TECHNICAL UNIVERSITY OF CRETE

Electronics and Computer Engineering Department

Microprocessor and Hardware Laboratory

Master Thesis

"CCproc: A custom VLIW cryptography co-processor for symmetric key ciphers"

Theodoropoulos Dimitris

Advisor

Associate Prof. Dionisis Pnevmatikatos

Examination Committee

Associate Prof. Dionisis Pnevmatikatos

Prof. George Stavrakakis

Assistant Prof. Matthias Bucher

Acknowledgements

First of all, I would like to thank my advisor associate Prof. Dionisis Pnevmatikatos for all his support and useful advices through this project's elaboration, the other two members of the examination committee, Prof. George Stavrakakis and assistant Prof. Matthias Bucher, and also Mr. Markos Kimionis.

Also, from this point, I would like to thank all my friends with who, through these 7 years in Chania, I have shared the best moments of my life so far:

Tikou Efi

Gikopoulos Loukas, Drougas Giannis, Kaklamanis Petros, Koidis Iosif, Konsolaki Maria-Nektaria, Kontaksaki Rena, Mouziouras Panagiotis, Mproustis Giannis, Potirakis Antonis, Sotiropoulos Stamatis and Strydis Christos.

Arhontaki Despina, Drakousi Aleksandra, Garbi Sofia, Katsoprinaki Stella, Kiratzi Irini, Ntegiannaki Maria, Papadimitriou Kiprianos, Sotiriadis Eyripidis and Voumvoulaki Eva.

To my Family

1. Introduction 6

2. Private-Key Block Ciphers Properties 10

2.1 Plaintext Encryption / Ciphertext Decryption Process 10

2.2 Structures and Arithmetic Operations in Symmetric Ciphers 12

3. Related Work 16

3.1 C and Assembly Implementations 16

3.2 Algorithm Specific Hardware Implementations 18

3.3 Symmetric Ciphers ISA extensions and Hardware Co-Processors 22

3.3.1 Symmetric cipher accelerators and ISA extensions 23

3.3.2 Hardware co-processors 23

4. CCproc Architecture 27

4.1 CCproc Design Considerations 27

4.2 CCproc Instruction Set Architecture 32

4.3 CCproc Datapath Structure 35

4.3.1 The Loop instruction controller circuit 35

4.3.2 Register File (RF) 37

4.3.3 Key Register File (KRF) 37

4.3.4 Arithmetic Logic Unit (ALU) 39

4.3.5 8-bit Galois Field (GF) Multiplier in GF(2^8) 42

4.3.6 32-bit Multiplier Modulo 2^{32} (MM) 44

4.3.7 Cipher Sboxes 45

4.3.8 Putting it all together: CCproc VLIW symmetric cipher co-processor 51

4.4 Efficient Data Exchange Among Clusters 56

5. Verification and Performance Evaluation of CCproc 59

5.1 Verification Tests Using a Python Assembler 59

5.2 Performance Evaluation on Xilinx Virtex 4 FPGA Devices 62

5.3 Performance Comparison with Other Implementations 63

6. Conclusions and Future Work 68

7. References 71

Appendix A: CCproc Complete Instruction Set 77

Appendix B: Setup and Usage of Assembler and CAD Tools 80

1. Introduction

It is generally accepted that, as years go by, internet usage grows rapidly. Current research indicates that in 2005 there are almost 1 billion internet users worldwide [I1]. Also, an older (2001) report from the "Stanford Institute for the Quantitative Study of Society" of Stanford University, focused on the ways that internet is being used and deduced the results which are shown in Chart 1 [I2].

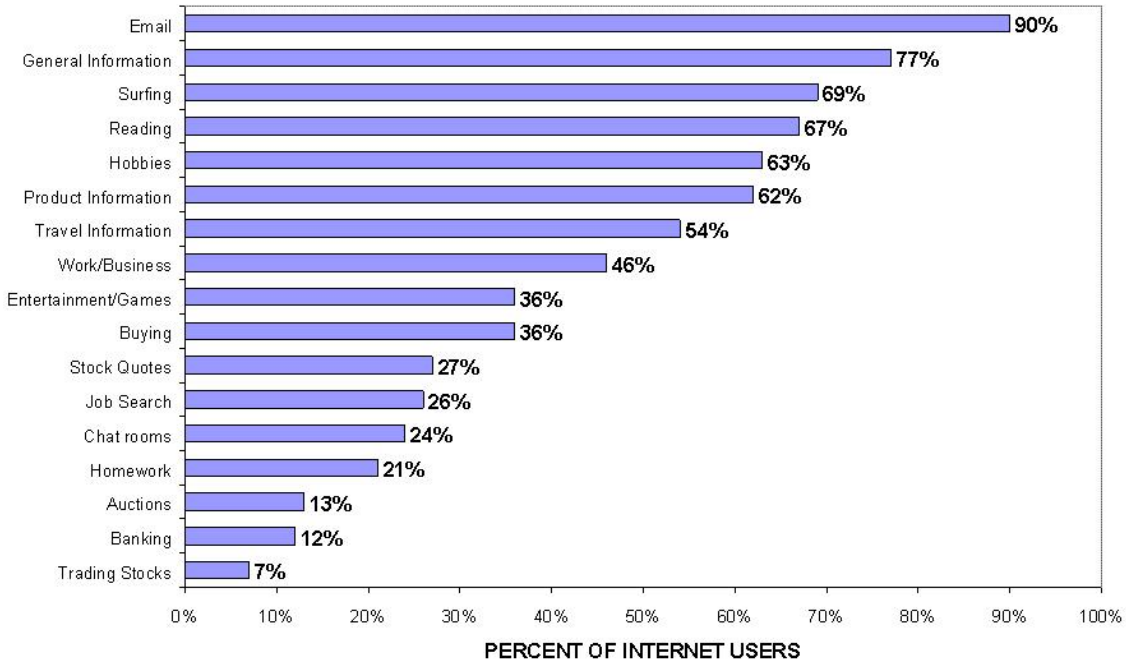

Chart 1 – What Internet users Do

Activities such as banking, buying, trading stocks, business and email, demand privacy and confidentiality during internet connection. For this reason protocols have been designed that create secure connections and protect data transmission from other malicious internet users. Among others, a well known such protocol, is the Secure Sockets Layer (SSL) [I3], which has been designed by Netscape and uses cryptography to protect data.

Cryptography comes from the Greek word "κρυπτογραφία" and means a way of altering a message with a secret key in a form that is almost impossible to recognise, but if the intended recipient has the appropriate secret key, then the original message can be retrieved. The process that a message is encrypted and decrypted using a specific key is called a cryptography algorithm or cipher. The first cipher in history was the "Caesar cipher" by Julius Caesar, which substitutes each letter in a message with the one that

corresponds to three places forward in the alphabet. An example is the word "SECRET", which, after the letter substitution, becomes "VHFUHW". Another very important historical example, based on the Caesar Cipher, is the "Enigma cipher" which was designed by Germans during the World War II [I4].

Today there are many ciphers and mostly are used to protect sensitive information during transmission on public communication networks. The entire process is consisted from two basic parts, the *encryption* and *decryption* of the message that is about to be transmitted (*plaintext*). During the encryption process the plaintext is transformed to another text (*ciphertext*) and the latter is transmitted. Once the transmission is over, the ciphertext is again transformed back to its original form, i.e. the plaintext, so the recipient is able to recognise it.

Two forms of cryptography are commonly used in information systems today, which are shown in Figure 1:

- Symmetric Key or Private Key ciphers.
- Asymmetric Key or Public Key ciphers.

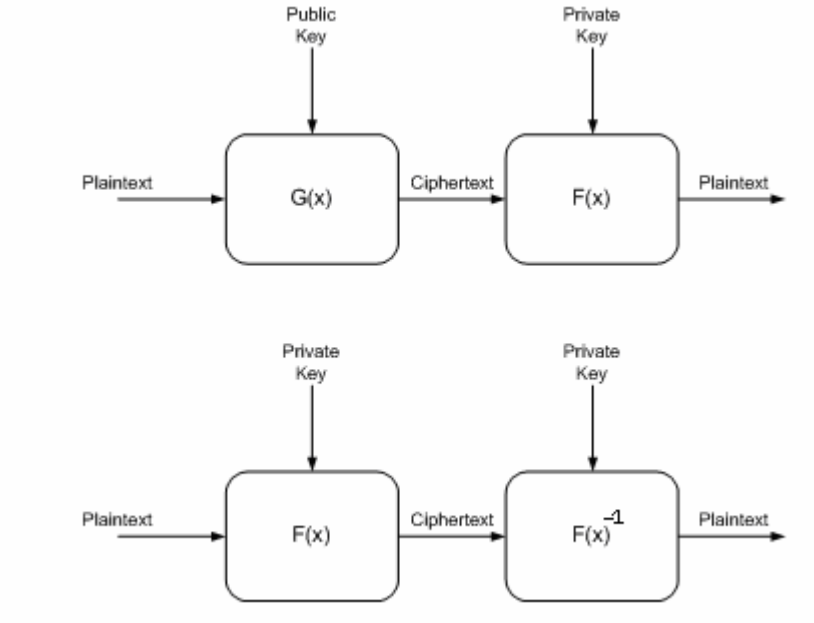

Figure 1 - Public-key (up) and Symmetric-key (down) algorithms

As it is shown from Figure 1, public key ciphers, use two types of keys, a public that is used to encrypt data, and a private that is used to decrypt the encrypted data. On the other hand, private key ciphers use only a private key for both data encryption and decryption. Another fact is that private key ciphers are much faster than public key

ciphers [I5], so during a secure data exchange, at first a private key is shared between the users with a public key cipher and then all other data are being transmitted with a private key cipher, that uses the previous private key for encryption / decryption.

More specifically, the entire process of a secure communication channel establishment is as follows:

1. Person *A* sends his public key to person *B* through an unsecured communication channel.
2. *B* encrypts its secret key with a public key cipher and sends it to *A*.
3. *A* decrypts the encrypted secret key with his private key.
4. From this moment all data are encrypted / decrypted with symmetric key ciphers.

However, there still are various ways to decipher encrypted communications without knowing the proper keys. Examples are *brute force* attacks, where all possible keys are being tried, *ciphertext-only* attacks, where the attacker tries to guess the plaintext with theoretical methods such linear and differential cryptanalysis [12], and *man-in-the-middle* attacks, where an adversary positions himself between A and B persons and intercepts each signal they send to each other [I5].

In this thesis we focus on the research of encryption and decryption process in many symmetric key ciphers, in order to find common processing parts among them and be able to design an ISA (Instruction Set Architecture) that effectively supports them. These similarities were deeply analyzed and the result is a hardware VLIW (Very Long Instruction Word) [13] co-processor called *CCproc* (Cryptography CoProcessor), with its own symmetric cipher specific instruction set and an extended RISC (Reduced Instruction Set Computer) datapath structure [13], capable to support many of today's symmetric key ciphers, functioning at very competitive speeds, plus also potential new ones.

The rest of this text is organized as follows: In chapter 2 we analyze the properties of symmetric ciphers, which includes the kinds of structures and arithmetic operations, plus analyzes the enciphering and deciphering process. Chapter 3 focuses on previous related work that was found to be done in software and hardware level. Chapter 4 describes the Instruction Set Architecture (ISA) and the datapath structure of CCproc. In

chapter 5 we show the verification process that was followed by running simulation tests. Also, we discuss the CCproc performance on Xilinx [I6] Virtex 4 Field Programmable Gate Arrays (FPGAs) devices [1] and compare it with other designs. Finally Chapter 6 offers conclusions of this thesis and shows which parts could be upgraded in possible future work.

In summary our design has the following characteristics:

- Efficient and flexible ISA capable to support many symmetric 128-bit ciphers
- 4-wide VLIW processor using 128-bit instructions with (Reduced Instruction Set Computer) RISC datapath structure
- Fits in small FPGAs, while multiple CCproc cores can be placed in larger ones to improve cipher performance
- Supports all Advanced Encryption Standard (AES) round two finalists
- Achieves an AES performance up to 616 Mbits/sec at 95 MHz in Interleaved Cipher Block Chaining (ICBC) mode using a 4-core CCproc implementation
- A 1-core CCproc VLSI implementation estimated running at 500 MHz, yields an AES throughput of 800 Mbits/sec
- Capable to saturate wide used protocols such as the 801.11g wireless and 802.3y 100 Mbits/sec Ethernet

2. Private-Key Block Ciphers Properties

In this chapter we analyze the properties of symmetric ciphers, which includes the kinds of arithmetic operations and analyzes the enciphering and deciphering process. Specific attention was paid to choose which algorithms to study, because many of them have weaknesses. So, in order to make our analysis as complete as possible, the following algorithms were chosen: Rijndael [2], MARS [3], Twofish [4], RC6 [5], Serpent [6], Blowfish [7], RC4 [I7], DES (Data Encryption Standard) [I8], RC5 [8], International Data Encryption Standard (IDEA) [9]. This group contains only the five AES (Advanced Encryption Standard) finalists of round 2 [I9], i.e. the strongest ones of the AES candidates. It also, has the previous standard encryption algorithm DES, plus Blowfish, IDEA, RC4, RC5 which are older and widely used. All these facts it is believed that led in a very realistic and representative choice of the best symmetric key ciphers ever designed, in order to proceed into further analysis.

2.1 Plaintext Encryption / Ciphertext Decryption Process

Every symmetric cipher has the following three important parameters:

1. The number of bits in its secret key.
2. The size of the data block that operates on (also in bits).
3. The number of processing rounds.

Depending on the size of data block, symmetric ciphers have two categories:

- Block ciphers that operate on large data blocks.
- Stream ciphers that operate usually on one bit.

Table 1 shows all these attributes for the ciphers mentioned before.

Figure 2 shows a generic schematic for the encryption / decryption process. Before message encryption starts, every symmetric cipher has an initialization phase, which is mainly the key expansion. More specifically, the secret key is processed in a certain way and the result is a number of other keys that some of them are used in different encipher / decipher rounds. In rare cases, also other required operations occur, such as in Blowfish, where its substitution boxes are being created.

Algorithm	Type	Key size (bits)	Block size (bits)
Blowfish	Block	up to 448	64
Twofish	Block	up to 256	128
DES	Block	64	64
Rijndael	Block	up to 256	128
MARS	Block	128 to 400	128
Serpent	Block	256	128
IDEA	Block	128	64
RC4	Stream	up to 2048	8
RC5	Block	up to 2040	>0
RC6	Block	up to 2040	>0

Table 1 – Symmetric ciphers categories

After the entire initialization phase is completed, encryption process begins. The latter consists of a certain number of various types of arithmetic operations that are being applied on the plaintext for a specific number of rounds. Once the defined round number has been reached, encryption process is finished and ciphertext is ready to be transmitted. Decryption process in most cases, if it is not identical, then it is almost the same as the encryption process, where again various types of arithmetic operations are being performed on ciphertext for a specific number of rounds, in order the recipient to retrieve the original message.

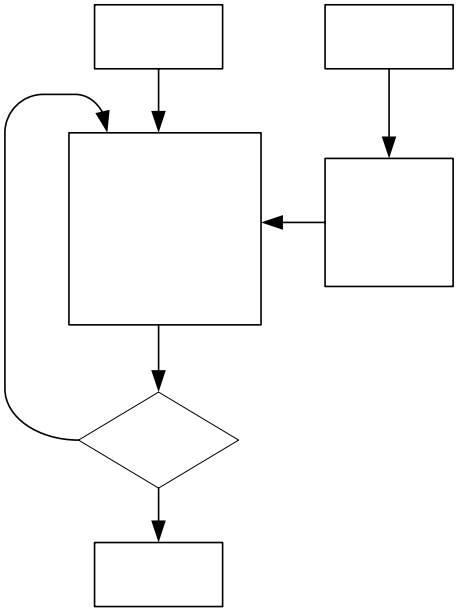

Figure 2 – Encryption / Decryption process

One final aspect of the symmetric key cipher is the operation mode [10], which describes how the entire message will be processed. The most commonly used modes are the ECB (Electronic Code Book), where each data block is encrypted separately and CBC (Cipher Block Chaining), where each data block is added modulo 2 (i.e. xor) to the previous encrypted data block. CBC offers much better security, but in ECB mode a message is encrypted in less time.

2.2 Structures and Arithmetic Operations in Symmetric Ciphers

Symmetric key ciphers are designed in such a way that it will as difficult to break as possible. In order to achieve the highest security level, designers have to consider, among others, the kind of arithmetic operations that will be used, the size of the data block and secret key, and the number of processing rounds.

Data block and key size affect the hardware resources that will be needed, mostly the number of registers and memory allocation. Key size also heavily contributes to the cipher's security level, because, when using brute force attack, the required computing power increases exponentially with it. Today an acceptable key size is at least 80-bit, while 128-bit will probably remain unbreakable by brute force attacks for the foreseeable future.

The number of rounds also affects considerably a cipher's security level, because, in each one of them, previous processed data get "scrambled" even more. It is on the designer's decision of how many total rounds a cipher will consist of. Fewer rounds mean lesser security, but on the other hand, quicker data block processing. As a result, the appropriate round number depends on the round's itself strength, i.e. the arithmetic operations that are applied to a data block in each one of them.

When the above ciphers had been designed, processors were still 32-bit and, consequently, most of the arithmetic operations are chosen to take advantage of it. Also, it is imperative that these operations present rapid bit diffusion, in order to increase the cipher's security. After deep analysis, it was concluded that the operations and structures most commonly used are:

1. Unsigned addition and subtraction modulo 2^{32}
2. Multiplication modulo 2^{32}
3. Exclusive or (xor) between 32-bit data

4. Fixed shifts and rotations
5. Data depended shifts and rotations
6. Finite field polynomial multiplication in 2^8 modulo a prime polynomial
7. Expansions and permutations (Xboxes)
8. Substitution boxes (Sboxes)
9. Feistel network structures[11]

In 32-bit processors, operations from 1 to 6 are implemented very fast, except from the finite field polynomial multiplication (FFM) modulo a prime polynomial, for which there is no efficient hardware support. The essential difference between a regular multiplication and FFM is that the first summarizes the partial products, while the second makes a XOR operation between them. However, symmetric ciphers perform FFM modulo a prime polynomial, which requires an additional division of the multiplication result with the value that represents the prime polynomial.

Additions, subtractions and XOR are the simplest operations, which are used to scramble data. Because they are very fast in software and hardware, they provide lesser security. However, they are used to isolate direct communication among other operations. Fixed rotations are mainly used in conjunction with software implementations to get specific data bits to places, from where they will be used by other operations. Data depended rotations can be performed quickly in software and hardware and if combined with arithmetic operations, such as addition, they are very effective against linear cryptanalysis. A problem is that rotation of a w-bit word depends on $\log_2 w$ bits, a fact that may lead into differential weaknesses of a cipher. However it can be bypassed if such an operation is combined with multiplication. In the latter the main cryptographic strength is the high order bits of the product, because they are almost depended on all operands bits in a non-linear fashion [3].

Besides arithmetic operations, there are common structures among ciphers. Sboxes are usually non-linear structures that map an n-bit value to an m-bit value, essentially Look Up Tables (LUT). A symmetric cipher may have one or more different Sboxes, with each one of them having arbitrary dimensions, as shown in Figure 3. Also the Sbox may even be the only non-linear part of the cipher.

Carefully chosen Sboxes can provide good resistance against linear and differential attacks, as well as good data and key bits avalanche. A drawback when using them is their relative slow software implementation. Also their index consists of a few bits (otherwise they would be too large), so they must deliberately be placed in a cipher.

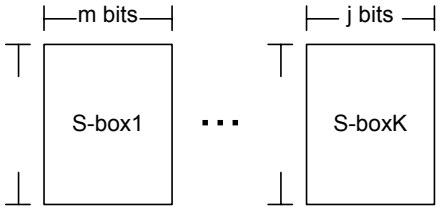

Figure 3 - Sboxes

Permutation is the structure where bits change place among each other, while in expansion, bits are also mixed but some of them appear more than once. They are linear operations, and thus not sufficient to guarantee security. However, when used with good non-linear Sboxes, they are vital for the security because they propagate the non-linearity uniformly over all bits.

