDEX == VEXC_VWB_INDEX) && (VEXC_VWB_REG_WRT))
      if (VEXC_VWB_MEM_READ)
        VEXM_VEXC_VT_VAL <= VEXC_VWB_DATAIN;
      else
        VEXM_VEXC_VT_VAL <= VEXC_VWB_RES;
    else if (VDF_PIPE_STALL != 1)
      VEXM_VEXC_VT_VAL <= VDF_VEXM_VT_VAL;
      VEXM_VEXC_OP <= vexm_op;
    if (!VDF_PIPE_STALL)
      VEXM_VEXC_FUNC <= VDF_VEXM_FUNC;
      VEXM_VEXC_VS_ADD <= VDF_VEXM_VS_ADD;
      VEXM_VEXC_VT_ADD <= VDF_VEXM_VT_ADD;
      VEXM_VEXC_VD_ADD <= VDF_VEXM_VD_ADD;
      VEXM_VEXC_SHAMT <= VDF_VEXM_SHAMT;
      VEXM_VEXC_IMM16 <= VDF_VEXM_IMM16;
      VEXM_VEXC_MEM_READ <= vexm_mem_read;
  end

```

D. VEMICRY VERILOG

```
VEXM_VEXC_MEM_WRITE <= vexam_mem_write;
VEXM_VEXC_INDEX     <= VDF_VEXM_INDEX;
VEXM_VEXC_CAR_WRT  <= vexam_car_write;
VEXM_VEXC_SBI_WRT  <= vexam_sbi_write;
VEXM_VEXC_SHIFT    <= vexam_shift;
VEXM_VEXC_WAY      <= vexam_way;
end // always @ (posedge clock)

assign carry_out      = VEXM_VEXC_CARRY;
assign car_write      = VEXM_VEXC_CAR_WRT;
assign sbi_write      = VEXM_VEXC_SBI_WRT;
assign shift          = VEXM_VEXC_SHIFT;
assign way            = VEXM_VEXC_WAY;
assign vexc_vt_val    = VEXM_VEXC_VT_VAL;
assign vexc_vs_val    = VEXM_VEXC_VS_VAL;
assign vexc_vs_add    = VEXM_VEXC_VS_ADD;
assign vexc_vt_add    = VEXM_VEXC_VT_ADD;
assign vexc_shamt     = VEXM_VEXC_SHAMT;
assign vexc_reg_wrt   = (((VEXM_VEXC_OP == 'vr1) && ((VEXM_VEXC_FUNC == 'vxor_func) ||
(VEXM_VEXC_FUNC == 'vspmult_func))) || (VEXM_VEXC_OP == 'vload) ||
(VEXM_VEXC_OP == 'vwshl) || (VEXM_VEXC_OP == 'vwshr) ||
((VEXM_VEXC_OP == 'vsmove) &&
(((VEXM_VEXC_INDEX*vr) + i_number) < VEXM_VEXC_IMM16)));
assign data_out = ((VEXM_VEXC_OP=='vwshl)|| (VEXM_VEXC_OP=='vwshr)) ? carry_in :
((VEXM_VEXC_OP=='vr1)&&(VEXM_VEXC_FUNC=='vxor_func)) ?
(VEXM_VEXC_VT_VAL ^ VEXM_VEXC_VS_VAL) :
((VEXM_VEXC_OP=='vr1)&&(VEXM_VEXC_FUNC=='vspmult_func)) ?
(carry_in ^ VEXM_VEXC_VS_VAL) : VEXM_VEXC_VS_VAL;

assign mem_read  = VEXM_VEXC_MEM_READ;
assign mem_write = VEXM_VEXC_MEM_WRITE;

always @(posedge clock or posedge reset)
if (reset) begin
VEXC_VWB_REG_WRT <= 0;
VEXC_VWB_VD_ADD <= 0;
VEXC_VWB_INDEX <= 0;
VEXC_VWB_RES <= 0;
VEXC_VWB_DATAIN <= 0;
VEXC_VWB_MEM_WRT <= 0;
VEXC_VWB_MEM_READ <= 0;
end
else begin
VEXC_VWB_REG_WRT <= vexc_reg_wrt;
VEXC_VWB_VD_ADD <= VEXM_VEXC_VD_ADD;
VEXC_VWB_INDEX <= VEXM_VEXC_INDEX;
VEXC_VWB_RES <= data_out;//vexc_res_val;
VEXC_VWB_DATAIN <= data_in;
VEXC_VWB_MEM_WRT <= VEXM_VEXC_MEM_WRITE;
VEXC_VWB_MEM_READ <= VEXM_VEXC_MEM_READ;
end

//  Vetor Lane - Vector Register Write back  //

always @(posedge clock) begin
if (vexc_reg_wrt == 1) begin
if (VEXM_VEXC_MEM_READ == 1) begin
VECREG[VEXM_VEXC_VD_ADD][VEXM_VEXC_INDEX] <= data_in;
end
else begin
VECREG[VEXM_VEXC_VD_ADD][VEXM_VEXC_INDEX] <= data_out;
end
end
end
end
endmodule // Vector_Lane Module
```

Appendix E

Measurements on Cycle-accurate VeMICry

p: number of elements per vector register.

r: number of lanes.

Area of vector co-processor in μm^2 for $q = 4$								
r	p	1	2	4	8	16	32	64
1		30586	34410	41820	57280	87474	144999	263565
2		□	56409	64052	78889	108375	166976	285364
4		□	□	106252	121272	150918	211918	329106
8		□	□	□	207723	237787	297070	418940
16		□	□	□	□	410831	471026	589579
32		□	□	□	□	□	817523	937961
64		□	□	□	□	□	□	1626447

Mean power of vector co-processor in mW for $q = 4$								
r	p	1	2	4	8	16	32	64
1		7.7	9.4	11.5	17.0	27.0	68.0	□
2		□	14.0	15.8	21.0	31.0	52.0	□
4		□	□	25.0	27.0	40.5	61.0	□
8		□	□	□	45.0	58.0	77.0	□
16		□	□	□	□	90	144	□
32		□	□	□	□	□	185	□
64		□	□	□	□	□	□	□

E. MEASUREMENTS ON CYCLE-ACCURATE VEMICRY

Performance of 512-bit Modular Multiplication in <i>clock – cycles</i>								
r	p	1	2	4	8	16	32	64
1		□	6236	5450	5156	5207	9847	19127
2		□	4275	3328	2968	2887	5207	9847
4		□	□	2331	1874	1727	2887	5207
8		□	□	□	1359	1147	1727	2887
16		□	□	□	□	873	1147	1727
32		□	□	□	□	□	873	1147
64		□	□	□	□	□	□	873

Performance of 1024-bit Modular Multiplication in <i>clock – cycles</i>								
r	p	1	2	4	8	16	32	64
1		31666	23972	20734	19310	18988	19607	38103
2		□	16211	12396	10842	10260	10359	19607
4		□	□	8483	6608	5896	5735	10359
8		□	□	□	4619	3714	3423	5735
16		□	□	□	□	2687	2267	3423
32		□	□	□	□	□	1721	2267
64		□	□	□	□	□	□	1721

Performance of 2048-bit Modular Multiplication in <i>clock – cycles</i>								
r	p	1	2	4	8	16	32	64
1		□	94004	80870	74690	72374	72764	76055
2		□	63123	47812	41374	38542	37900	39127
4		□	□	32307	24716	21626	20468	20663
8		□	□	□	16899	13168	11752	11431
16		□	□	□	□	9195	7394	6815
32		□	□	□	□	□	5343	4507
64		□	□	□	□	□	□	3417

References

- ACHIICMEZ, O., ÇETIN KAYA KOÇ & SEIFERT, J.P. (2006a). On the Power of Simple Branch Prediction Analysis. *Cryptology ePrint Archive* (<http://eprint.iacr.org/>), **Report 2006/351**. 29, 46
- ACHIICMEZ, O., ÇETIN KAYA KOÇ & SEIFERT, J.P. (2006b). Predicting Secret Keys via Branch Prediction. *Cryptology ePrint Archive* (<http://eprint.iacr.org/>), **Report 2006/288**. 29, 46
- AKKAR, M.L. (2004). *Attaques et Méthodes de Protections de Systèmes cryptographiques embarqués*. Thèse de doctorat, Université de Versailles Saint-Quentin-en-Yvelines. 31, 122, 125
- AKKAR, M.L. & GIRAUD, C. (2001). An Implementation of DES and AES, Secure against Some Attacks. In Çetin Koç, D. Naccache & C. Paar, eds., *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, vol. 2162 of LNCS, 309–318, Springer-Verlag, Paris, France. 31, 48, 83
- ANDERSON, R. & KUHN, M. (1996). Tamper resistance - a cautionary note. In *Second USENIX Workshop on electronic commerce*, 1–11. 27, 30
- ANDERSON, R. & KUHN, M. (1997). Low Cost Attacks on Tamper-Resistant Devices. In M. Lomas & al, eds., *Security Protocols 5th International Workshop*, vol. LNCS, 125–136, Springer-Verlag, Paris, France. 27, 30
- ANSI (1997). Public Key Cryptography for the Financial Services Industry: The ECDSA. Tech. Rep. ANSI X9.62-199x, American Bankers Association. 61
- AOKI, K., HOSHINO, F., KOBAYASHI, T. & OGURO, H. (2001). Elliptic Curve Arithmetic Using SIMD. In *ISC '01: Proceedings of the 4th International Conference on Information Security*, 235–247, London, UK. 40
- ARCHC (2003). Retargeting GCC to ArchC models - Mini-Howto. Tech. Rep. v1.2, LSC - University of Campinas. 81
- ARCHC (2004). The Archc Architecture Description Language - Reference Manual. Tech. Rep. v1.2, University of Campinas. 81, 130
- ARM (2002). http://www.arm.com/products/CPUs/SecurCore_SC200.html. 34, 131
- ASANOVIĆ, K. (1998). *Vector Microprocessors*. Ph.D. thesis, University of California, Berkeley. 37, 39
- BATINA, L., ORS, S.B., PRENEEL, B. & VANDEWALLE, J. (2003). Hardware Architectures for Public Key Cryptography. *Integration, the VLSI Journal*, 34, 1–64. 34