Finally, Feistel network is the structure that most symmetric ciphers use, and combines all processing rounds with their inner operations [11]. As shown in Figure 4, plaintext is split into smaller blocks and one of them is passed through an F function with the combination of an expanded key K_i, where i, is the appropriate round number. After that, the result is xored with other blocks and, before next rounds initiates, a data block rotation occurs. The Feistel structure has the advantage that encryption and decryption operations are very similar, even identical in some cases, requiring only a reversal of the key schedule. Therefore the size of the code or circuitry required to implement such a cipher, is nearly halved.

n places

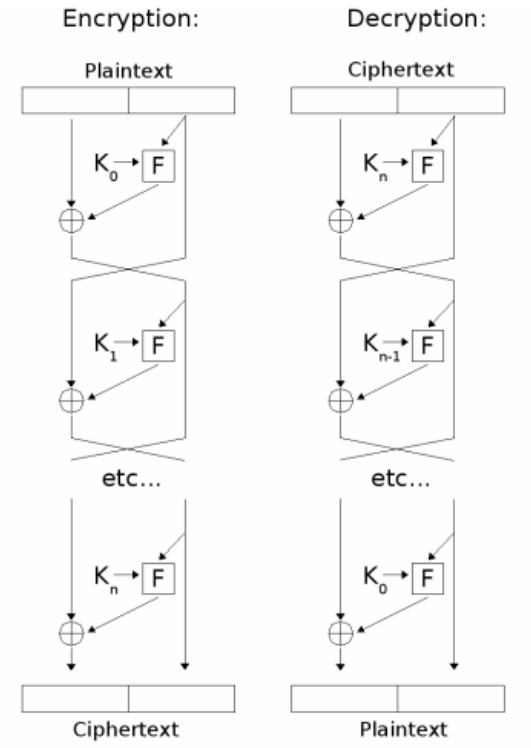

Figure 4 – Feistel network structure

3. Related Work

This chapter focuses on related work that has been done in software (assembly) and hardware level. As it is easily comprehended, there are vast implementations for all symmetric ciphers; however many older ones may be abandoned, because of their low security level. Also, the National Institute for Standards and Technology (NIST) exhaustively researched for three years for the five round two AES finalists among many AES candidates [I11]. These facts indicate that, from now on, these ciphers will be the most commonly used ones, a conclusion, which is confirmed from the fact that, most of the related work in symmetric algorithms, also focuses on these ciphers, as it is shown in the next sections.

3.1 C and Assembly Implementations

In this section we provide some of the best implementations for the five AES finalists reported so far in literacy, using assembly or the C software language, which can be found in [14], [22] and [I10]. Performance is measured in clock cycles, a metric which does not depend on processor's operating frequency with the same ISA. Chart 2 shows implementations for the Intel's Pentium Pro [16], Pentium II [17], Pentium III [18] families, plus Alpha 21164 [20] and Sun's SPARC processors [19], while Chart 3 for Pentium Pro, Pentium II, Pentium III families, plus Digital Equipment's Alpha 21164, Itanium 64 [15], and Precision Architecture (PA) RISC 8500 processors [21].

As it becomes clear from these charts, for every cipher on Pentium Pro, Pentium II, Pentium III families and Alpha 21164 processors, the relative assembly implementation is much faster than the C one. Exception is Serpent, where on Intel's processors the C implementation (759 cc) is slightly faster than in assembly (771 cc). Also, once the Rijndael cipher is the new AES standard, it was considered necessary to examine some additional more recent assembly and C implementations. In Chart 4 they are shown for all previous processors, plus Intel's Pentium 4 [23], Digital Equipment's Alpha 21264 [24], AMD's (Advanced Micro Devices) Athlon [25] and PowerMac G4 processors [I12].

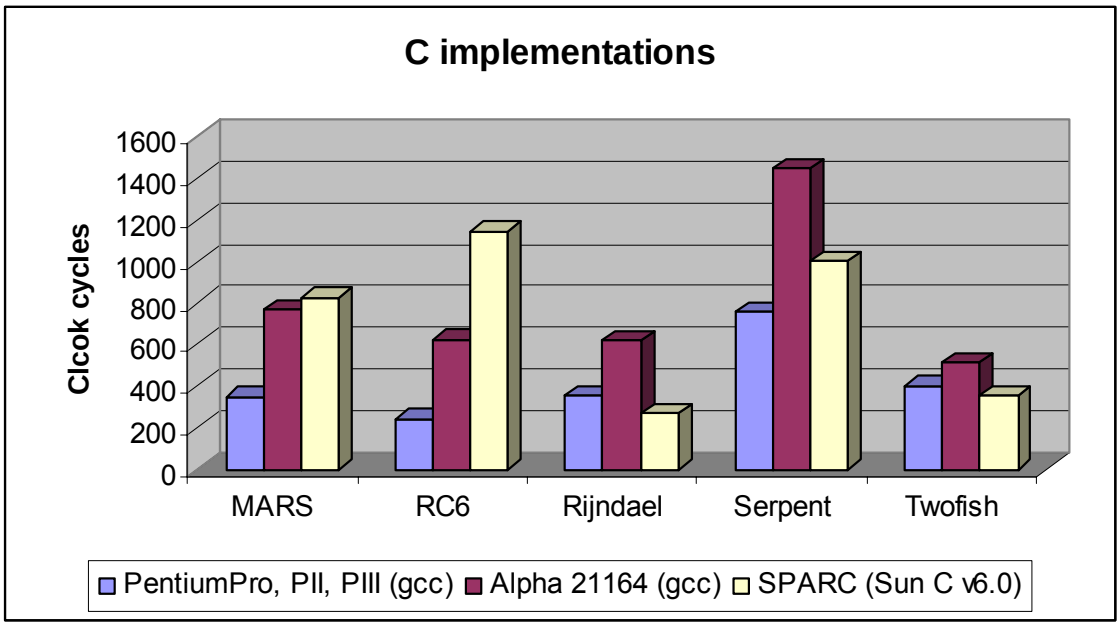

Chart 2 – C implementations

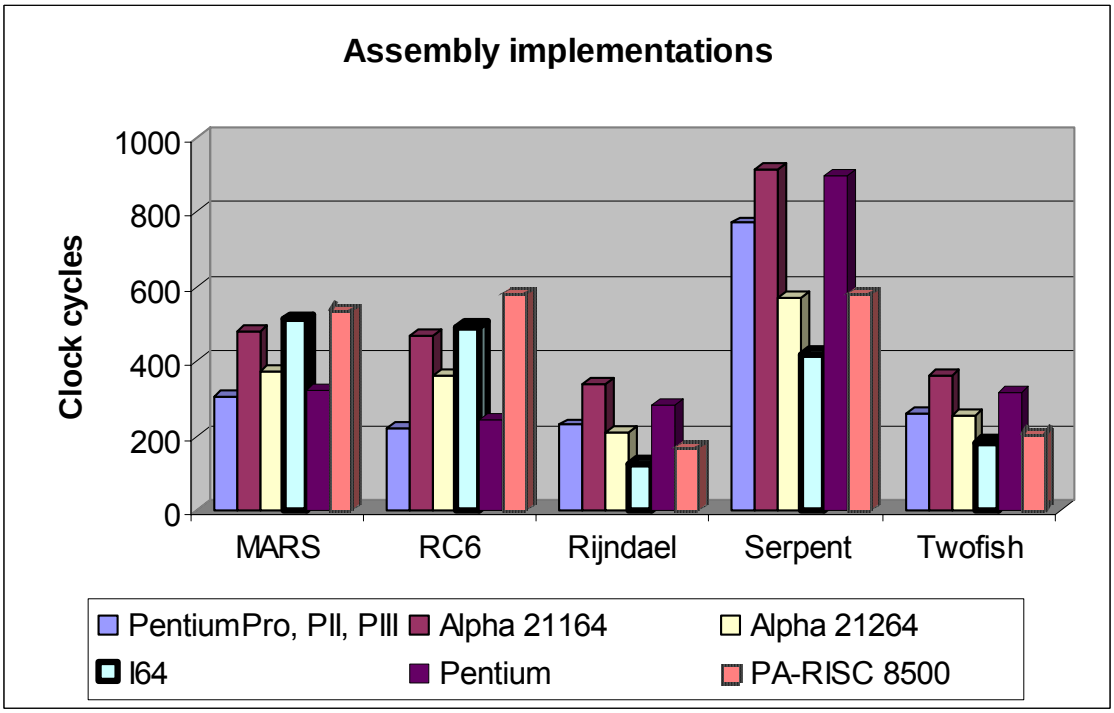

Chart 3 – Assembly implementations

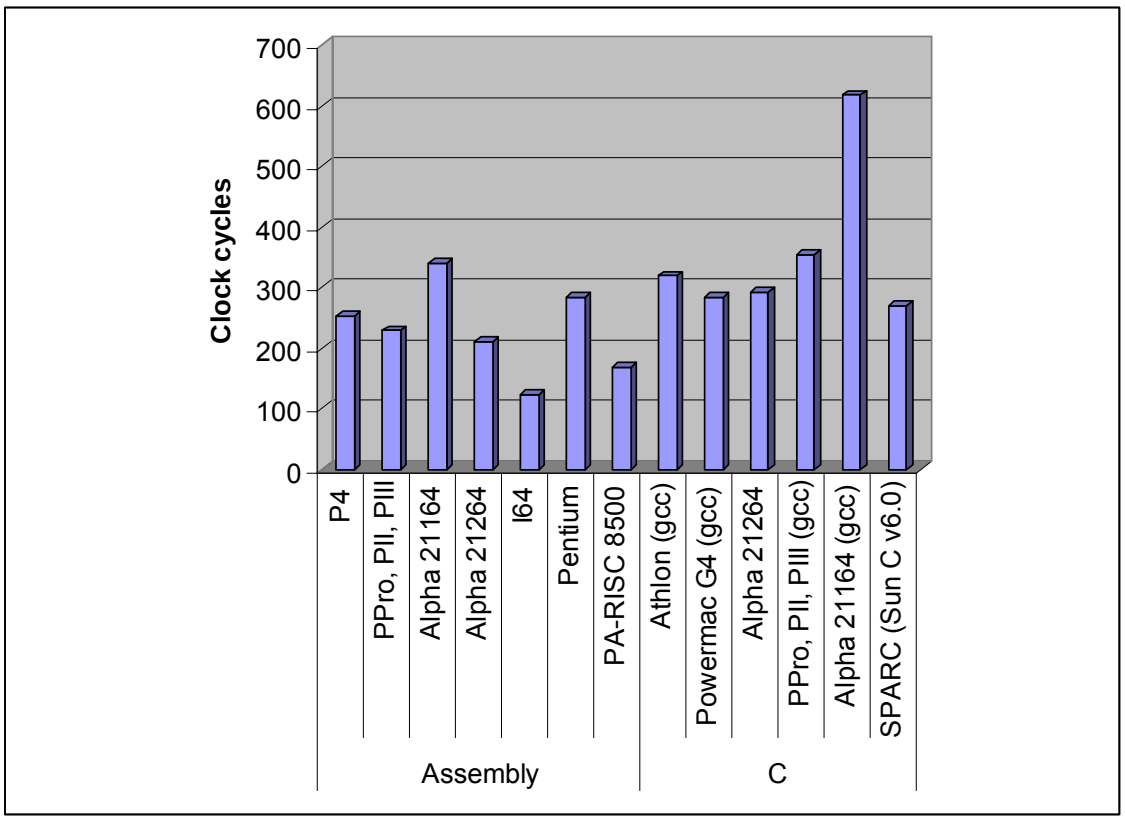

Chart 4 – Rijndael assembly and C implementations

3.2 Algorithm Specific Hardware Implementations

Besides software implementations, there are also many hardware specific ones, based on FPGA devices or ASICs (Application Specific Intergraded Circuits). This implementation category provides ultra speed performance (much higher than in software) for each symmetric algorithm, because of the dedicated hardware processors.

Few of the fastest AES implementations are the ones below, which are also summarized in Chart 5:

- Alireza Hodjat et al in [26] use a VirtexII Pro FPGA [27] and achieve a 21.4 Gbits/sec throughput with a latency of 31 cycles.
- Maire McLoone et al in [28] utilize LUTs in a VirtexE FPGA [29] to implement the entire encryption process, achieving a 12 Gbits/sec throughput.
- P. Chodowiec et al in [30] use a Virtex XCV1000 FPGA [31] and introduce the usage of pipeline stages inside of a cipher round, achieving a 12.1Gbits/sec throughput.

- Alireza Hodjat et al in [32] present a 0.18μm CMOS (Complementary Metal-Oxide Semiconductor) technology AES crypto coprocessor that runs at 330 MHz with a 3.84 Gbits/sec throughput.
- Sumio Morioka et al in [33] describe a 0.13μm CMOS technology AES IP (Intellectual Property) core that runs at 880 MHz with a 10Gbits/sec throughput.

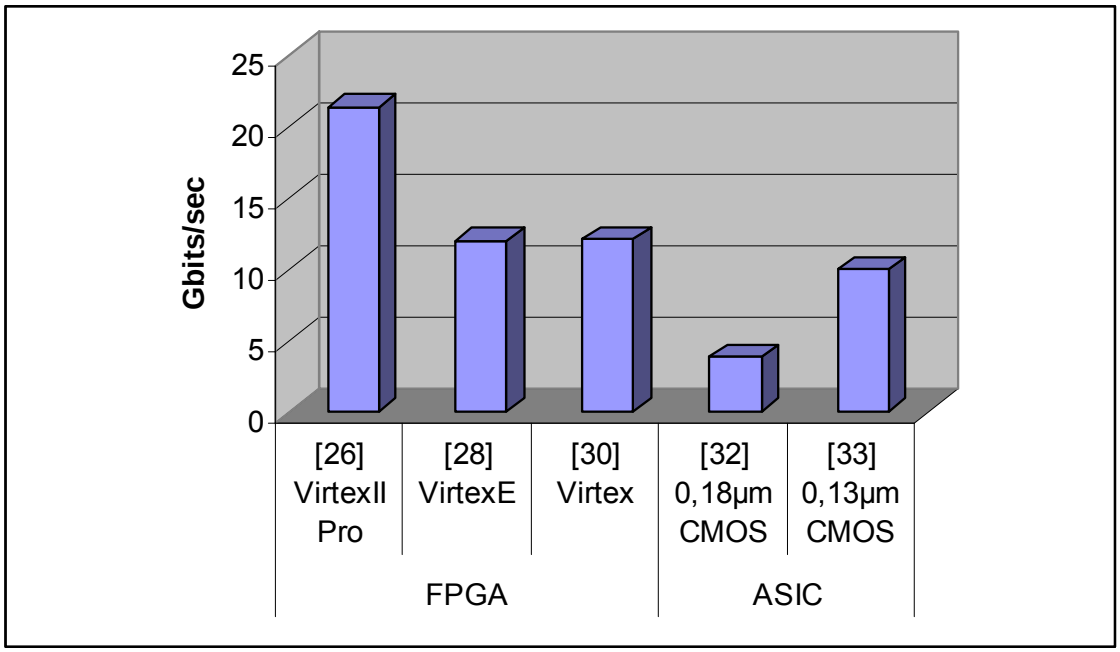

Chart 5 – AES hardware specific implementations

Few of the fastest RC6 implementations are the ones below, which are also summarized in Chart 6:

- Jean-Luc Beuchat et al in [34] use VirtexE and Virtex2 FPGAs, and achieve a maximum 15.2 Gbits/sec throughput.
- Elbirt et al in [35] use Virtex FPGAs to implement all AES finalists except MARS and the maximum achieved throughput is 2.4 Gbits/sec.
- Ichikawa et al in [36] use Mitsubishi's 0.35 μm CMOS ASIC library and yield a throughput of 203.96 Mbits/sec.

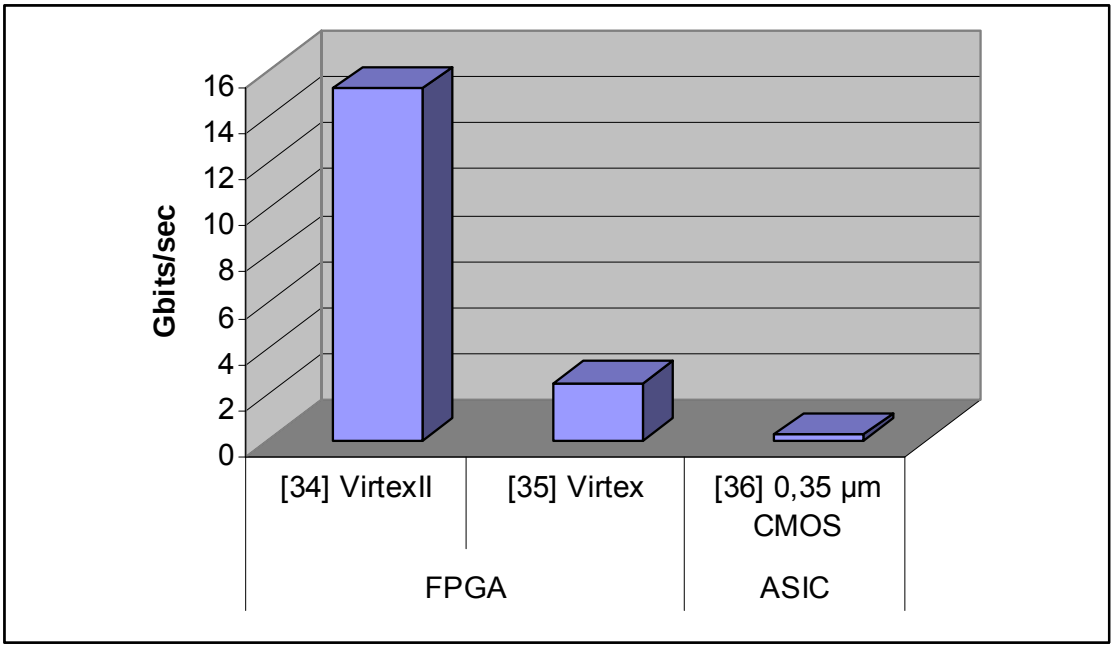

Chart 6 – RC6 hardware specific implementations

Few of the fastest Serpent implementations are the ones below, which are also summarized in Chart 7:

- Elbirt et al in [37] use a Virtex 1000 FPGA and achieve a 4.86 Gbits/sec throughput.
- Ichikawa et al in [36] use Mitsubishi's 0.35 μm CMOS ASIC library and yield a throughput of 931.58 Mbits/sec.
- Bora and Czajka in [38] use an Altera Flex 10K FPGA [39] and achieve a maximum throughput of 301 Mbits/sec.

Few of the fastest Twofish implementations are the ones below, which are also summarized in Chart 8:

- Elbirt et al in [37] use a Virtex 1000 FPGA and achieve a 1.58 Gbits/sec throughput.
- Schneier et al in [42] give hardware sizes and speed estimates that function at 150 MHz with a maximum 1.2 Gbits/sec throughput.
- Ichikawa et al [36] use Mitsubishi's 0.35 μm CMOS ASIC library and yield a throughput of 394.08 Mbits/sec.