REFERENCES

- BELGIANID (2006). <http://eid.belgium.be/en/navigation/12000/index.html>. 20
- BERNSTEIN, D.J. (2004). Cache timing attacks on AES. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>. 46
- BERTONI, G., BREVEGLIERI, L., KOREN, I., MAISTRI, P. & PIURI, V. (2002). A parity code based fault detection for an implementation of the advanced encryption standard. In *DFT '02: Proceedings of the 17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, 51–59, IEEE Computer Society, Washington, DC, USA. 30, 48
- BERTONI, G., ZACCARIA, V., BREVEGLIERI, L., MONCHIERO, M. & PALERMO, G. (2005). Aes Power Attack Based on Induced Cache Miss and Countermeasure. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, 586–591, IEEE Computer Society, Washington, DC, USA. 46
- BHASKAR, R., DUBEY, P.K., KUMAR, V. & RUDRA, A. (2003). Efficient Galois Field Arithmetic on SIMD Architectures. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, 256–257, ACM Press, New York, NY, USA. 40
- BIHAM, E. & SHAMIR, A. (1997). Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th International Advances in Cryptology Conference – CRYPTO'97*, vol. LNCS, 513–525. 29, 47
- BLAKE, I., SEROUSSI, G. & SMART, N. (1999). *Elliptic Curves in Cryptography*, vol. 265 of *Lecture Note Series*. London Mathematical Society. 24, 41, 60, 61, 63, 67
- BRIER, E., CLAVIER, C. & OLIVIER, F. (2003). Optimal Statistical Power Analysis. *Cryptology ePrint Archive* (<http://eprint.iacr.org/>), **Report 2003**. 28, 46, 51
- BRIER, E., CLAVIER, C. & OLIVIER, F. (2004). Correlation Power Analysis with a leakage model. In M. Joye & J.J. Quisquater, eds., *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, vol. 3156 of *Lecture Notes in Computer Science*, 16–29, Springer-Verlag. 28, 46
- BURGER, D. & AUSTIN, T.M. (2004). The SimpleScalar Tool Set. <http://www.simplescalar.com/>. 80
- CC (1998). Protection Profile Smartcard Integrated Circuit. *Common Criteria* (<http://www.crsc.nist.gov/cc>). 26, 46
- CC (1999). Common Criteria for Information Technology Security Evaluation. *Common Criteria*. 25, 46
- ÇETIN KOÇ, ACAR, T. & KALISKI, B.S. (1996). Analysing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16, 26–33. 84, 85, 116
- CLAVIER, C. & JOYE, M. (2001). Universal Exponentiation Algorithm: A First Step towards Provable SPA-resistance. In Çetin Koç, D. Naccache & C. Paar, eds., *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, vol. 2162 of *LNCS*, 300–308, Springer-Verlag, Paris, France. 31, 48, 122

- CLAVIER, C., CORON, J.S. & DABBOUS, N. (2000). Differential Power Analysis in the Presence of Hardware Countermeasures. In C. Koc & C. Paar, eds., *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'00)*, vol. LNCS, 252–263, Springer-Verlag, Worcester, USA. 31
- CORON, J.S. (1999). Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç.K. Koç & C. Paar, eds., *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, no. 1717 in Lecture Notes in Computer Science, 292–302, Springer-Verlag. 31
- CRANDALL, R. & KLIVINGTON, J. (1999). Vector Implementation of Multiprecision Arithmetic. Tech. rep., Apple Computers, Inc. 40, 131
- DHEM, J.F. (1998). *Design of an efficient public-key cryptographic library for RISC-based smart-cards*. Ph.D. thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium. 33, 45
- DHEM, J.F., KOENE, F., LEROUX, P.A., MESTRÉ, P., QUISQUATER, J.J. & WILLEMS, J.L. (1998). A Practical Implementation of the Timing Attack. In J.J. Quisquater & B.Schneier, eds., *Proceedings of the 3rd Smart-card Research and Advanced Application Conference (CARDIS'98)*, vol. LNCS, 167–182, Springer-Verlag, UCL, Louvain-la-Neuve, Belgium. 27, 46
- DIEFENDORFF, K., DUBEY, P., HOCHSPRUNG, R. & SCALES, H. (2000). AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 0272, 85–95. 40
- DIFFIE, W. & HELLMAN, M.E. (1976). New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22, 644–654. 24
- DIXON, B. & LENSTRA, A.K. (1992). Massively Parallel Elliptic Curve Factoring. In *EURO-CRYPT'92*, 183–193. 40
- EBERLE, H., SHANTZ, S., GUPTA, V., GURA, N., RARICK, L. & SPARCKLEN, L. (2005). Accelerating NEXT-Generation Public-Key Cryptosystems on General-Purpose CPUs. *IEEE Micro*, 0272-1732, 52–59. 35, 49
- ELDERSHAW, C. & BRENT, R.P. (1995). Factorisation of Large Integers on some Vector and Parallel Computers. Tech. rep., The Australian National University. 40
- EMV (2004). EMV Integrated Circuit Circuit Specifications for Payment Systems - Security and Key Management. Book 2 Version 4.1, EMVCo. 20
- ESAPASA, R., ARDANAZ, F., EMER, J., FELIX, S., GAGO, J., GRAMUT, R., HERNANDEZ, I., JUAN, T., LOWNEY, G., MATTINA, M. & SEZNEC, A. (2002). Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, vol. 1063 of *IEEE Computer Society*, IEEE. 39
- ESPASA, R., VALERO, M. & SMITH, J.E. (1998). Vector Architectures: Past, Present and Future. In *Proceedings of ICS 98*, 425–432. 38, 129
- FISCHER, W., GIRAUD, C., KNUDSEN, E.W. & SEIFERT, J.P. (2002). Parallel Scalar Multiplication on General Elliptic Curves over \mathbb{F}_p hedged against Non-Differential Side-Channel Attacks. *Cryptology ePrint Archive, Report 2002/007* (<http://eprint.iacr.org/>). 41

REFERENCES

- FISKIRAN, A.M. & LEE, R.B. (2005). On-Chip Lookup Tables for Fast Symmetric-Key Encryption. In *Proceedings of IEEE International Conference on Application Specific Systems Architectures and Processors (ASAP'05)*, 356–363, IEEE Press. [45](#)
- FLYNN, M.J. (1972). Some computer organizations and their effectiveness. *IEEE Transaction on Computers*, **C-21**, 948–960. [35](#)
- FLYNN, M.J. (1995). *Computer Architecture : Pipelined and Parallel Processor Design*. Jones and Bartlett. [35](#), [37](#), [38](#), [72](#), [133](#)
- FOLEGNANI, D. & GONZÁLEZ, A. (2001). Energy Effective Issue Logic. In *Proceedings of 28th Annual International Symposium on Computer Architecture 2001 (ISCA'2001)*, 230–239. [51](#), [129](#)
- FOURNIER, J. & MOORE, S. (2004). La carte à puce et les circuits asynchrones. *Numéro Spécial de la Revue de l'Electricité et de l'Electronique des Technologies de l'Information et de la Communication*. [18](#)
- FOURNIER, J. & TUNSTALL, M. (2006). Cache Based Power Analysis Attacks on AES. In L.M. Batten & R. Safavi-Naini, eds., *11th Australasian Conference on Information Security and Privacy — ACISP 2006*, vol. 4058 of *Lecture Notes in Computer Science*, 17–28, Springer-Verlag. [18](#), [29](#), [46](#)
- FOURNIER, J.J. & MOORE, S. (2006a). Hardware-Software Codesign of a Vector Co-processor for Public Key Cryptography. In V. Muthukumar, ed., *Proceedings of the ninth EuroMicro Conference on Digital System Design (DSD'06)*, 439–446, IEEE Computer Society, Cavtat, Croatia. [18](#)
- FOURNIER, J.J. & MOORE, S. (2006b). A Vector approach to Cryptography Implementation. In R. Safavi-Naini & M. Yung, eds., *Proceedings of 1st International Conference on Digital Rights Management - Technologies, Issues, Challenges and Systems (DRMTICS'2005)*, vol. LNCS, 277–297, Springer-Verlag Berlin Heidelberg 2006. [17](#)
- FOURNIER, J.J., MOORE, S., LI, H., MULLINS, R. & TAYLOR, G. (2003). Security Evaluation of Asynchronous Circuits. In C. Walter & al., eds., *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, vol. LNCS, 137–151, Springer-Verlag, Cologne, Germany. [18](#), [29](#), [30](#), [47](#)