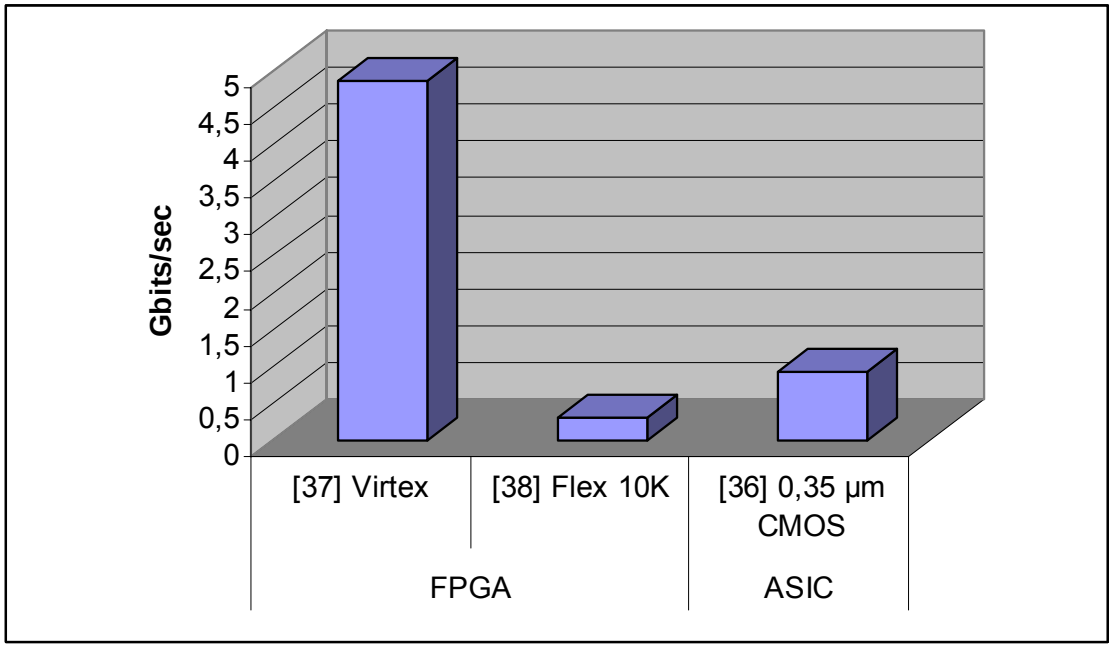

Chart 7 – Serpent hardware specific implementations

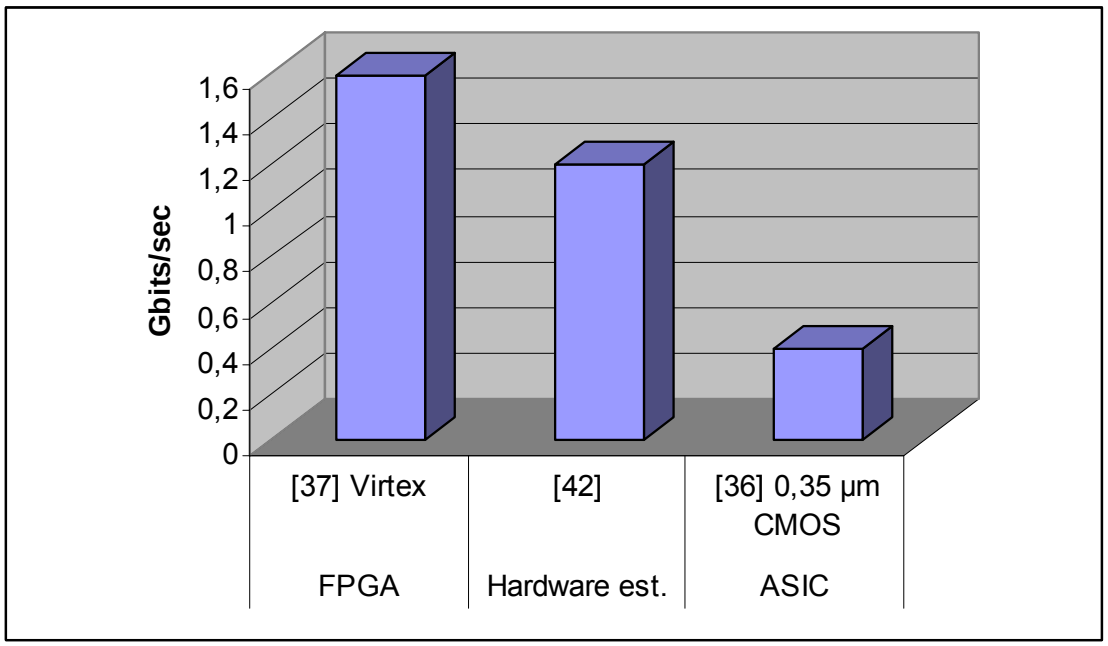

Chart 8 – Twofish hardware specific implementations

MARS has the smallest implementations number because of its complexity, large source utilization amount, slow processing speed and its Sboxes did not fulfill all NIST requirements [36], [37], [43]. Few of the fastest implementations are the ones below, which are also summarized in Chart 9:

- Ichikawa et al [36] use Mitsubishi's 0.35 μm CMOS ASIC library and yield a throughput of 225.55 Mbits/sec.
- Gaj and Chodowiec in [44] use a Virtex 1000 and achieve a 61 Mbits/sec throughput.
- Dandalis et al in [45] use Virtex FPGAs and achieve a 203.77 Mbits/sec throughput.

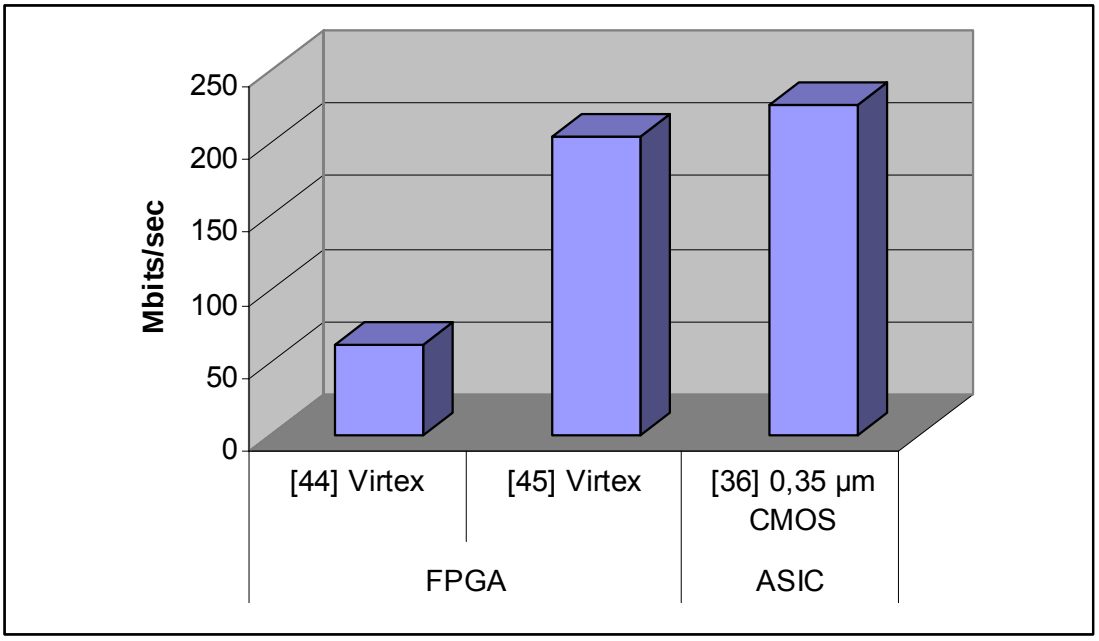

Chart 9 – MARS hardware specific implementations

From the various implementations that were mentioned above, it can be concluded that the Rijndael cipher performs the fastest processing speeds, while MARS the slowest ones. Indeed Rijndael offers very small implementation complexity, simple inner structure, high processing speed and security level, and low resource utilization, facts that deservingly made it the new AES [46].

3.3 Symmetric Ciphers ISA extensions and Hardware Co-Processors

The last category in the related work that has been done, is somehow "in the middle" of the previous two ones and it focuses on symmetric ciphers specific hardware co-processors. These designs may extend an existing processor's architecture in order to support more efficiently symmetric ciphers, or even introduce new co-processors specifically for some of them. As it may be easily comprehended, this category can be

characterized as the hardest of all, because it requires deep parallel analysis of many symmetric ciphers and extra effort, in order to obtain a balanced between performance and flexibility design.

3.3.1 Symmetric cipher accelerators and ISA extensions

Burke et al in [47] are trying to improve the performance of symmetric ciphers for the Alpha 21264 processor by examining eight algorithms. After analysis of bottleneck in these ciphers, they conclude to an extended ISA that consists of hardware rotations, modulo multiplication, permutation and Sbox access instructions and may achieve up to a 74% speedup over the baseline machine.

Murat Fiskiran et al in [54] study the effect of different addressing modes that can be used to calculate the effective address during Sbox access. More specifically they determinate how performance is affected on 1, 2, 4 and 8 wide EPIC (Explicitly Parallel Instruction Computer) processors depending on addressing mode of the architecture, issue width of the processor and number of memory ports. The results indicate that speedups exceeding 2x can be obtained when fast addressing modes are used.

Another similar approach comes from [53], where the same authors describe a new hardware module called PTLU (Parallel Table Look Up). It consists of multiple LUTs that can be accessed in parallel and its purpose is again Sbox access acceleration. Their results show maximum speedups of 7.7x for AES and 5.4x for DES, all tested on a single-issue 64-bit RISC processor.

Finally, Jung et al in [49] are trying to accelerate multiplication in GF (2^n) execution, an operation rather frequent in symmetric ciphers as stated in section 2.2. To be more specific, in this project they automate the design process for this kind of multipliers with VHDL (Very high speed intergraded circuit Hardware Description Language) and compare their results with other GF multipliers both on FPGA and ASIC implementations.

3.3.2 Hardware co-processors

Wu et al in [50] introduce the Cryptomaniac processor, a fast and flexible co-processor for cryptographic workloads. As it is mentioned on the paper, first they perform a cipher kernel bottleneck analysis on five symmetric ciphers and, in order to improve performance, a 4-wide 32-bit VLIW machine with no cache and a simple branch

predictor was designed. Its ISA consists of three instruction classes (tiny, short and long), giving a throughput of 512 Mbits/sec for AES. The design runs at 360 MHz in 0.25μm process and consumes 606mW. Figure 5 shows a high level schematic of Cryptomaniac's architecture, where BTB and FU stand for "Branch Target Buffer" and "Functional Unit" respectively.

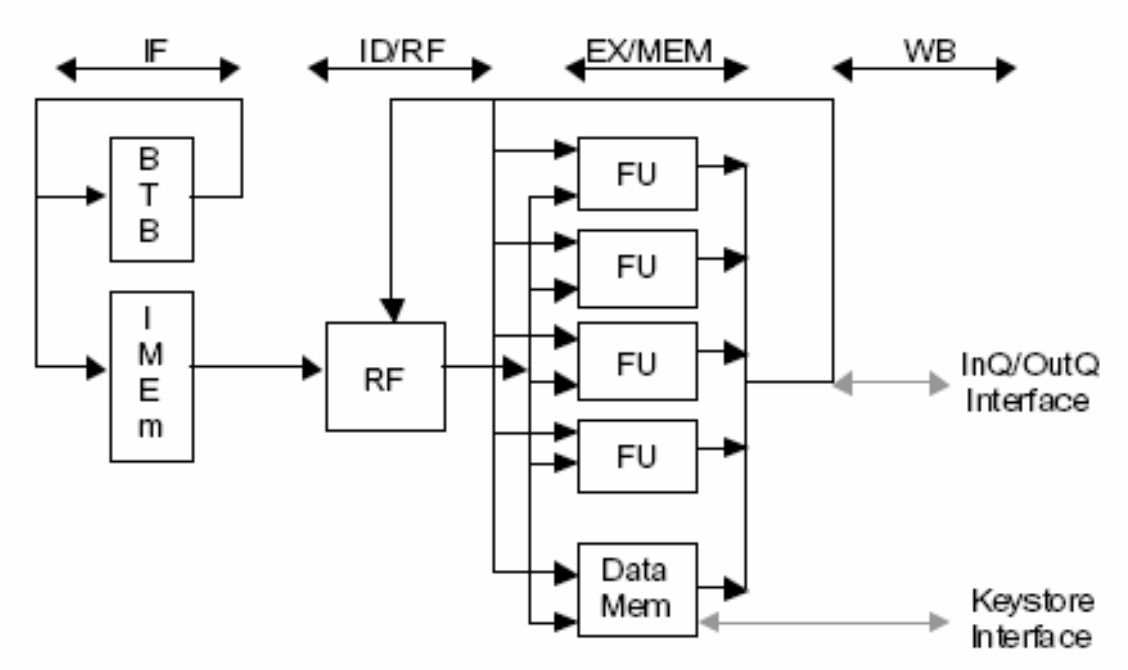

Figure 5 – Cryptomaniac's high level schematic architecture

Another similar approach comes from [52], where Oliva et al describe the Cryptonite, a programmable processor tailored to the needs of cryptography algorithms. The target frequency was 400 MHz in TSMC's 0.13 μm process [I14]. It consists of a three-stage pipeline datapath with two clusters and uses 64-bit instructions. Figure 6 shows its architecture overview. Each cluster consists of 4 64-bit registers, which are used from the two ALUs (Arithmetic Logic Units) and can also be exchanged. In addition, it has an "Address Generation Unit", which is being used from local data memory, in order to efficiently implement Sbox access operations. Results show a 68 Mbits/sec 3DES and a ~700 Mbits/sec AES performance.

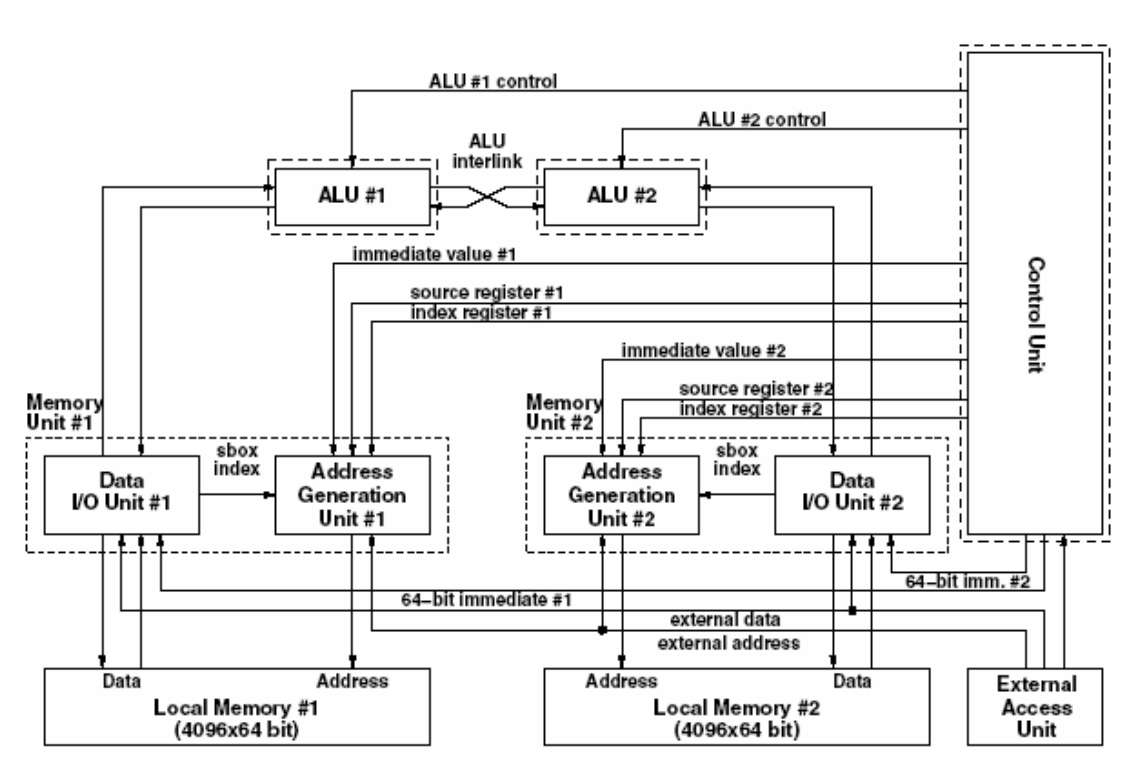

Figure 6 – Cryptonite architecture overview

Elbirt et al in [51] describe a design named COBRA, a specialized reconfigurable architecture that is optimized for the implementation of block ciphers. In order to be developed, many ciphers were analyzed leading to an ISA that supports arithmetic operations, modulo multiplication, GF multiplication and Sbox access modes. Figure 7 shows COBRA's schematic architecture, where RCE stands for "Reconfigurable Cryptographic Element". Each one of them performs a specific operation which can be selected from its 80-bit ISA. A notable characteristic of the COBRA design is that its reconfiguration capability affects function frequency; RC6, AES and Serpent are processed at 60.975 MHz, 102.41 MHz and 54.054 MHz respectively, achieving a maximum throughput of 3.9Gbits/sec, 1.451 Gbits/sec and 2.306 Gbits/sec respectively, while targeting a 0.35 micron Synopsys Design Compiler library [I17].

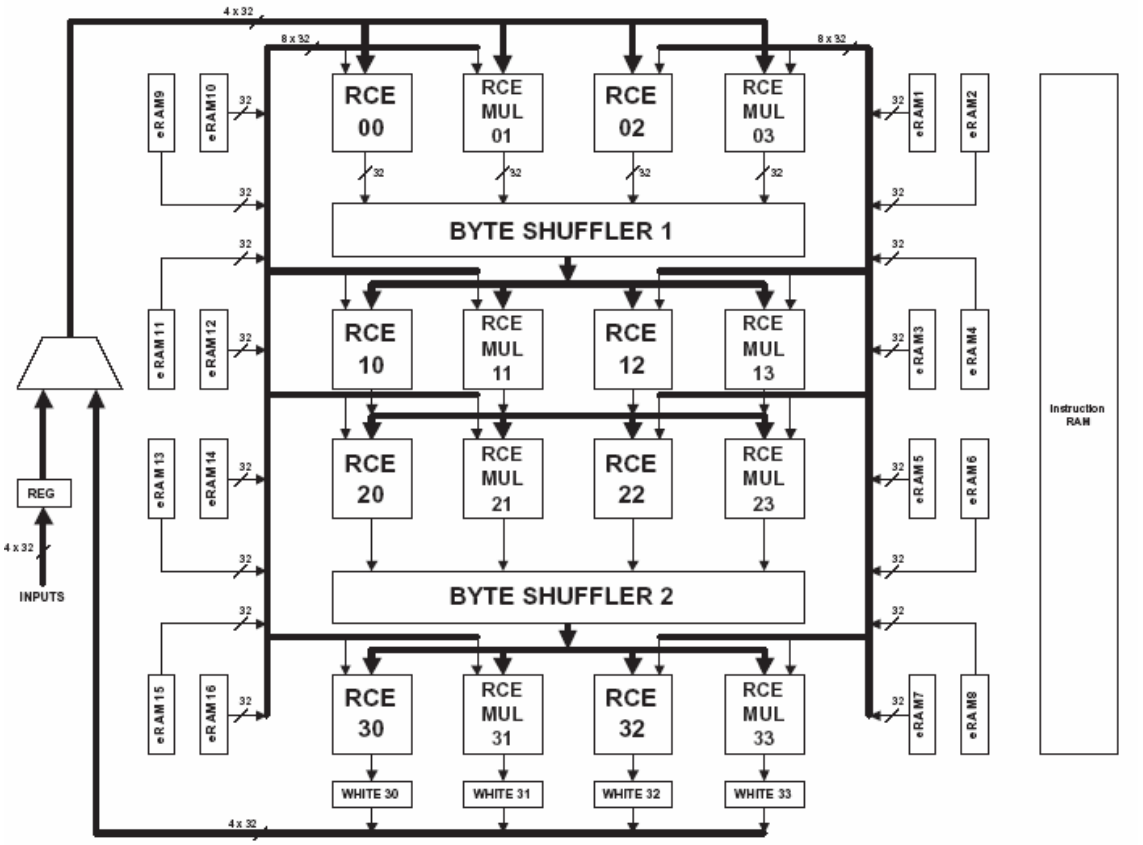

Figure 7 – COBRA architecture and interconnection

Dongara and Vijaykumar in [48] introduce another encryption mode called ICBC (Interleaved Cipher Block Chaining), which loosens the recurrence imposed by CBC and enables multiple encryption streams to be overlapped. ICBC evaluation is done with Wisconsin Wind Tunnel II [55] on SMP (Symmetric MultiProcessor). Various test for eight symmetric ciphers are performed on 2, 4, 8 and 16-processor schemes operating at speeds from 1 to 4 GHz. Results indicate a maximum speedup factor of 10x, achieving about 800 Mbits/sec for AES.

A final project that has few similarities with CCproc, is [I13] from Princeton University, called PAX. Until now there are no publications or official performance results. However, as it is stated in its official web page, PAX is datapath-scalable, minimalist cryptographic processor architecture for mobile and wireless information appliances, based on a simple RISC ISA, extended with few low-cost instructions. As far as it is known this project, as CCproc, are the only ones that their ultimate goal is to provide efficient symmetric cipher hardware process acceleration, beyond any specific algorithm in mind.

4. CCproc Architecture

In this chapter we focus on describing CCproc's architecture. First, a few design considerations will be discussed about the goal of this project and how it differs from other designs. After that follows CCproc's ISA and datapath structure in a detailed description along with documentation.

4.1 CCproc Design Considerations

As it can be concluded from related work that was presented in section 3.3 about symmetric ciphers hardware co-processors, each one of them utilizes its own unique architecture, targeted on a selected group of cryptographic algorithms. The only exception is the PAX project, which is not yet completed and, as stated before, is based on a simple RISC ISA.

Our initial motivation of this project was a hardware design, flexible enough to support many of today's popular symmetric ciphers, but also potential new ones. As years go by, previous symmetric ciphers that use keys smaller than 128-bit are likely to be abandoned, because they will be vulnerable to brute-force attacks. So, after we analyzed them carefully, we discovered that some of their functional principles were not adopted by the newest ones.

- A first example is bit permutation or expansion, which is primary used from DES. Although in combination with carefully designed Sboxes, it offers strong security, none of the AES round two finalists used it. Only Serpent has an initial and final permutation, from which, as mentioned in its official submission, the bitslice version is much more efficient. In fact, these permutations have no security purpose, but they are only used to switch the cipher from regular to bitslice mode. Although there have been studied structures that offer arbitrary bit permutations, they require a considerable amount of hardware [56] and though it was decided not to be used.
- Another example is arithmetic operations using data with other than 32-bit or multiples of it. Again DES is an example, which starts with a 64-bit key that is immediately reduced to a 56-bit data value and then is split into two 28-bit data values. In contrast, all of the AES round two finalists use only 32-bit multiples

data values, because 32-bit processors were their initial target. As a result, this observation led us to use 32-bit operations.

- Complexity of the key expansion process is another aspect, which were not inherited to new 128-bit ciphers. Indeed there were cases where this part of a symmetric cipher was very slow, although it is not required during data exchange. However, this factor is prohibitive for such algorithms utilization in restricted devices (smartcards), where key changes every few milliseconds. A strong example is Blowfish where, in order to complete the key expansion process, Blowfish itself is required to run 1042 times. In contrast, all AES round two finalists consist of much simpler key expansion processes, with the exception of MARS. The latter in fact was negatively criticized, because, among others, of its complex and weird key expansion process. As a result, in order to explicitly support a cipher's key expansion process, the only extra functional unit that we would add is a KRF (Key Register File) memory module in order to store all expanded keys, plus an ISA expansion to support operations between RF's (Register File) data and KRF's data. The latter is explicitly described in section 4.3.3.

- Another characteristic that was not used by any of the AES round two finalists is variable Sboxes. In contrast, every one of them uses its own Sboxes, which remain constant during the key expansion and encryption / decryption process. Twofish is the only one that interleaves two 32-bit XOR operations between its three Sboxes structures, which simply alters the final Sbox output. Older algorithms, such as Blowfish and RC4, required their Sboxes first to be initialized before encryption commences. As a result we decided in CCproc to have few ROMs (Read Only Memory) as cipher specific Sboxes and small RAMs (Random Access Memory) for new Sboxes support, plus available data space during the key expansion process. However, [53] that was described in section 3.3 and published after CCproc's first version design was completed, proposes flexible structures capable to implement various Sbox sizes. Integration of such structure might help to reduce CCproc's second version design complexity somewhat and increase even more its functional frequency and flexibility.

Another consideration was to record in symmetric ciphers their instruction types, an aspect described in Chapter 2, and frequency occurrence. This research revealed a high frequency occurrence of two dependent, back-to-back instructions. Examples are double additions, subtractions and XOR, and addition or subtraction followed by a XOR. In order to save valuable computing clock cycles, we decided that this type of double-instructions should be included in CCproc's ISA.

We also observed that all AES round two finalists treat 128-bit plaintext as four 32-bit words. Initial thoughts consisted of a single 32-bit RISC datapath structure extended to support an enhanced symmetric cipher ISA. After a few cipher implementations in such a design, it was quickly discovered that the latter was too narrow to achieve an adequate performance. Many times during processing, symmetric ciphers require 64-bit, 96-bit or even 128-bit data values at the same time in order to proceed, so with a narrower datapath, additional clock cycles are spent on fetching all appropriate data to the functional unit that will use them. This performance obstacle led to the decision to examine the level of parallelism that can be reached for each one of the AES round two finalists when breaking into four small threads and running in a hypothetical 5-stage pipelined datapath structure, which could be able to process up to four 32-bit operations in one clock cycle. Its abstract schematic overview is shown in Figure 8 where each cluster consists of a 4-stage pipelined datapath (decode, execution, data memory, write back), and the results are shown in Chart 10.

This Chart shows utilization of the previously mentioned datapath for each one of the ciphers while taking advantage of the double-instructions mentioned before. In order to extract these results, first we wrote an assembly program for each cipher and then we optimized it as much as possible by taking advantage of the four clusters. More specifically, we analyzed each one of these programs and detected which of the instructions could be concurrently processed. The worst cases are when only one, or even none instructions can be processed during a clock cycle, because data have not been yet computed, while the best case is when each cluster processes an instruction, leading to maximum parallelism. In Chart 10, the higher a column is, the better the parallelism. For example AES and Serpent can benefit more from a wider datapath structure than the other ciphers. Having in mind a design that would achieve competitive performance, it was finally considered a VLIW processor that would

consist of four 32-bit clusters, capable to process four 32-bit instructions in one clock cycle.

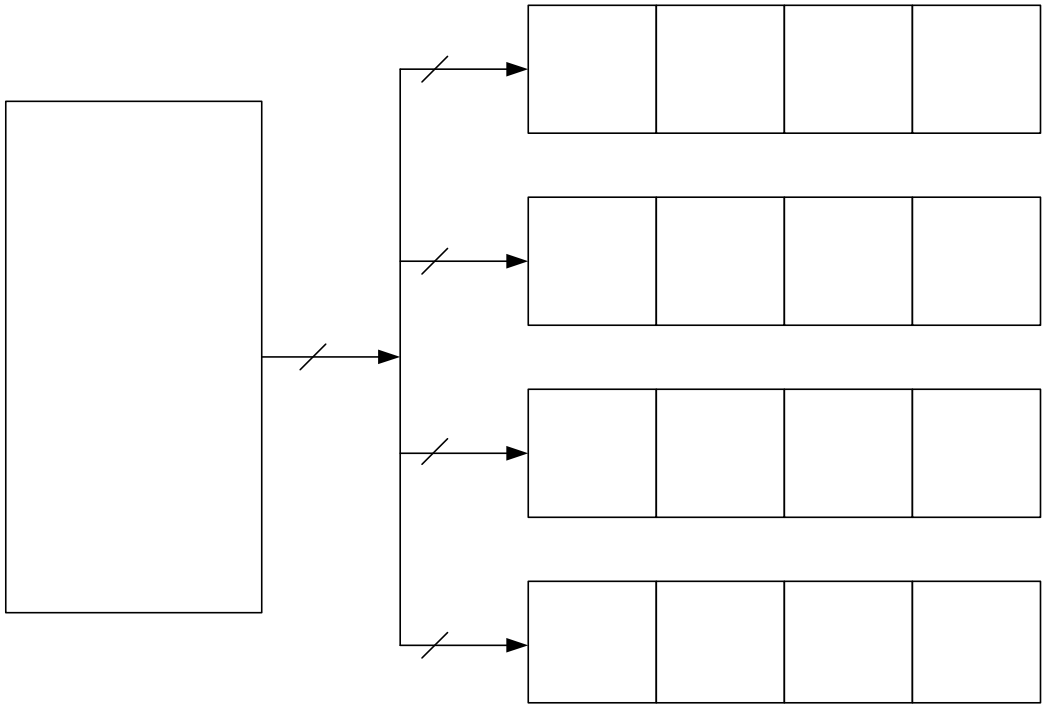

Figure 8 – CCproc's abstract schematic overview

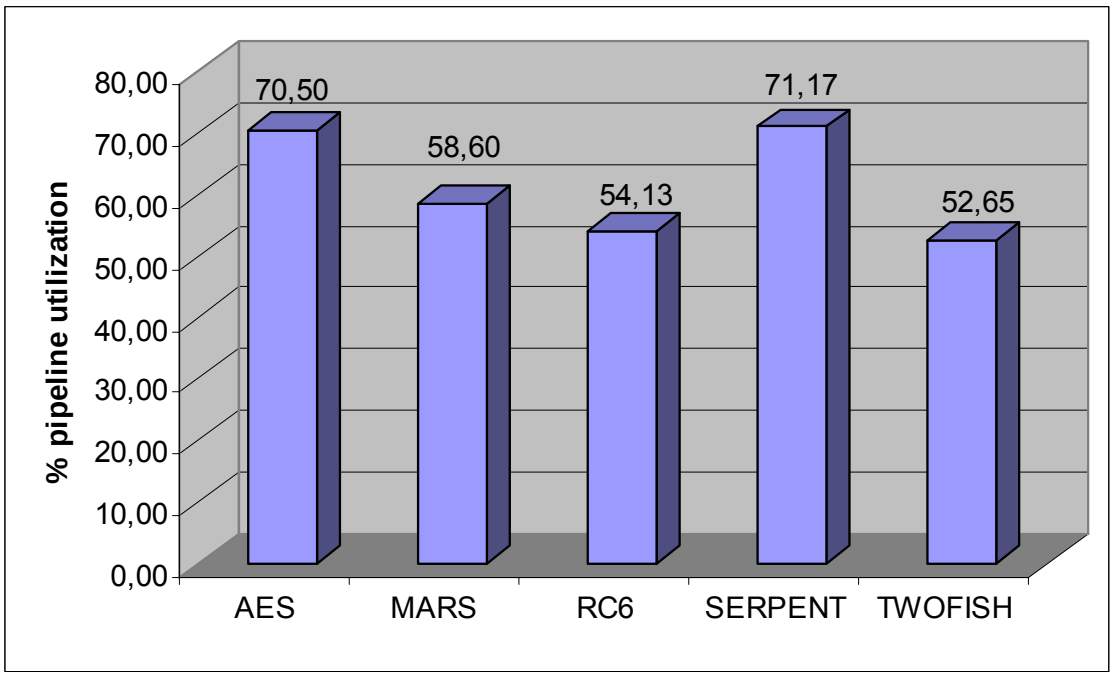

Chart 10 – AES round two finalists level of parallelism in a 128-bit datapath structure

Another conclusion that came up from the cipher analysis above was the small number of 32-bit registers utilization. In each cluster a maximum of four registers were used ending up to a total of sixteen 32-bit registers among the four clusters. However, it was decided an 8x32 RF in each cluster, in order to cover the case where a cipher, that was not tested, might need additional registers.

Also, an important consideration was if all clusters would be identical to each other. First thoughts consisted of not to include all large functional units, such as modulo multipliers, in every cluster, for hardware reduction resources reasons. But after further analysis, it was discovered that this way certain cipher threads would need to be "locked" on running in specific clusters, in order to avoid additional data movement between them, leading to an increased number of processing clock cycles, plus a reduced datapath flexibility. Having in mind a future design being capable to process independent threads from different ciphers, resulting to a significantly increased throughput, it was decided to implement four almost identical clusters.

Another aspect regarding the efficient CCproc's ISA expansion was the kind of control instructions that would be supported, such as branches or jumps. A fact that characterizes every symmetric cipher is the determined rounds number during the key expansion process plus encryption / decryption. As a result they can be written in a way that requires absolutely no branch tests. This observation led to the decision of not to support any kind of branch instruction that its direction could not be pre-evaluated, costing additional datapath stalls and hardware resources. In contrast, we decided, after detailed cipher analysis, to support only a "loop" instruction that would add no stalls, because the round number would have been a priori specified. As it was verified from later implementation tests, this scheme worked very well eliminating nearly all of the branch-related pipeline stalls. A more detailed hardware description is in section 4.3.1.

A final consideration was cipher support. As we mentioned earlier in this section, every cipher that uses keys smaller than 128-bit are considered as non-secure. In addition, as Huffmire in [61] mentions, a cryptography co-processor should be able to support as many ciphers as possible, in order to provide a strong security level against various kinds of attacks. This fact lead us to the decision to design a co-processor capable to process today's (2005) best ciphers, i.e. the AES round two finalists, plus to have an extended ISA efficient and general enough to cover future algorithms.

4.2 CCproc Instruction Set Architecture

After evaluating all the above considerations, we designed the ISA of CCproc. Supported instructions are categorized to four primary formats, Register, Immediate, loop and cipher, which are shown in Table 2. Also Table 3 shows the field meaning.

	31-27	26-22	21-19	18-16	15-13	12-10	9-8	7-4	3	2	1	0
R	opcode	func	rdx	rsa	rsb	rsc	mx	nu	KRFWrEn	nu	KRFPInc	nu
I	opcode	func	rdx	rsa	Ix[15..0]							
loop	opcode	nu			label[11..8]		mx	label[7..0]				
cipher	opcode	nu	rdx	rsa	rsb	nu					opt	

Table 2 – MyDesgin's ISA formats

Each instruction format, as already stated, is 32-bit, while bits are numbered from 31 down to 0, with 31 being the MSB (Most Significant Bit). First row shows the bits that each field uses, while the others show how fields have been split in each format.

The "opcode" field is used by every format, in order to distinct from each other and for easier instruction decoding. It is a 5-bit field and is analyzed in Table 4.

Field	Explanation	Description
opcode	operation code	Determines instruction format
func	ALU function	Function that ALU, GFM or MM will perform
rdx	RF destination register	Register that will be written
rsa	RF source a register	1st register to be read
rsb	RF source b register	2nd register to be read
rsc	RF source c register	3rd register to be read
mx	move from cluster	Specifies if a cluster will get data from another cluster
label	instruction label	Beginning address of a loop
Ix	immediate	Immediate data value
KRFWrEn	KRF write enable	Enables KRF's write enable
KRFPInc	KRF pointer increment	Enables KRF's pointer increment by 1
opt	cipher options	Specifies various modes of cipher instructions
nu	not used	These bits are not used

Table 3 – Field explanations

Bit	Description
4	Specifies if RF will be written
3	Shows if it is an I format instruction or not
2	Shows if it is a Cipher format instruction or not
1-0	Select Sboxes

Table 4 – "opcode" field analyzation

R and I formats use the "func" field, which, as mentioned in Table 2, specifies the ALU's (Arithmetic Logic Unit), GF (Galois Field) multiplier or MM (Modulo Multiplier) operation. **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.** summarizes all supported instructions in R, I and loop formats.

Format	Operation	Syntax	Description
R	add	add rdx,rsa,rsb	adds rsa with rsb and stores the result to rdx
	and	and rdx.rsa,rsb	logic and between rsa and rsb, and stores the result to rdx
	gfm	gfm rdx,rsa	galois field multiplication in GF(28) between rsa and GF operand x and the result is stored to rdx
	krfpaz	krfpaz	resets KRF's pointer to first address
	ldgfmr	ldgfmr rsa	loads 8-bit mr register with rsa's value, which holds modulo polynomial in Galois Field multiplication
	ldgfopx	ldgfopx rsa	loads Galois Field operand x with rsa's value
	ldlc	ldlc #a	loads 6-bit lc register with #a
	mmult	mmult rdx,rsa,rsb	modulo 2^{32} multiplication between rsa and rsb and the result is stored to rdx
	or	or rdx,rsa,rsb	logic or between rsa and rsb, and stores the result to rdx
	r2c/c2r	r2c / c2r rdx,rsa	toggles between rows and columns in a 128-bit data value
	rol	rol rdx,rsa,rsb	rotates left rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	ror	ror rdx,rsa,rsb	rotates right rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	shl	shl rdx,rsa,rsb	shifts left rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	shr	shr rdx,rsa,rsb	shifts right rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	sub	sub rdx,rsa,rsb	subtracts rsb from rsa and stores the result to rdx
	xor	xor rdx,rsa,rsb	logic xor between rsa and rsb, and stores the result to rdx
	addadd	addadd rdx,rsa,rsb,rsc	adds rsa with rsb, adds the result to rsc and stores it to rdx
	addsub	addsub rdx,rsa,rsb,rsc	adds rsa with rsb, subtracts rsc from the result and stores it to rdx
	addxor	addxor rdx,rsa,rsb,rsc	adds rsa with rsb, logic xor between rsc and the result and stores it to rdx
	subadd	subadd rdx,rsa,rsb,rsc	subtracts rsb from rsa, adds the result to rsc and stores it to rdx
	subsub	subsub rdx,rsa,rsb,rsc	subtracts rsb from rsa, subtracts rsc from the result and stores it to rdx

	subxor	subxor rdx,rsa,rsb,rsc	subtracts rsa with rsb, logic xor between rsc and the result and stores it to rdx
	xoradd	xoradd rdx,rsa,rsb,rsc	logic xor between rsa and rsb, adds the result to rsc and stores it to rdx
	xorsub	xorsub rdx,rsa,rsb,rsc	logic xor between rsa and rsb, subtracts rsc from the result and stores it to rdx
	xorxor	subxor rdx,rsa,rsb,rsc	logic xor between rsa and rsb, logic xor between rsc and the result and stores it to rdx
I	addi	addi rdx,rsa,#a	adds rsa with #a and stores the result to rdx
	andi	andi rdx.rsa,#a	logic and between rsa and #a, and stores the result to rdx
	lui	lui rdx,#a	loads 16-bit value #a to rdx's 16 MSBs
	ori	ori rdx,rsa,#a	logic or between rsa and #a, and stores the result to rdx
	roli	roli rdx,rsa,#a	rotates left rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	rori	rori rdx,rsa,#a	rotates right rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	shli	shli rdx,rsa,#a	shifts left rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	shri	shri rdx,rsa,#a	shifts right rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	subi	subi rdx,rsa,#a	subtracts #a from rsa and stores the result to rdx
	xori	xori rdx,rsa,#a	logic xor between rsa and #a, and stores the result to rdx
loop	loop	loop label	jumps to the beginning of a loop which starts at address “label”

Table 5 – Supported operations in R, I and loop formats. Bold means double instructions and italic means custom instructions.

We should note that these instructions are also used for operations between KRF’s and RF’s registers, plus for data movement between clusters. In section 5.1 there is a detailed description of CCproc’s Python [I15] assembler and cipher examples that show how this kind of operations is supported.

Cipher instruction format is used from MyDesing’s first version design and its purpose is an efficient Sbox access, depending on the cipher that is processed. Note that should the cipher Sboxes are replaced with a more dynamic structure in a future version, this format will need to be updated. Table 6 shows all supported cipher instructions.

Instruction	Syntax	Description
aesX	aesX rdx,rsa	Sbox access during AES encryption or decryption (X=E,D) with rsa and the result is stored to rdx
marsX	marsX rdx,rsa	Sbox access during MARS forward mode, backward mode, or E function (X=F,B,E) with rsa and the result is stored to rdx
serX	serX rdx,rsa	Sbox access during Serpent encryption or decryption (X=E,D) with rsa and the result is stored to rdx
tX	tsld rsa,rsb / tsbox rdx,rsa	during Twofish, loads to S0 and S1 rsa and rsb respectively / Sbox access with rsa and the result is stored to rdx

Table 6 – Cipher format instructions

4.3 CCproc Datapath Structure

After presenting CCproc's ISA, this section focuses on describing its datapath structure, having as target device a Virtex 4 FPGA. First it shows in detail how every functional unit works and finally there is sub-section 4.3.8, where everything is put together to assemble CCproc co-processor.

4.3.1 The Loop instruction controller circuit

As we mentioned in section 4.1, after closer analysis of various symmetric ciphers, we concluded that a "loop" instruction it was enough to handle all control hazards. As it can be seen from Figure 9, which shows the loop controller circuit, there is "lc" register, two multiplexers A and B, a '1' constant subtraction unit and a comparator.

When an instruction is being fetched from instruction cache, it is checked if an "ldlc" or "loop" occurred. If it is the first case, then multiplexer A gives to "lc" the rounds number that a loop will be repeated. The latter is complete when a "loop" instruction occurs, and if "lc" value is greater than 1, then its current value is reduced by 1 and "nPCsel" signal is asserted, in order to enable a new instructions loop commencement. If it is 1, it means that the appropriate rounds number has been completed, "nPCsel" is not asserted and program execution continues normally.