- GAJ, K. & CHODOWIEC, P. (2002). Comparison of the hardware performance of the AES candidates using reconfigurable hardware. <http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/papers/22-kgaj.pdf> /. [33](#)
- GANDOLFI, K., MOURTEL, C. & OLIVIER, F. (2001). Electromagnetic Analysis: Concrete Results. In Çetin Koç, D. Naccache & C. Paar, eds., *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, vol. 2162 of LNCS, 251–261, Springer-Verlag, Paris, France. [28](#)
- GNU (2005). <http://gcc.gnu.org/>. [99](#)
- GROSSSCHÄDL, J. & KAMENDJE, G. (2003). Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields $GF(2^m)$. In *Proceedings of IEEE International Conference on Application Specific Systems Architectures and Processors (ASAP'03)*, 455–468. [35](#), [49](#), [52](#)

- GROSSCHÄDL, J. & SAVAS, E. (2004). Instruction Set Extensions for Fast Arithmetic in Finite Fields $GF(p)$ and $GF(2^m)$. In M. Joye & J.J. Quisquater, eds., *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, vol. 3156 of *Lecture Notes in Computer Science*, 133–147, Springer-Verlag. 49
- GROSSCHÄDL, J., KUMAR, S.S. & PAAR, C. (2004). Architectural Support for Arithmetic in Optimal Extension Fields. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, 111–124, IEEE Computer Society, Washington, DC, USA. 49
- GUTUB, A., SAVAŞ, E., TENCA, A.F. & ÇETIN K. KOÇ (2003). Scalable and unified hardware to compute montgomery inverse in $GF(p)$ and $GF(2^n)$. In C. Walter & al., eds., *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, no. 2779 in LNCS, 484–499, Springer-Verlag, Cologne, Germany. 35
- HANDSCHUH, H. & PAILLIER, P. (2000). Smart Card Crypto-Coprocessors for Public-Key Cryptography. In *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, 372–379, Springer-Verlag, London, UK. 33, 131
- HANKERSON, D., HERNANDEZ, J.L. & MENEZES, A. (2000). Software Implementation of Elliptic Curve Cryptography over Binary Fields. In C. Koc & C. Paar, eds., *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'00)*, no. 1965 in *Lecture Notes in Computer Science*, 1–24, Springer-Verlag, Worcester, USA. 63
- HENNESSY, J.L. & PATTERSON, D.A. (2003). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edn. 37, 100, 108, 133
- INFINEON (2003). Security & Chip Cards ICs SLE88CGFx4000P. Preliminary Short Product Information 04.03, Infineon Technologies, St.-Martin-Strasse 76, D-81541 Mnchen. 33, 34, 131
- ITOH, K., TAKENAKA, M., TORII, N., TEMMA, S. & KURIHARA, Y. (1999). Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, 61–72, Springer-Verlag, London, UK. 40, 131
- IZU, T. & TAKAGI, T. (2002). Fast Elliptic Curve Multiplications with SIMD Operations. In *Fourth International Conference on Information and Communications Security (ICICS'02)*, no. 2513 in LNCS, 217–230. 41
- JOYE, M. & TYMEN, C. (2001). Protections against Differential Analysis for Elliptic Curve Cryptography - An Algebraic Approach. In Çetin Koç, D. Naccache & C. Paar, eds., *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, vol. 2162 of LNCS, 377–390, Springer-Verlag, Paris, France. 62
- KARRI, R., KUZNETSOV, G. & GOESSEL, M. (2003). Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers. In C. Walter & al., eds., *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, no. 2779 in LNCS, 113–124, Springer-Verlag, Cologne, Germany. 30
- KOÇ, C. & ACAR, T. (1998). Montgomery Multiplication in $GF(2^m)$. *Designs, Codes and Cryptography*, 14, 57–69. 64, 65

REFERENCES

- KOCHER, P. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings of Advances in Cryptology (CRYPTO'96)*, LNCS, 104–113, Springer-Verlag. 27, 31, 46