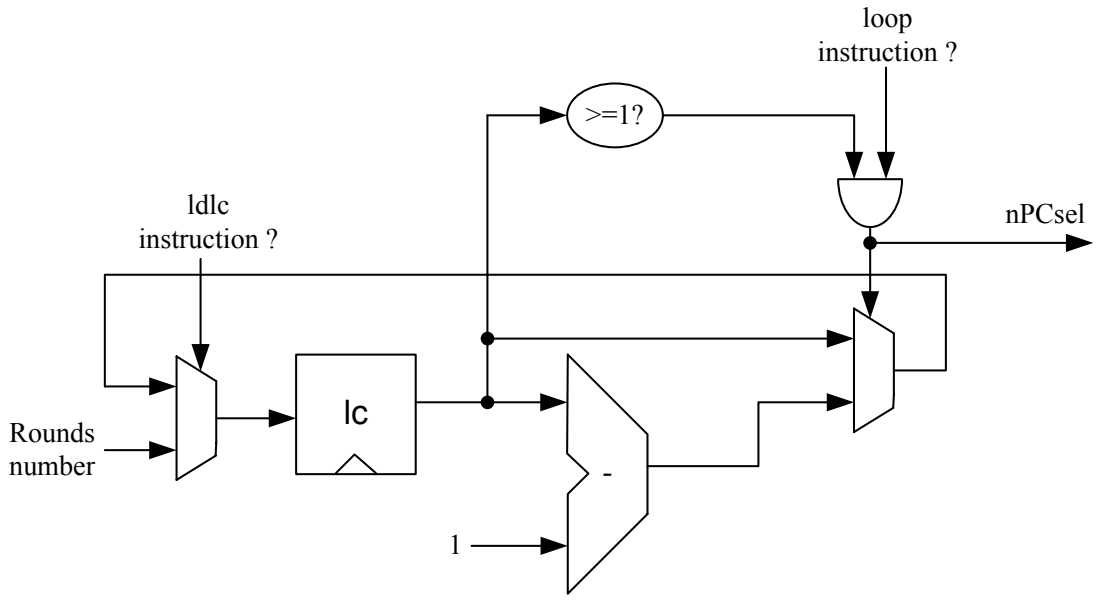

Figure 9 – Loop controller

For example, when we have a loop in C language, we can write it in CCproc's assembly as it shown in Figure 10. Suppose that a, b, c and d variables are stored in each cluster's r1 register. In CCproc's assembly we first initialize the loop counter (lc) to 10 (a in hexadecimal) and then we begin the for-loop. It should be noted that each quad is executed in every clock cycle, so this loop will take 10 clock cycles to complete, however it will not issue any pipeline stall at all!

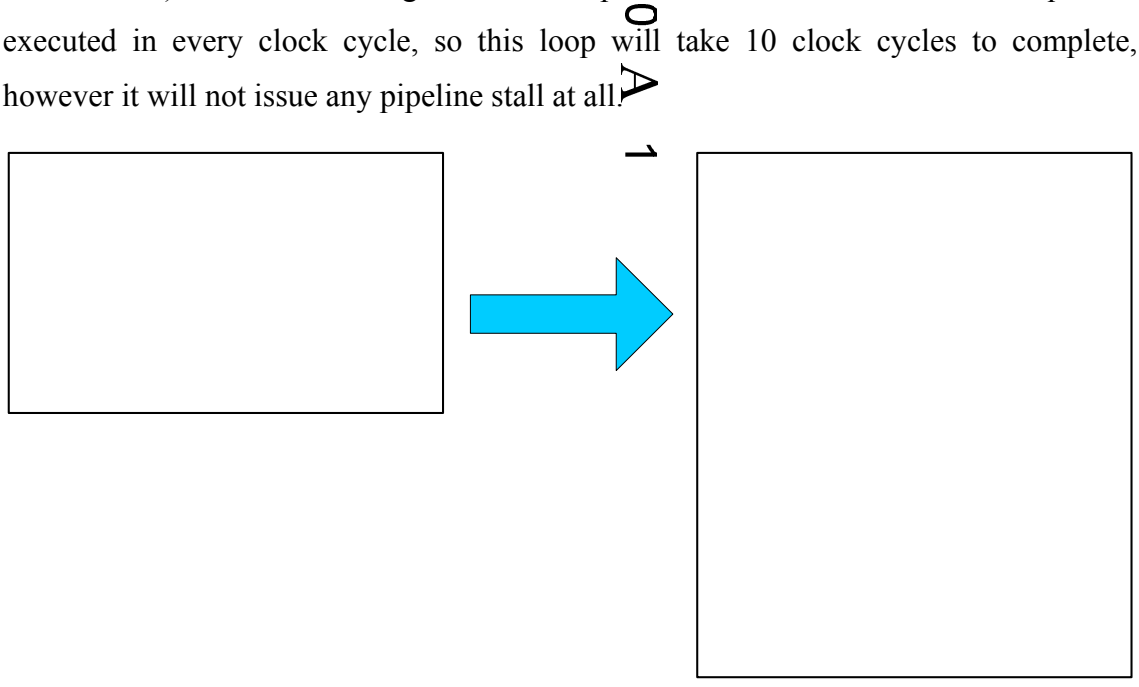

Figure 10 – A C and its equivalent CCproc assembly language loop

4.3.2 Register File (RF)

As we mentioned in section 4.1, cipher processing required a total amount of four 32-bit registers per cluster for each of the AES round two finalists. As a result CCproc's RFs through each cluster, as was also explained earlier, are 8x32, which means eight 32-bit registers. RF is fully synchronous, which means that reading from and writing to it occurs on the positive clock edge.

In order to be able to read up to three different registers in a single clock cycle, a three port RF was designed, as shown in Figure 11, having three copies of an 8x32 register set. During a read operation, each one of them can provide an independent 32-bit register, through each one of the "RdAddr1", "RdAddr2" and "RdAddr3" address signals, resulting up to three 32-bit registers to "DataOut1", "DataOut2" and "DataOut3" signals in single clock cycle. This is particularly useful when double-instructions occur where three operands are needed at the same time. However, all copies must always be identical to each other, so there is only one "WrAddr", "WrEn" and "DataIn" signal, writing every time the same data in each RF core. At this point we should note that a full-custom implementation of this RF would of course be much more efficient.

When there is a case where one or more of the "RdAddr1", "RdAddr2" and "RdAddr3" signals are equal to the "WrAddr" signal, i.e. a write and read operation occur on the same register, there is logic that passes immediately "DataIn" value to the appropriate "DataOutX" signal. In other words, this RF utilizes a Read-After-Write scheme.

4.3.3 Key Register File (KRF)

The KRF is a special RAM in each cluster's decode stage, where a cipher's expanded keys are stored. After analysis of AES round two finalists, these keys can be separated between CCproc's KRFs in such a way that they would not need to be moved between clusters during a cipher's encryption / decryption process. Every KRF is a 64x32 data space, meaning it has sixty four 32-bit registers. This size was chosen after observation of the expanded key's data size and finding that it did not exceed a total of thirty three 32-bit data values per cluster.

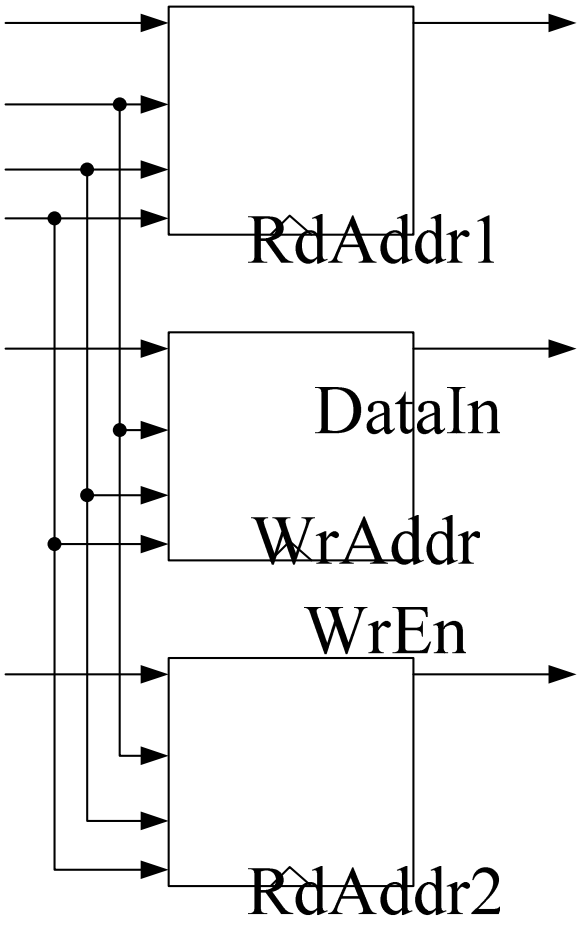

Figure 11 – CCproc's main 8x32 RF

Another fact is that every cipher uses its expanded keys serially forwards or backwards in some cases during decryption. As a result there is no need for a separate field in an instruction's format to specify a KRF address, as it is being done during RF register read or write operations. In contrast, we decided to use only bit "KRFPInc", which would enable a serial auto-increment KRF access. A similar KRF write scheme was also decided. More specifically, there is "KRFWrEn" bit, which when asserted, it enables serial data write to KRF with auto-incremented address generation. However if a cipher requires its expanded keys backwards, they should first be written in reverse order.

Figure 12 shows the entire KRF circuit. As we can see, there is "KRFP" register, which is KRF's pointer. Each time "KRFPinc" bit is asserted, multiplexer A will auto-increment "KRFP" register by 1 in the next clock cycle, while the latter's present value is used for KRF access. When an instruction uses KRF's data during an operation, multiplexer B does not select RF's "DataOut1" signal, but KRF's "Dout" signal. During a KRF write operation, "KRFWrEn" bit is asserted, which directly connects to KRF's

"WrEn" input and is also used for "KRFP" auto-increment, in order to point to the next address when another KRF write operation occurs.

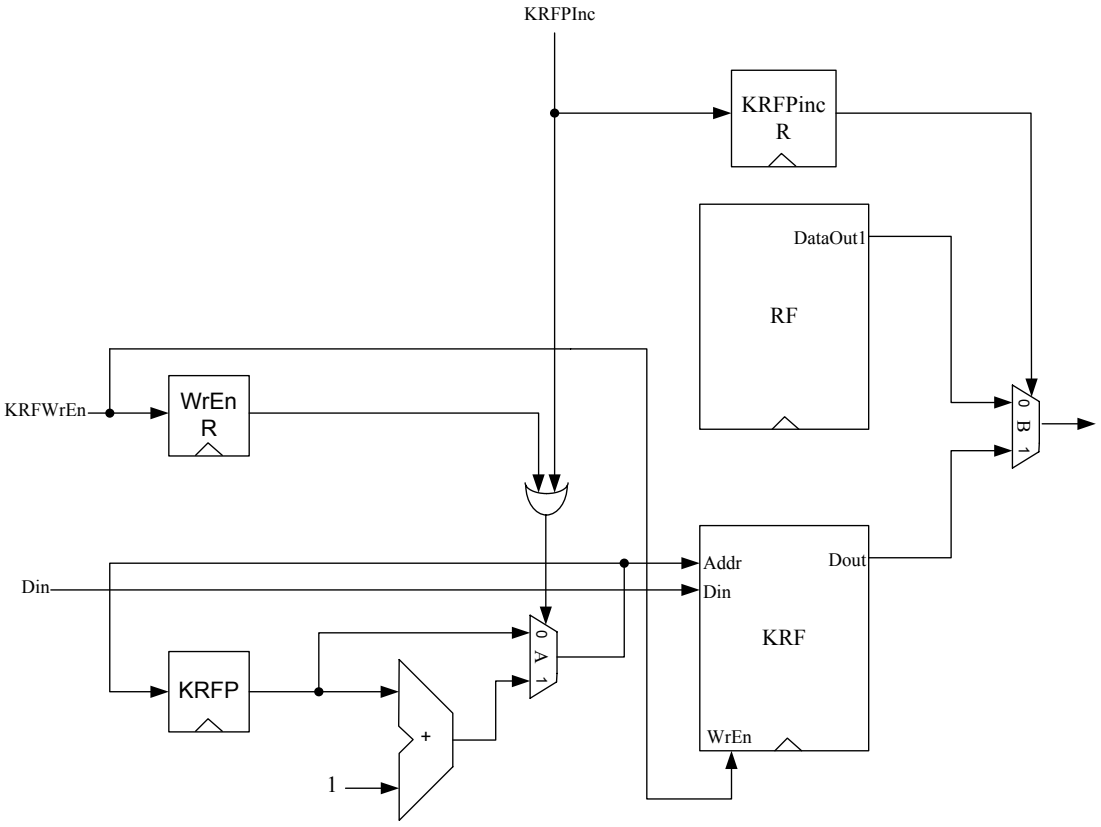

Figure 12 – KRF circuit

This functional unit supports continuous KRF write operations, i.e. an expanded key can be written every clock cycle. However, when the key expansion process has finished, "KRFP" must be reset to point again to KRF's first position. Before this can be done, user must interleave a non KRF write operation between the last write operation and "krfpaz" operation, or else the final expanded key will not have enough time to be written. Also, in order to proceed to a KRF read operation there must be interleaved two clock cycles between the "krfpaz" operation and first read operation, in order to "KRFP" have enough time to be initialized. After these two clock cycles, again this circuit is capable to read an expanded key each clock cycle.

4.3.4 Arithmetic Logic Unit (ALU)

The ALU functional unit is the processor's beating heart, because most of the instructions issued, use it. As we mentioned in section 4.2, there are many double-

instructions that require three operands, which results to an ALU that has three 32-bit inputs and one 32-bit output.

As it can be observed from Figure 13, there are three 32-bit ASUs (Addition / Subtraction Units), three 2-input 32-bit xors and three multiplexers. ASU A adds or subtracts inputs "In1" and In2", while gate A makes a xor operation between them. If there is a double-instruction, results from ASU A and gate A, are passed, in combination with "In3", through ASUs B and C, and gates B and C. Finally multiplexers A, B and C are used to select appropriate data depending on the value of "func" field while, in arrows before multiplexer C is shown operation allocation. ALU instructions have the below specific format:

```
Result ← (In1 op1 In2) op2 In3
```

where "In1", "In2" and "In3" are the three "ALU core 1" inputs and op1, op2 are the two operations that may be performed.

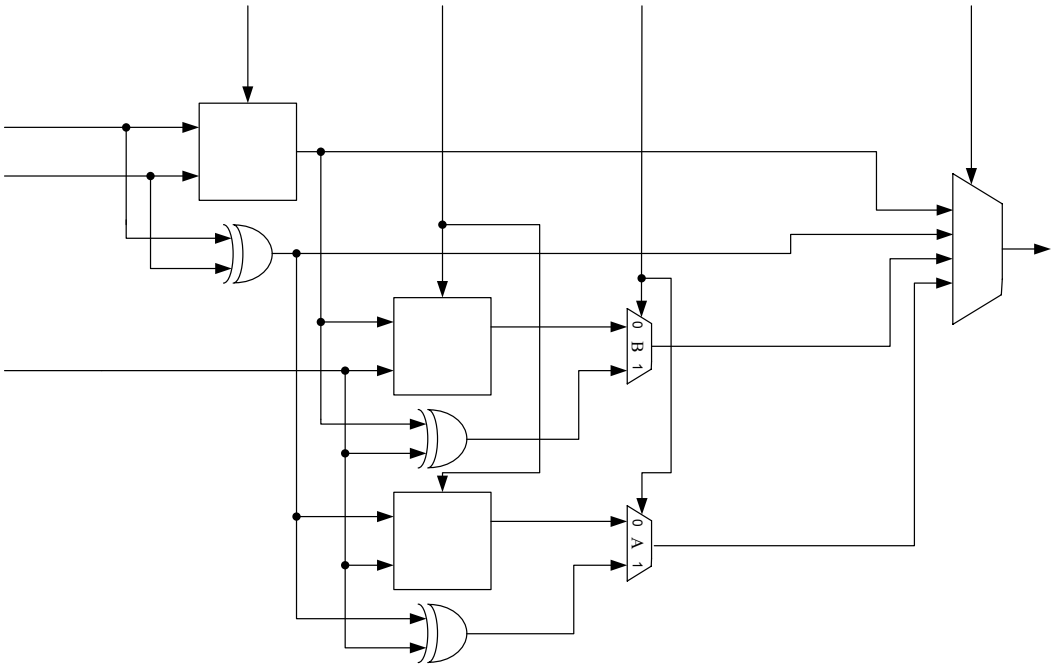

Figure 13 – CCproc's "ALU core 1"

In Table 7 we analyze "func" field and show how its value is assembled depending on operation that is about to be performed. As we mentioned in section 4.2, "func" is a 5-bit field, with each one having its own meaning:

- Double indicates if there is a double-instruction or not.
- Xor2 means that op2 = xor.
- Xor1 means that op1 = xor.
- Add2 means that op2 = add / sub.
- Add1 means that op1 = add / sub.

Besides the "ALU core 1" there is another functional unit, called "ALU core 2", that is used for data rotations and shifts, and is shown in Figure 14. More specifically there are two SRUs (Shift / Rotate Units), which take as inputs 32-bit "DataIn" that will be shifted / rotated, a 5-bit "amount" that indicates the specific shift / rotation amount plus a "shift / rotate" signal that selects shift or rotation. Once the two SRU's have finished, multiplexer A selects the appropriate direction, depending on instruction that were issued.

Instruction	double (bit 4)	xor2 (bit 3)	xor1 (bit 2)	add2 (bit1)	add1 (bit 0)
add	0	0	0	0	0
sub	0	0	0	0	1
xor	0	0	1	0	0
addadd	1	0	0	0	0
addsub	1	0	0	1	0
addxor	1	1	0	0	0
subadd	1	0	0	0	1
subsub	1	0	0	1	1
subxor	1	1	1	0	1
xoradd	1	0	1	0	0
xorsub	1	0	1	1	0
xorxor	1	1	0	0	0

Table 7 – "func" field analysis

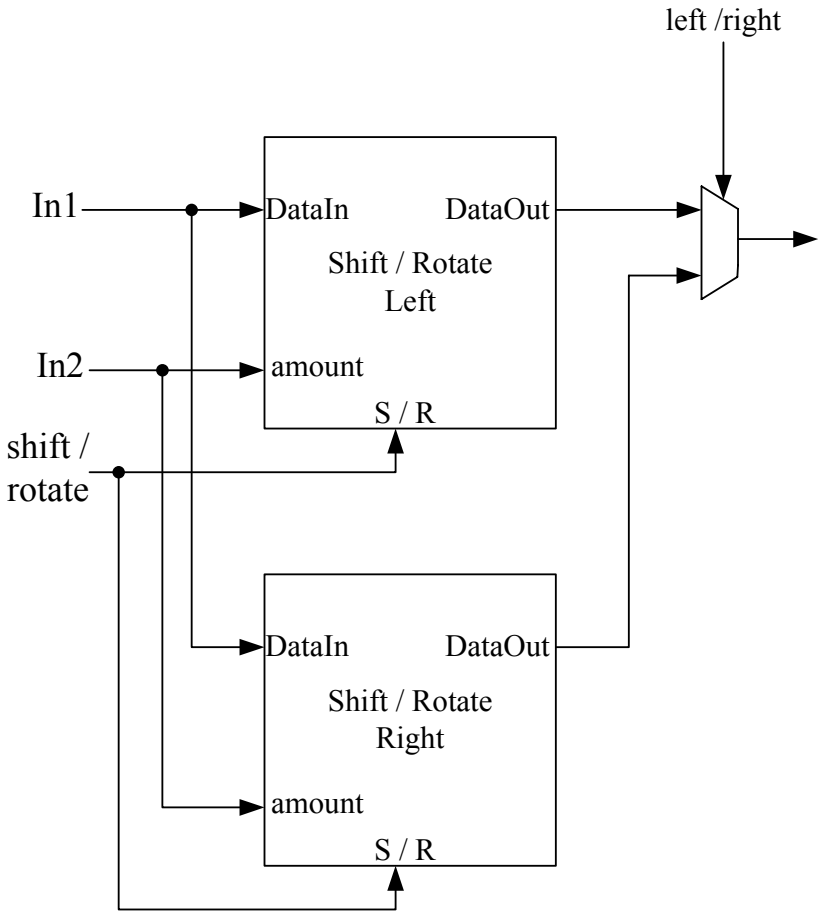

Figure 14 – CCproc's "ALU core 2"

4.3.5 8-bit Galois Field (GF) Multiplier in GF(2^8)

As we mentioned in section 4.2, the GF multiplier that is used in CCproc, performs 8-bit multiplications modulo a prime polynomial over GF(2^8). From the AES round two finalists, AES and Twofish use this kind of operation, both utilizing static 4x4 matrices, where each cell contains a byte and modular polynomials.