- KOCHER, P., JAFFE, J. & JUN, B. (1999). Differential Power Analysis. In *Proceedings of the 19th International Advances in Cryptology Conference (CRYPTO'99)*, no. 1666 in LNCS, 388–397, Springer-Verlag. 28
- KÖMMERLING, O. & KUHN, M.G. (1999). Design Principles for Tamper-Resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology, Smartcard'99*, vol. ISBN, 9–20. 30, 33
- KOZYRAKIS, C. & PATTERSON, D. (2002). Vector vs Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, vol. 1072 of *IEEE Computer Society*, IEEE. 39
- KRAHINSKY, R., BATTEN, C., HAMPTON, M., GERDING, S., PHARRIS, B., CASPER, J. & ASANOVIĆ, K. (2004). The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*, vol. 1063. 39
- KULIKOWSKI, K., SMIRNOV, A. & TAUBIN, A. (2006). Automated design of cryptographic devices resistant to multiple side-channel attacks. In L. Goubin & M. Matsui, eds., *Proceedings of 8th Workshop on Cryptographic Hardware and Embedded Systems (CHES'06)*, no. 4249 in LNCS, Springer-Verlag, Yokohama, Japan. 30
- KUMAR, S.S. & PAAR, C. (2004). Reconfigurable Instruction Set Extension for Enabling ECC on an 8-bit Processor. In *FPL*, 586–595. 49
- KUO, H. & VERBAUWHEDE, I. (2001). Architectural Optimization for a 1.82 Gbits/sec VLSI Implementation of the AES Rijndael Algorithm. In Çetin Koç, D. Naccache & C. Paar, eds., *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, vol. 2162 of *LNCS*, 51–64, Springer-Verlag, Paris, France. 48
- LEADBITTER, P., PAGE, D. & SMART, N. (2007). Nondeterministic multithreading. *IEEE Transactions on Computers*, 56, 992–998. 47
- LENSTRA, A.K. & VERHEUL, E.R. (2001). Selecting Cryptographic Key Sizes. *Journal of Cryptology*, 14, 255–293. 84, 86
- LI, H., MARKETOS, T. & MOORE, S. (2005). Security Evaluation Against Electromagnetic Analysis at Design Time. In B.Sunar & J.Rao, eds., *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05)*, no. 3659 in LNCS, 280–292, Springer-Verlag, Edinburgh, Scotland. 30
- LOPEZ, J. & DAHAB, R. (1999). Fast Multiplication on Elliptic Curves over $GF(2^m)$ without precomputation. In Ç.K. Koç & C. Paar, eds., *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*, no. 1717 in LNCS, 316–327, Springer-Verlag. 63
- MAY, D., MULLER, H.L. & SMART, N.P. (2001). Non-deterministic Processors. In *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, 115–129, Springer-Verlag, London, UK. 47

- MCGREGOR, J.P. & LEE, R. (2003). Architectural techniques for accelerating subword permutations with repetitions. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, **11**, 325–335. [45](#), [49](#)
- MENEZES, A., OORSCHOT, P.V. & VANSTONE, S. (1997). *Handbook of Applied Cryptography*. CRC Press LLC. [22](#), [23](#), [24](#)
- MESSERGES, T., DABBISH, E. & SLOAN, R. (1999). Power Analysis Attacks of Modular Exponentiation in Smartcards. In Ç.K. Koç & C. Paar, eds., *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*, no. 1717 in LNCS, 144–157, Springer-Verlag. [31](#), [122](#)
- MIPS (1996). MIPSpro™ Assembly Language – Programmer’s Guide. Tech. Rep. 007-2418-001, Silicon Graphics Inc. [53](#), [95](#)
- MIPS (2001a). MIPS™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture. Tech. Rep. MD00082, Revision 0.95, MIPS Technologies, 1225 Charleston Road, Mountain View, CA 94043-1353. [51](#)
- MIPS (2001b). MIPS™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set. Tech. Rep. MD00086, Revision 0.95, MIPS Technologies, 1225 Charleston Road, Mountain View, CA 94043-1353. [51](#), [53](#)
- MIPS (2001c). MIPS™ Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture. Tech. Rep. MD00090, Revision 0.95, MIPS Technologies, 1225 Charleston Road, Mountain View, CA 94043-1353. [51](#)