In order to design a small and rather fast GF multiplier, based on [49], it was decided first to design a PPG (Partial Product Generator) and then use it to implement the entire unit. Figure 15 shows PPG's schematic, which takes as input a byte "In1" that first is shifted left one bit. The result is then xored with input byte "mp", which is the modular polynomial that a cipher uses. Multiplexer A selects shifted data or xored data, dependeing on "In1" MSB and then multiplexer B selects as final output '0' or multiplexer's A output, according to b_i. The latter bit is the ith bit of the second operand that is used in GFM, where i=0...7. As a result, a PPG has two outputs, "PPG (i+1)" that goes to the next PPG, and "pout" that is the ith's PPG partial product.

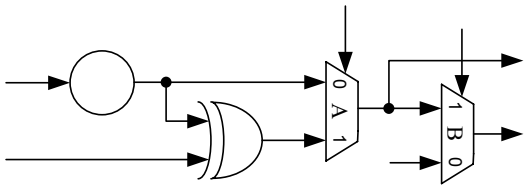

Figure 15 – Partial product generator

These partial generators now are cascaded in the way that is shown in Figure 16, assembling an 8x8 GF multiplier. However, while testing, its timing characteristics as a complete combinational circuit were not satisfactory, so we pipelined the multiplier. Instead of using adders to summarize complete product from the partial ones, we are using XOR gates, according to the specification of $GF(2^8)$ multiplication that XOR intermediate results.

The GF multiplier 8x8 unit can compute the product between two bytes in $GF(2^8)$. However in AES and Twofish there were four 32-bit words (intermediate plaintext) and a 4x4 static matrix with each cell consisting of a byte, resulting to another four 32-bit words. As a result processing a cipher would require a total of sixteen 8x8 multiplications, each having a 1 clock cycle latency. This fact leads to a tremendous cycle-consuming multiplier, which slows down significantly a cipher's process.

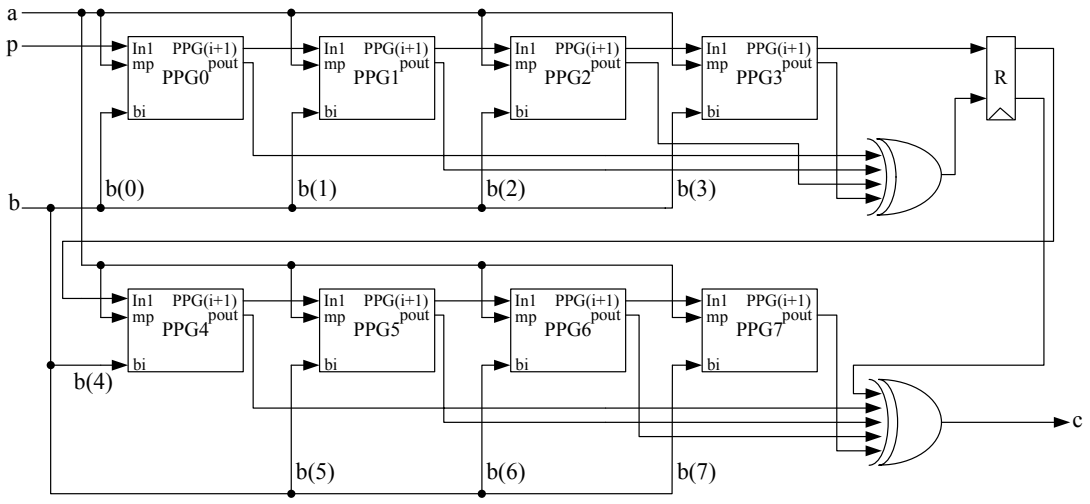

Figure 16 – GFM 8x8

In order to avoid such latency, we decided to increase the number of GFM's 8x8 to sixteen, to take advantage of the fact that these multiplications can be computed entirely in parallel. Figure 17 shows a GF multiplier 16x8x8 schematic, where "a" is a 32-bit

input consisting of bytes [a3:a2:a1:a0], “op3”, “op2”, “op1” and “op0” are the four matrix columns ,”mr” is a register that holds the appropriate modular polynomial and “R” is GF multiplier’s result.

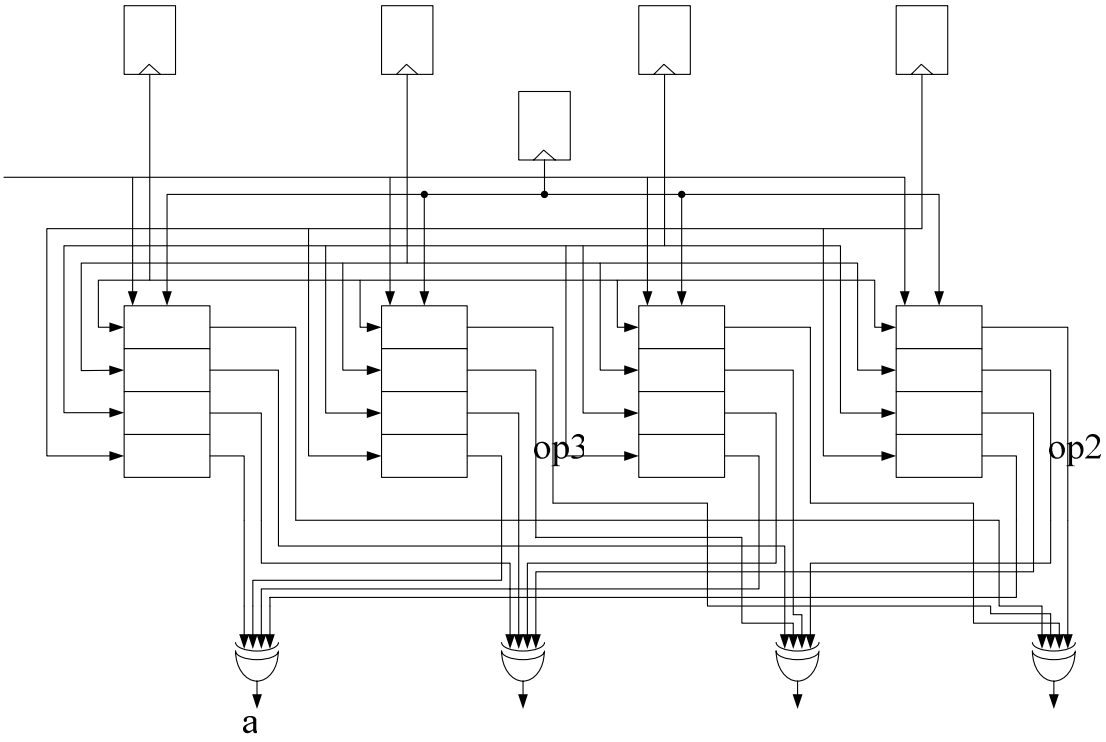

Figure 17.3 GFM 16x8x8

For example, suppose that the following multiplication has to be performed:

$$\begin{bmatrix} op33 & op32 & op31 & op30 \\ op23 & op22 & op21 & op20 \\ op13 & op12 & op11 & op10 \\ op03 & op02 & op01 & op00 \end{bmatrix} \bullet \begin{bmatrix} a3 \\ a2 \\ a1 \\ a0 \end{bmatrix} = \begin{bmatrix} op33 \cdot a3 \oplus op32 \cdot a2 \oplus op31 \cdot a1 \oplus op30 \cdot a0 \\ op23 \cdot a3 \oplus op22 \cdot a2 \oplus op21 \cdot a1 \oplus op20 \cdot a0 \\ op13 \cdot a3 \oplus op12 \cdot a2 \oplus op11 \cdot a1 \oplus op10 \cdot a0 \\ op03 \cdot a3 \oplus op02 \cdot a2 \oplus op01 \cdot a1 \oplus op00 \cdot a0 \end{bmatrix} = \begin{bmatrix} c33 & c23 & c13 & c03 \\ c32 & c22 & c12 & c02 \\ c31 & c21 & c11 & c01 \\ c30 & c20 & c10 & c00 \end{bmatrix}$$

There are sixteen internal GF(2^8) multiplications, whose intermediate results are xored. GF multiplier 16x8x8 can concurrently compute all of them in just 1 clock cycle. Note that circuit complexity has been increased, affecting slightly its maximum operating frequency, however computing clock cycles remain the same as in GF multiplier 8x8.

4.3.6 32-bit Multiplier Modulo 2^{32} (MM)

Newest FPGAs, such as the Virtex 4, Spartan 3 [57] and Virtex 2 series, have embedded 18x18 multipliers in order to maximize performance. In addition, Virtex 4 FPGAs have a new block called “XtremeDSP slice” that intergrades an 18x18 multiplier along with a 48x48 adder. Reference [58] has an application note, which was used, on how to form a

32x32 multiplier from these smaller ones; however its 32 MSBs were omitted, in order to perform modulo 2^{32} computations, as it is shown in Figure 18.

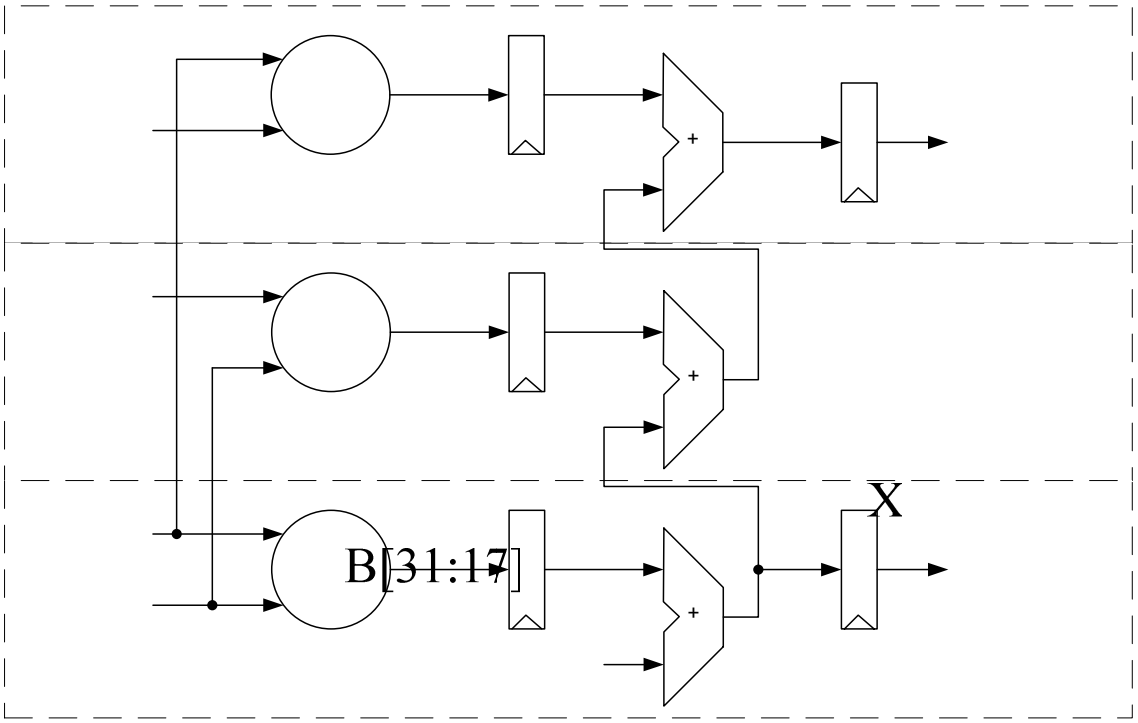

Figure 18 – 32x32 multiplier modulo 2^{32}

As it was mentioned, in each slice there is an 18x18 multiplier, resulting in the utilization of three such units along with their respective adders. Slices 1 and 3 are used to produce the final result's bits, while slice 2 to compute an intermediate product. It should be also noted that every slice has, among others, a register between multiplier and adder called "M", plus one before each output called "P" and have been used to increase its maximum operating frequency. These registers in combination with the external "R" one, lead to a cost of 2 computing clock cycles per modulo multiplication.

A[31:17]
X
M

4.3.7 Cipher Sboxes

As we stated in section 4.1, in CCproc's first version have been used specific cipher Sboxes, instead of a more general and flexible scheme. This section focuses on the description of these Sboxes separately for every cipher, and their integration to the entire design. All Sboxes have been placed to each cluster's memory stage, with the exception of Twofish, where a small portion is also in execution stage for reasons explained below. It should be noted that there is no reference to RC6 cipher, because it does not utilize any Sboxes at all.

A[16:0]
X
M
B[16:0]

- AES cipher Sboxes

Rijndael algorithm uses, one could say, the simplest structure for its Sboxes comparing to the other AES round two finalists. They consist only of two 256x8 Sboxes, one used for the encryption and one for the decryption process.

Figure 19 shows how Sboxes have been implemented in CCproc's each cluster, where "E" stands for encryption and "D" stands for decryption mode. In order to maximize parallelism, every 32-bit data that come through the "In" signal, are separated to four bytes each utilizing a Sbox. After Sbox access, there have been produced four bytes from E-Sboxes and four from D-Sboxes, which are concatenated into two 32-bit words. Finally, depending on the cipher mode, multiplexer A, which is controlled from "mode" signal, passes the appropriate word to the exit.

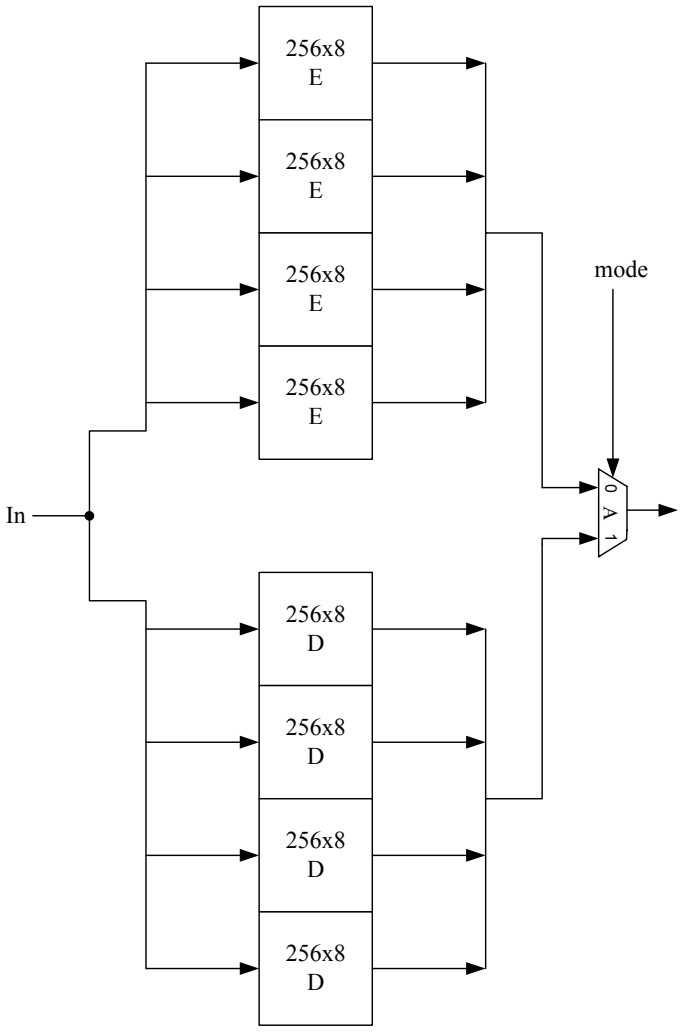

Figure 19 – AES Sboxes in CCproc

- Twofish Sboxes

Twofish uses a different Sbox structure from others ciphers, in a way that its final result depends on the secret key that is being used. Also Twofish uses the same Sbox structure for both encryption and decryption process. Figure 20 shows its Sbox structure, where "S0" and "S1" are two of the subkeys, "In" is the Sbox input, and "q0" and "q1" are 8x8 Sboxes.

In the beginning we planned to place the entire Twofish Sbox structure to the Sbox stage as a combinational circuit. After evaluating this implementation, we quickly discovered that there was a considerable negative impact to the Sbox stage's operating frequency, so a next attempt was to split Figure 20 in two pieces by inserting a register in the second column. Although this alteration improved frequency, it also increased Sbox latency to 2 clock cycles, resulting to a significantly lower cipher performance. Finally we decided to keep the initial implementation (Figure 20), but to place first half in the execution stage and the second half in sbox stage, as shown in Figure 21. Note that the portion that belongs to execution stage, it did not insert any considerable negative consequences, because it was placed in parallel with the other functional units, as it is shown in Figure 22.

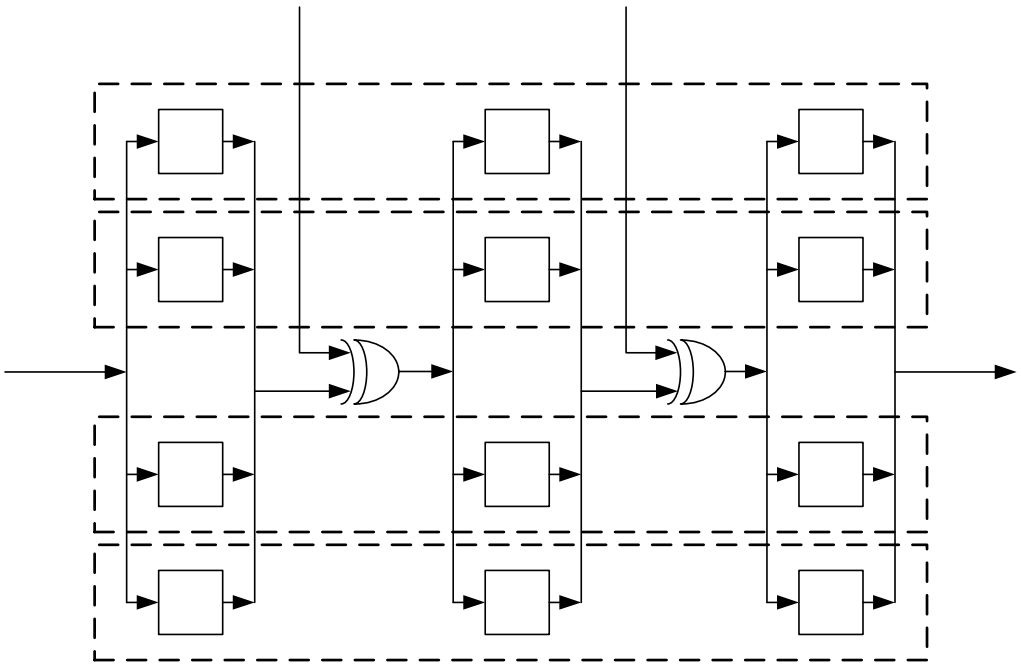

Figure 20 – Twofish Sboxes

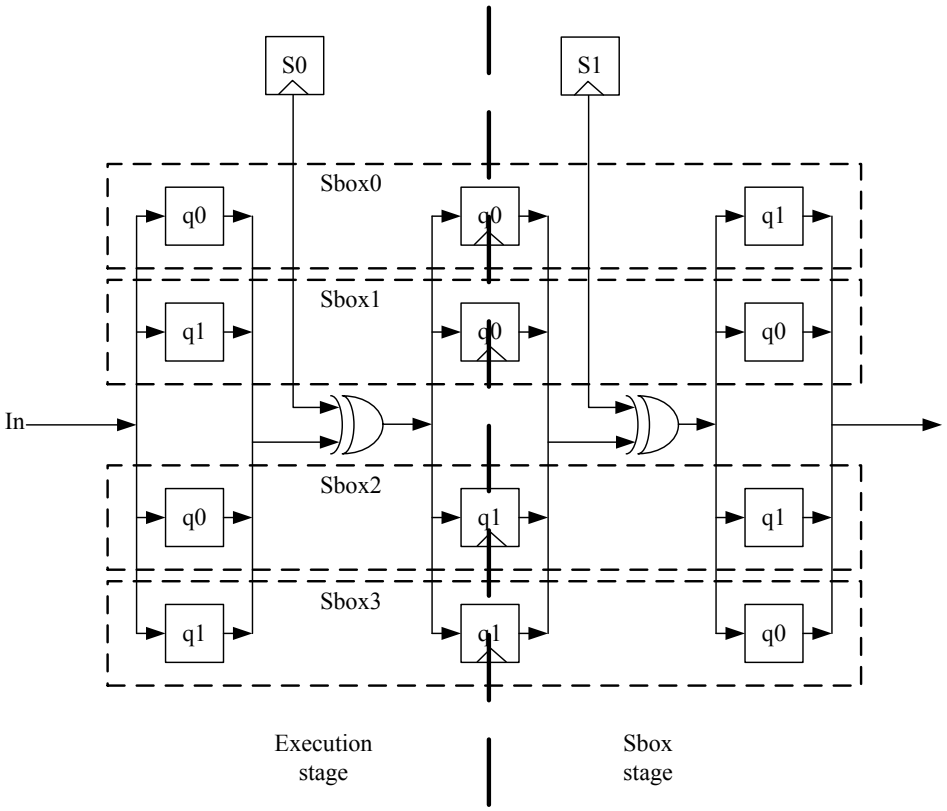

Figure 21 – Twofish Sboxes in CCproc

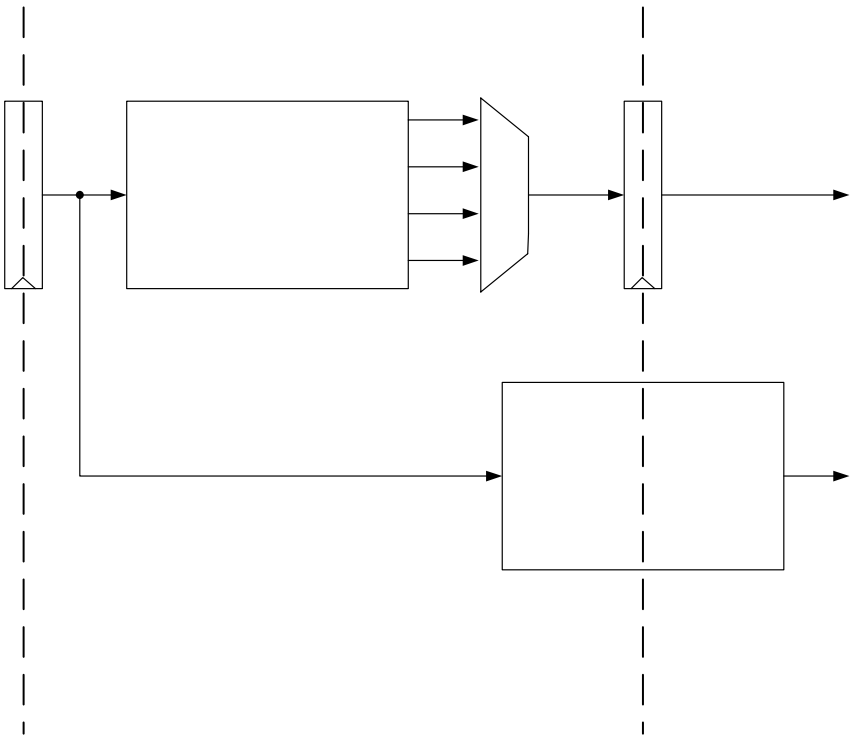

Figure 22 – Implementation of Twofish Sboxes in datapath

- Serpent Sboxes

Serpent is the only cipher that uses permutations in its beginning and end of processing, however these steps are omitted in bitslice mode to be more efficient, so CCproc uses this mode. Sbox access is somehow different than other ciphers and will be presented through the following example.

Suppose that X0, X1, X2, X3 are the four 32-bit words of plaintext where X0 is the most significant one, and consider that each word's the four MSBs in hexadecimal are:

X0 = hex"6…", X1 = hex "a…", X2 = hex "f…", X3 = hex "8…"

Table 8 shows these numbers also in binary while each column indicates the respective bit. Last column "weight" shows the value that emerges when computing each column's in decimal.

hex	bit 31	bit 30	bit 29	bit 28	weight
6	0	1	1	0	2^0
a	1	0	1	0	2^1
f	1	1	1	1	2^2
8	1	0	0	0	2^3

Table 8 – Serpent Sbox access example

For example, "bit31" $= 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 14$, which is the Sboxj's access address, where j is the round number. If j = 0, then Sbox0 [14] = 9. Similarly the other columns emerge the following values for j = 0:

"bit30" $= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 5$, Sbox0 [5] = 6

"bit29" $= 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 7$, Sbox0 [7] = 11

"bit28" $= 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 4$, Sbox0 [4] = 10

In Table 9, "bit" columns contain the above results, in binary according the "weight" column. Indeed "bit31" = 9, "bit30" = 6, "bit29" = 11 and "bit28" = 10.

hex	bit 31	bit 30	bit 29	bit 28	weight
a	1	0	1	0	2^0
7	0	1	1	1	2^1
4	0	1	0	0	2^2
b	1	0	1	1	2^3

Table 9 – Serpent Sbox results example

Finally, if the resulting lines are considered as binary values, with each cell in "bit31" column containing the MSB, column "hex" translates them to hexadecimal and these are the final replacements: 6↔a, a↔7, f↔4, and 8↔b.

As we can see, Serpent requires all four 32-bit words at the same time, in case a simultaneous Sbox access is desired, leading to an implementation shown in Figure 23.

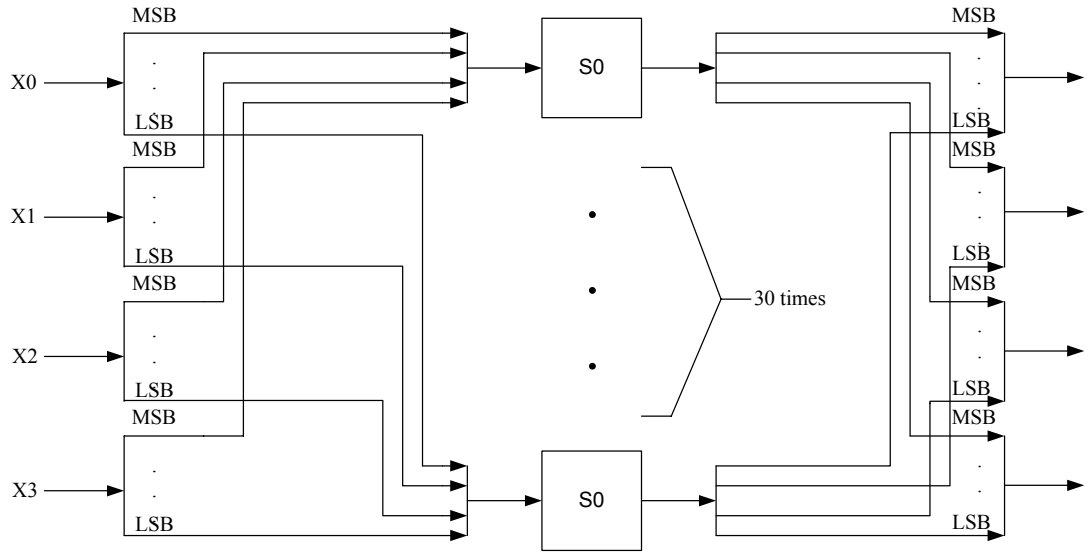

Figure 23 – Serpent Sboxes in CCproc

The above design is the same for the other Sboxes as well. In order to finally select the appropriate one depending on process round, there has been implemented a small counter, which is enabled each time there is a Sbox access. It should be noted that again there is no need for an additional instruction argument for Sbox access, because it is always serial, so the previously mentioned simple 3-bit counter suffices.

- MARS Sboxes

MARS algorithm uses two Sboxes S0 and S1, 256x32 each, however it has the unique property that they are also used concatenated as one 512x32 Sbox. This is another case where all four 32-bit plaintext words are needed at the same time, each one accessing the same structure in its own way. As a result, we decided to implement all four of these access combinations to the four clusters, enabling a fully parallel utilization, as shown in Figure 24.

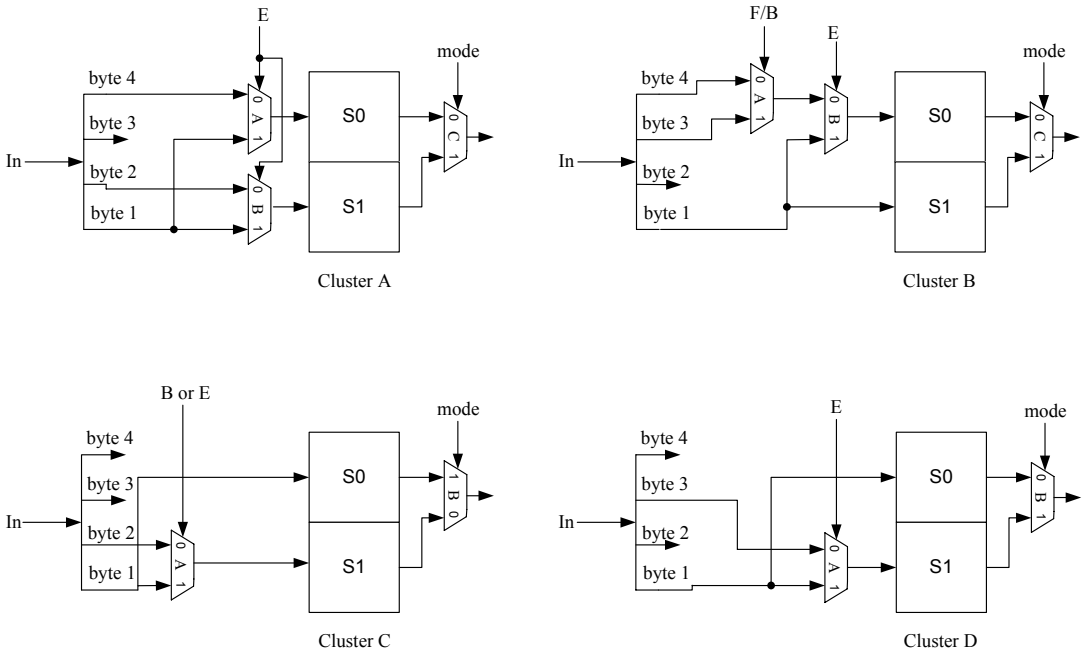

Figure 24 – MARS Sboxes in CCproc

In MARS processing, in each cluster appears one 32-bit word that passes through these four structures, resulting in four simultaneous Sbox accesses in 1 clock cycle. Before Sbox access there are one or two multiplexers, which in combination with control signals "E", "F" and "B", pass the appropriate byte. Depending on cipher's current processing mode, again a control signal "mode" selects the required 32-bit word.

From the above Sbox analysis and implementation description, there are many possible structures that a cipher may utilize. As a result it was initially decided to design specific cipher Sbox structures, which of course may be replaced with more general and flexible ones in future CCproc's versions.

4.3.8 Putting it all together: CCproc VLIW symmetric cipher co-processor

After describing each functional unit and Sbox structure in detail, this section puts them all together to assemble CCproc VLIW cryptography co-processor. As it was mentioned in section 4.1, after deep cipher analysis, it was decided to build a VLIW co-processor with four clusters, as shown in Figure 25.

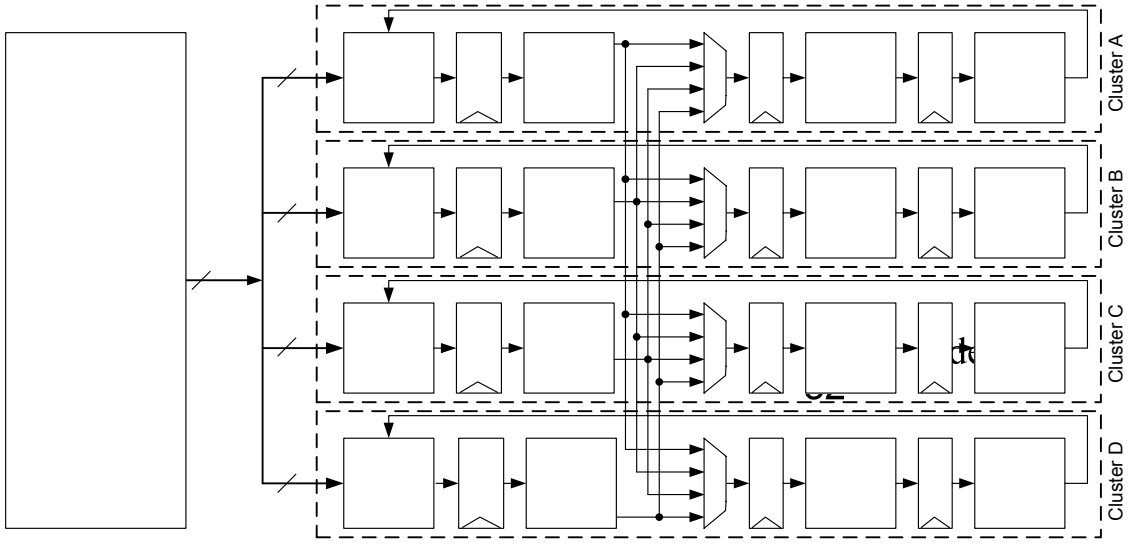

Figure 25 – CCproc's schematic overview

First of all, there is an instruction fetch unit, which takes on to fetch a 128-bit data value and pass it to the four clusters as four 32-bit instructions. Bits 127 down to 96 form cluster's A instruction, bits 95 down to 64 form cluster's B instruction, bits 63 down to 32 form cluster's C instruction and bits 31 down to 0 form cluster's D instruction. As it can be seen from Figure 26, there is an instruction cache 256x128 size, where quads of 32-bit instructions are stored. PC is the program counter register that holds the instruction cache's access address. Multiplexer A selects with "nPCsel" between PC address and an address "label" generated from the "loop controller" unit, in case there is a loop instruction, as it was described in 4.3.1, while multiplexer B selects again with "nPCsel" between "label" and PC to pass into the adder for next instruction's address effective calculation.

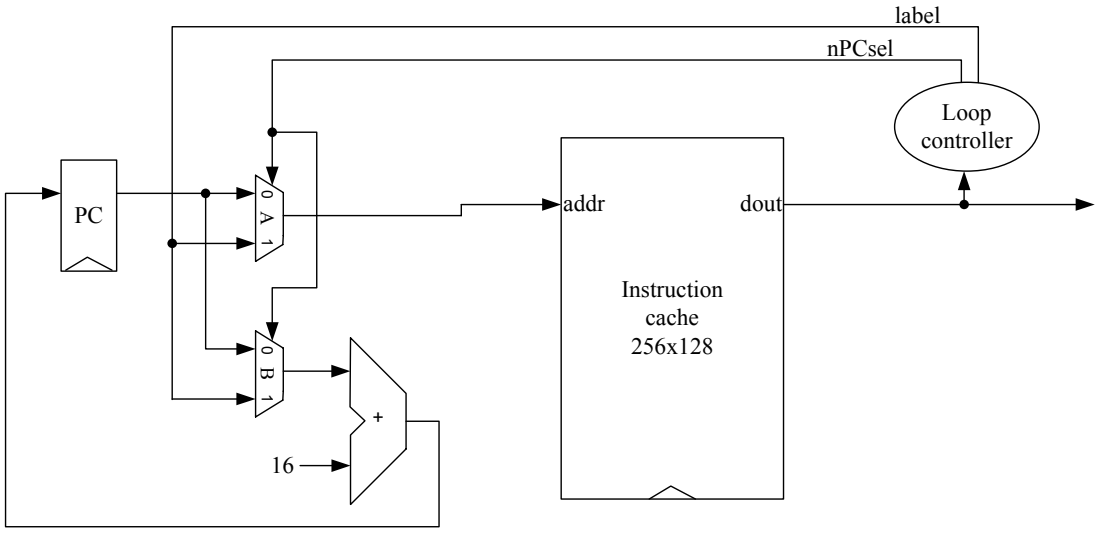

Figure 26 – CCproc's instruction fetch unit

After a 128-bit data value has been fetched, it is separated to four 32-bit instructions that are directly connected with each cluster's decode stage, whose high level schematic is shown in Figure 27.

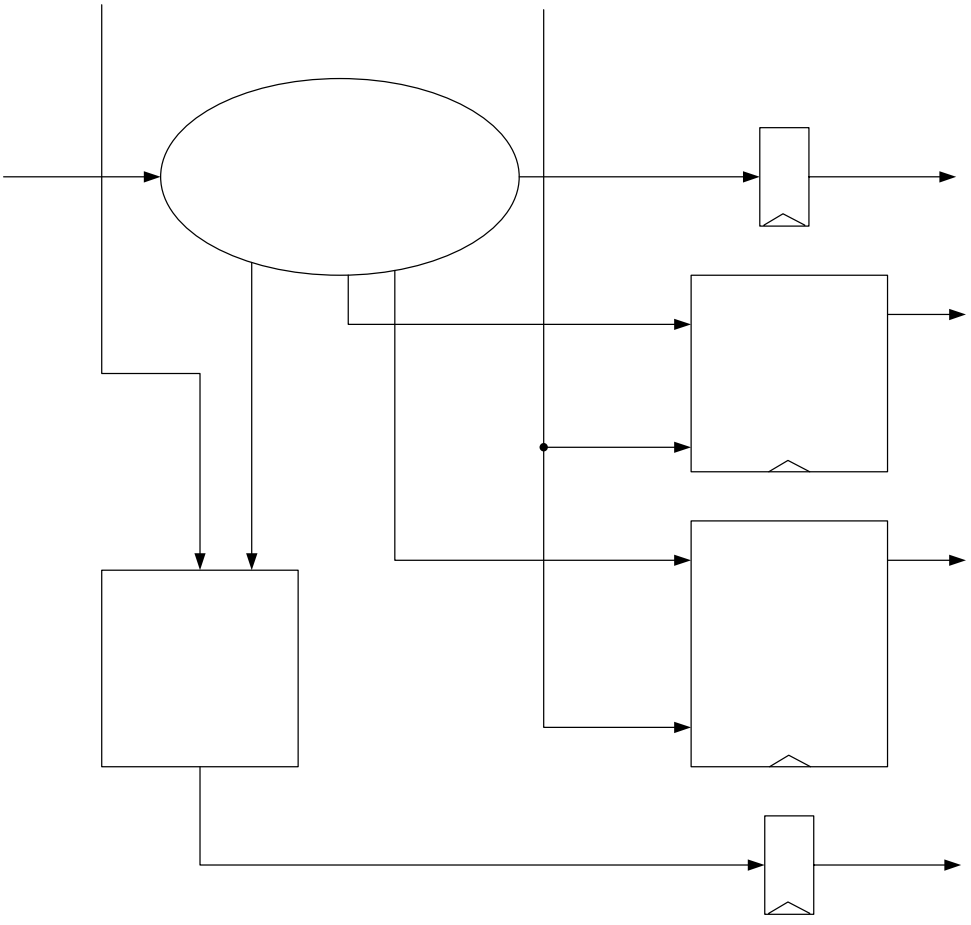

Figure 27 – Decode unit

The main unit in this stage is the Decode controller, which decodes each 32-bit "Instr" instruction comes from the "instruction fetch" unit. It produces valid RF and KRF addresses, plus many other control signals that pass through the next stages via the R1 pipeline register. "AluOutWB" contains every data that will be stored to RF or KRF. Finally there is an "Address Comparator" unit that compares "addresses" signals, which contain RF write addresses to next stages with current's instruction target register in RF. Every control signal that this unit produces, pass through pipeline register R2. It should be noted that R1 and R2 pipeline registers have the same meaning with the R one between "decode" and "execution" stages in Figure 25.

Next stage is the execution stage, where all logic and arithmetic operations are performed and is shown in Figure 28. The ALU, GF multiplier 16x8x8 and MM functional units are all placed here. RF outputs that have come from the previous pipeline stage are their inputs and depending on the operation that needs to be performed, multiplexer A selects the appropriate result. As it was shown in Figure 25, before the next pipeline register, there is a "MX" (X=A, B, C, D) 4-to-1 multiplexer that selects among the multiplexer's A outputs of each cluster's execution stage. These multiplexers are used in case data need to be switched between clusters, by using the appropriate "move" instruction, as described in section 5.1. Also in this stage there is Twofish's Sboxes first portion, as was described in section 4.3.7.