- MIPS (2005). http://www.mips.com/content/products/cores/32-bit_cores/MIPS32_4KS_Family.php#specifications. [33](#), [34](#), [35](#), [49](#), [131](#)
- MONTGOMERY, P. (1985). Modular Multiplication without Trial Division. *Mathematics of Computation*, **44**, 519–521. [41](#), [45](#), [64](#), [84](#), [116](#)
- MOORE, S., ANDERSON, R., CUNNINGHAM, P., MULLINS, R. & TAYLOR, G. (2002). Improving Smart Card Security using Self-timed Circuits. In *Proceedings of 8th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC’ 02*, vol. IEEE, 23–58. [28](#), [29](#), [30](#), [47](#)
- MOORE, S., ANDERSON, R., MULLINS, R., TAYLOR, G. & FOURNIER, J. (2003). Balanced self-checking asynchronous logic for smart card applications. *Microprocessors and Microsystems Journal (IEEE)*, **27**, 421–430. [18](#), [28](#), [30](#)
- NACCACHE, D. & M’RAIHI, D. (1996). Arithmetic Co-processors for Public-key Cryptography: The State of the Art. In P.H.Hartel, P.Paradinas & J.J.Quisquater, eds., *Proceedings of the second International Conference on Smart Card Research and Applications (CARDIS’96)*, 39–58, Amsterdam, The Netherlands. [33](#), [48](#), [131](#)
- NIST (1993). Data Encryption Standard (DES). Tech. Rep. FIPS PUB 46-2, Federal Information Processing Standards. [23](#), [45](#)
- NIST (2001a). Security Requirements for Cryptographic Modules. Tech. Rep. FIPS PUB 140-2, Federal Information Processing Standards. [26](#), [46](#)

REFERENCES

- NIST (2001b). Specification for the Advanced Encryption Standard. Tech. Rep. FIPS PUB 197, Federal Information Processing Standards. [23](#), [40](#), [45](#), [52](#)
- NIST (2002). Secure Hash Standard. Tech. Rep. FIPS PUB 180-2, Federal Information Processing Standards. [22](#)
- OIKONOMAKOS, P., FOURNIER, J. & MOORE, S. (2006). Implementing Cryptography on TFT Technology for Secure Display Applications. In P. Paradinas, Y. Deswarte & A.E. Kalam, eds., *Proceedings of the seventh Smart Card Research and Advanced Applications IFIP Conference (CARDIS'06)*, vol. 3928 of *Lecture Notes in Computer Science*, 32–47, Springer Berlin, Tarragona, Spain. [18](#)
- OMA (2005). DRM Specification V2.0 Candidate Version 2.0 - 26 April 2005. Tech. Rep. OMA-DRM-DRM-V2_0-20050426-C, Open Mobile Alliance (OMA). [21](#)
- OPENSSL (2000). Cryptographic libraries for OpenSSL. <http://www.openssl.org/docs/crypto/crypto.html>. [22](#), [44](#)
- PADEGS, A., MOORE, B.B., SMITH, R.M. & BUCHHOLZ, W. (1988). The IBM System/370 Vector Architecture: Design Considerations. *IEEE Transactions on Computers*, **37**, 509–520. [39](#)
- PADUA, D.A. & WOLFE, M.J. (1986). Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, **29**, 1184–1201. [39](#)
- PAGE, D. (2002). Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *Cryptology ePrint Archive* (<http://eprint.iacr.org/>), **Report 2002/169**. [46](#)
- PAGE, D. (2004). Defending against cache based side-channel attacks. *Information Security Technical Report*, **8**, 30–44. [46](#)
- PAGE, D. & SMART, N.P. (2004). Parallel Cryptographic Arithmetic using a Redundant Montgomery Representation. *IEEE Transaction on Computers*, **53**, 1474–1482. [40](#)
- PELEG, A. & WEISER, U. (1996). MMX technology extension to the Intel architecture. *IEEE Micro*, **16**, 42–50. [39](#), [129](#)
- PHILIPS (2004). SmartMX. <http://www.nxp.com/products/identification/smartmx/index.html>. [34](#)
- QUISQUATER, J.J. & SAMYDE, D. (2001). Electromagnetic Analysis (EMA): Measures and countermeasures for smart cards. *Smart Card Programming and Security*, **LNCS**, 200–210. [28](#)
- RANKL, W. & EFFING, W. (2000). *Smart Card Handbook*. [32](#)
- RIVEST, R.L., SHAMIR, A. & ADLEMAN, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, **21**, 120–126. [24](#), [84](#)
- RSA (2000). PKCS-11v2 Specifications. <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/>. [44](#)
- RUSSELL, R.M. (1978). The CRAY-1 Computer System. *Communications of the ACM*, **21**, 63–72. [39](#), [129](#)

- SAKIYAMA, K., BATINA, L., PRENEEL, B. & VERBAUWHEDE, I. (2006a). HW/SW Co-design for Accelerating Public-Key Cryptosystems over GF(p) on the 8051 μ -controller. In *Proceedings of IFMIP-WAC 2006, Special Sessions on Information Security and Hardware Implementations*. 41
- SAKIYAMA, K., BATINA, L., PRENEEL, B. & VERBAUWHEDE, I. (2006b). Superscalar Coprocessor for High-speed Curve-based Cryptography. No. 4249 in *Lecture Notes in Computer Science*, 415–429. 41
- SAVAŞ, E., TENCA, A.F. & ÇETIN K. KOÇ (2000). A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m). In C. Koc & C. Paar, eds., *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'00)*, vol. LNCS, 277–292, Springer-Verlag, Worcester, USA. 34, 35, 45
- SCHNEIER, B. (1995). *Applied Cryptography: Protocols, Algorithms and Source Code in C (2nd Edition)*. John Wiley and Sons. 22
- SHI, Z., YANG, X. & LEE, R. (2003). Arbitrary bit permutations in one or two cycles. In *Proceedings of the IEEE 14th International Conference on Application-Specific Systems Architectures and Processors ASAP'2003*. 45
- SINGH, S. (2000). *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books (Random House), New York, USA, first anchor books edition. 22
- SINKOV, A. (1966). *Elementary Cryptanalysis: A Mathematical Approach*. Mathematical Association of America. 27
- SKOROBOGATOV, S. & ANDERSON, R. (2002). Optical Fault Induction Attacks. In B. Kaliski & al., eds., *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, vol. LNCS, 2–12, Springer-Verlag. 29
- TENCA, A.F. & ÇETIN K. KOÇ (1999). A Scalable Architecture for Montgomery Multiplication. In Ç.K. Koç & C. Paar, eds., *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*, vol. LNCS, 94–108, Springer-Verlag. 34, 45, 84
- TI (2004). TMS320C6201 Fixed Point Digital Signal Processors. <http://focus.ti.com/lit/ds/symlink/tms320c6201.pdf>. 41
- TILlich, S. & GROSSCHÄDL, J. (2005). Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography. In *ICCSA (2)*, 665–675. 49
- TILlich, S. & GROSSCHÄDL, J. (2006). Instruction Set Extension for Efficient AES Implementation on 32-bit Processors. In L. Goubin & M. Matsui, eds., *Proceedings of 8th Workshop on Cryptographic Hardware and Embedded Systems (CHES'06)*, vol. LNCS, Springer-Verlag, Yokohama, Japan. 49
- TIRI, K. & VERBAUWHEDE, I. (2003). Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card Technology. In C. Walter & al., eds., *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, no. 2779 in LNCS, 125–136, Springer-Verlag, Cologne, Germany. 30

REFERENCES

- TPG (2006). The Trusted Computing Group Mobile Specification: Securing Mobile Devices on Converged Networks. White paper, Mobile Phone Work Group, Trusted Computing Group. 20
- TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M. & MIYAUCHI, H. (2003). Cryptanalysis of DES Implemented on Computers with Cache. In C. Walter & al., eds., *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, no. 2779 in LNCS, 62–76, Springer-Verlag, Cologne, Germany. 46
- WALKER, J.F. & ALIBHAI-SANGHRAJKA, A. (2004). Using FIB to hack security chips. *European Focused Ion Beam Users Group (EFUG 2004) presentation*, <http://www.imec.be/efug/EFUG20h.html>, siVenture, Maidenhead, UK. 27
- ZAMBRENO, J., CHOUDHARY, A., SIMHA, R. & NARARI, B. (2004). Flexible Software Protection Using Hardware/Software Codesign Techniques. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, vol. 01, 10636. 48, 122