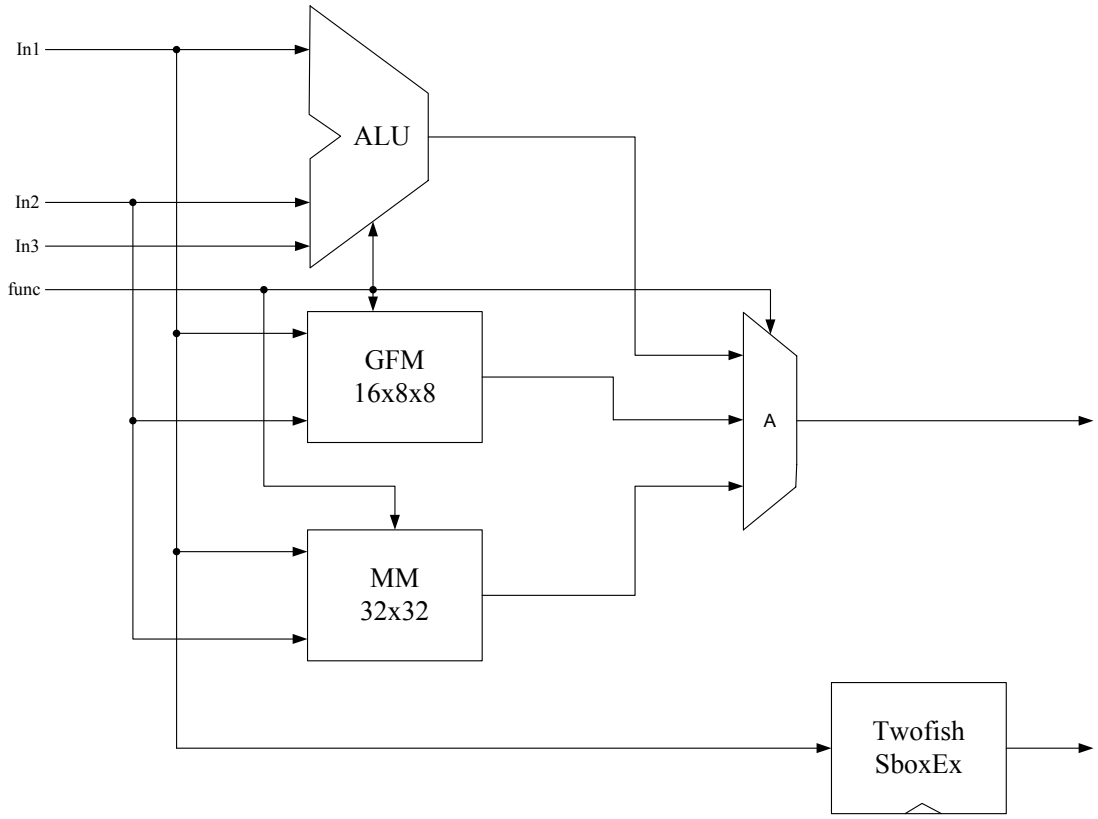

Figure 28 – Execution stage

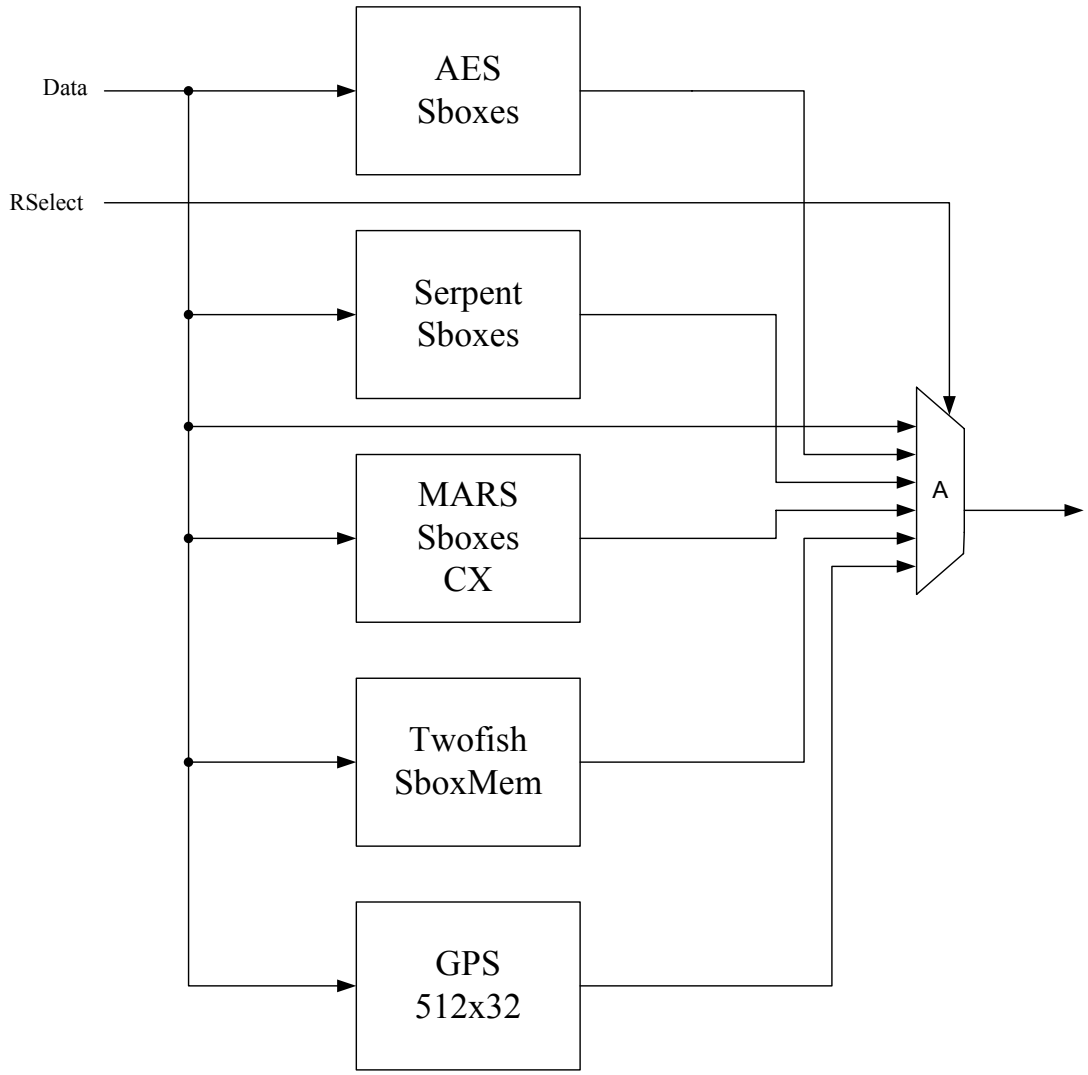

Figure 29 – Sbox stage

4.4 Efficient Data Exchange Among Clusters

In this section we focus on describing how data can be switched among clusters through some instruction examples. Some instructions may have an "mx" (x=a, b, c, d) prefix before them that indicates from which cluster data will be expected. An example is the following instruction quad:

```
CA: mdrd r1,r1   --A<-D
CB: mard r1,r1   --B<-A
CC: mbrd r1,r1   --C<-B
CD: mcrd r1,r1   --D<-C
```

"CX" (X=A, B, C, D) indicates the cluster that the respective instruction will pass through. "Rd" is a pseudo-instruction that reads a register, while in fact performs an

"add r0, rsa, r0" instruction. In cluster's A instruction, putting "md", consequences to select data from cluster D to pass to sbox stage. The same also happens for the other three clusters as shown in Figure 30. This is a very efficient way to switch data among clusters, because it consumes only 1 clock cycle. We should note that also all other R format operations (except double-instructions) can be performed before data switch between clusters, whose entire list is shown in Appendix A.

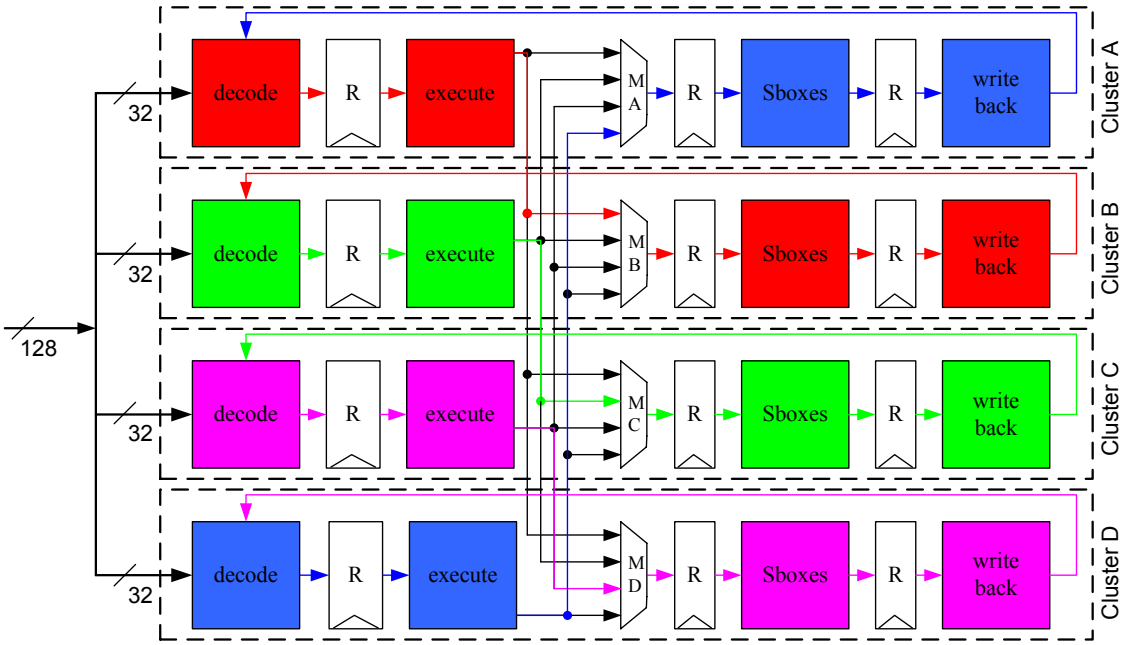

Figure 30 – Data exchange among clusters

Another case is when a cluster is needed to broadcast data to all other clusters. An example is the following instruction quad:

```
CA: mdnop r2   --r2<-CD.r1
CB: mdnop r2   --r2<-CD.r1
CC: mdnop r2   --r2<-CD.r1
CD: rd r1
```

In cluster D register r1 is read and all others will do a "nop" operation. But because there is an "md" prefix before "nop" multiplexers "MA, "MB" and "MC" will select cluster's D r1 to pass to the respective Sbox stage, as shown in Figure 31.

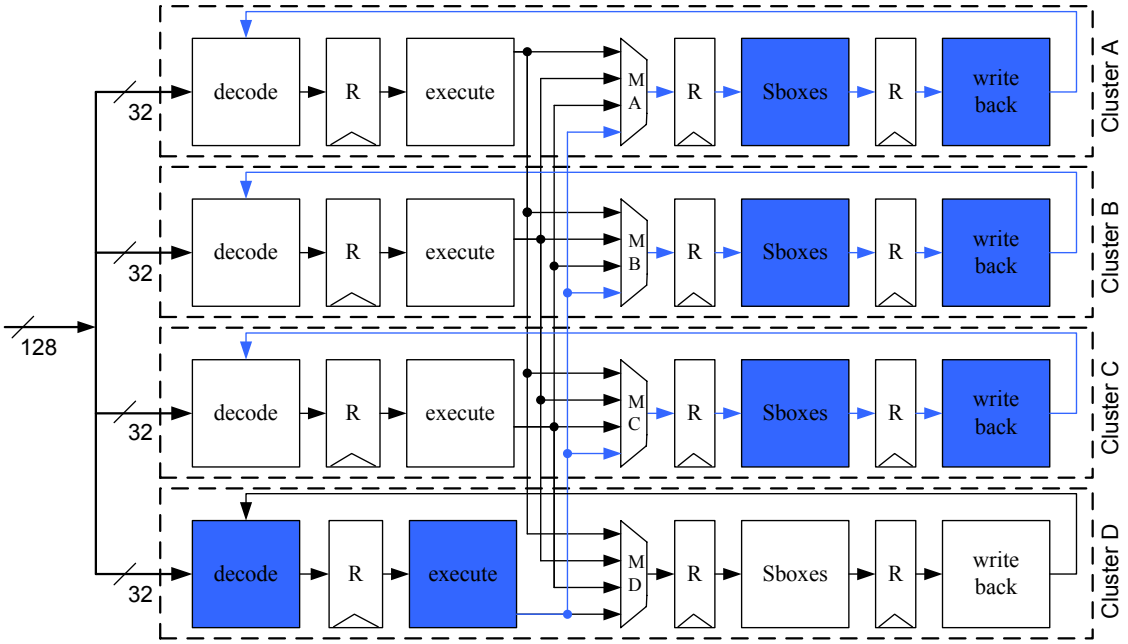

Figure 31 – Data broadcasting

5. Verification and Performance Evaluation of CCproc

After the detailed description of CCproc's ISA and datapath structure, in this chapter we focus on verification tests that were made in order to confirm its functionality, evaluation of hardware results such as operating frequency and occupied area, and finally on a comparison with other similar hardware designs. Note that for synthesis and implementation we used the Xilinx's ISE Foundation Series 7.1i [I6] and for simulation Mentor Graphics Modelsim SE 6.0a [I16].

A first prototype has been built based on Xilinx's Virtex 4 VLX FPGAs resources. Instruction cache, KRF, GPS and MARS Sboxes, have been mapped to single port block memory modules [59], while all other Sboxes have been mapped to distributed memory modules [60]. Also, in order to maximize performance as possible, many other IP cores have been used, such as adders, subtraction units, multipliers and comparators, all generated from Xilinx's Core Generator.

5.1 Verification Tests Using a Python Assembler

In order to perform as many verification tests as possible in a considerable amount of time, it was decided to build an assembler. Among other software languages, we selected Python, because of its ease of usage and remarkable speed. Python has very high level dynamic data types and dynamic typing, plus its software implementations are portable, running on various versions of UNIX, Windows and many other platforms.

Figure 32 shows an abstract schematic of Python assembler. Every time there is a new line, the instruction's format is firstly recognized. Once it is confirmed as a correct CCproc instruction, its arguments are counted, i.e. how many registers, or plain numbers have been used, and then validated for syntax errors. In case an instruction is correct, its arguments are converted to the appropriate binary value. If a new line consists of a label that marks the start point of a loop, its location in the program is stored, until a "loop" instruction uses it as argument. Finally, if a new line consists only of comments (starting with "--"), they are discarded. It should be noted though that comments can be anywhere, except between an instruction text.

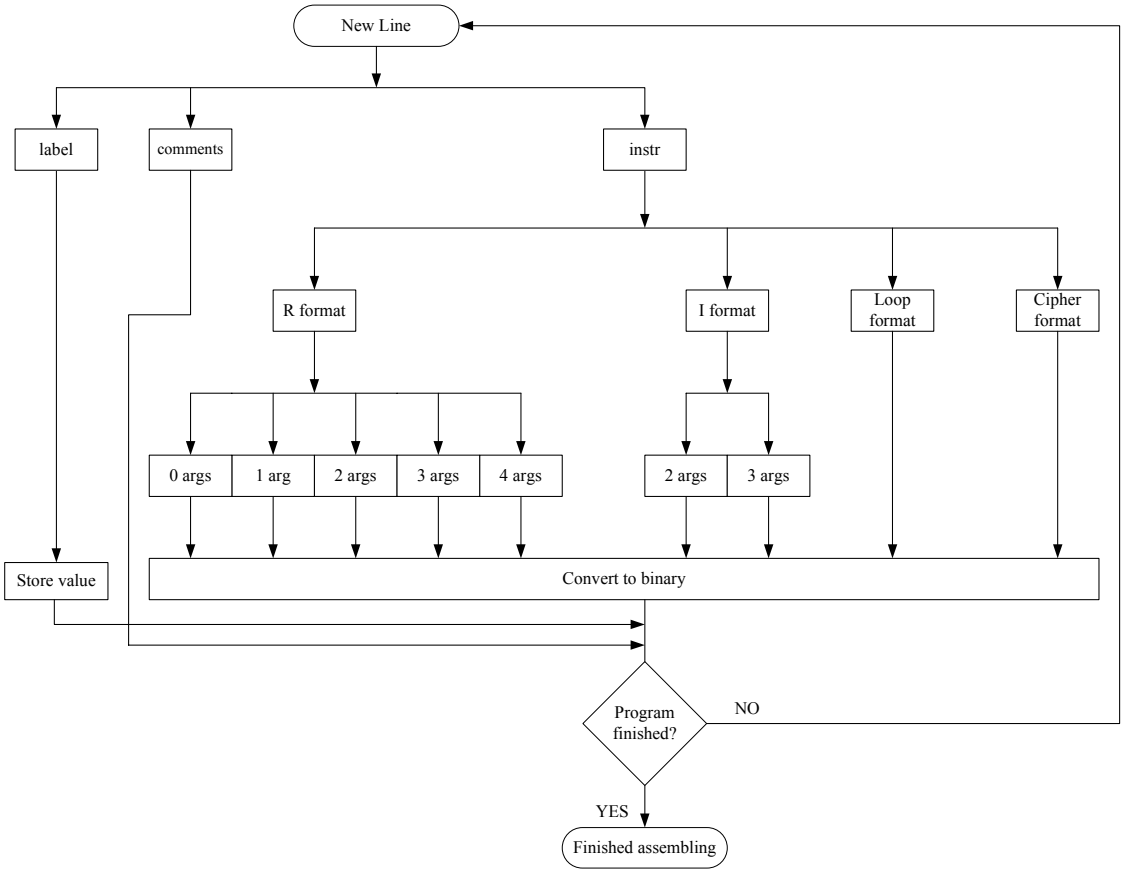

Figure 32 – Python assembler flowchart

In order to verify that every instruction works ok and the correct results are being produced, various tests were written (they are available on the thesis's CD. By using Modelsim and performing simulation tests, both in functional and post-place and route levels, we discovered (and fixed) several implementation problems (bugs).

Once this first verification phase was completed, we wrote five programs that implement the five AES round two finalists. Figure 33 shows an example of RC6's encryption kernel loop and how instructions are written. First it begins with label "encrypt" that indicates a loop's start point and then there are instruction quads, each having on its left "CX" (X=A, B, C, D) indicating the cluster that will pass through.

```
encrypt:
      CA: shli r2,r1,1  --r2<-r1*2
      CB: nop
      CC: shli r2,r1,1  --r2<-r1*2
      CD: nop

      CA: addi r2,r2,1  --r2<-r2+1=r1*2+1
```

```
CB: nop
CC: addi r2,r2,1  --r2<-r2+1=r1*2+1
CD: nop

CA: mmult r3,r1,r2  --r3<-r1*r2=r1*(r1+1)
CB: nop
CC: mmult r3,r1,r2  --r3<-r1*r2=r1*(r1+1)
CD: nop

CA: roli r3,r3,5  --r3<-<<<5(r3)=<<<5(r1*(r1+1)) (u)
CB: nop
CC: roli r3,r3,5  --r3<-<<<5(r3)=<<<5(r1*(r1+1)) (t)
CD: nop

CA: rd r3
CB: manop r2  --r2<-CA.r3=u
CC: rd r3
CD: mcnop r2  --r2<-CC.r3=t

CA: nop
CB: xor r1,r1,r2  --r1<-r1 xor u=C xor u
CC: nop
CD: xor r1,r1,r2  --r1<-r1 xor t=A xor t

CA: rd r3
CB: mcnop r2  --r2<-CC.r3=t
CC: rd r3
CD: manop r2  --r2<-CA.r3=u

CA: nop
CB: rol r1,r1,r2  --r1<-<<<r2(r1)=<<<t(r1)
CC: nop
CD: rol r1,r1,r2  --r1<-<<<r2(r1)=<<<u(r1)

CA: nop
CB: add r1,krfpa,r1  --r1<-r1+S[2*i]
CC: nop
CD: add r1,krfpa,r1  --r1<-r1+S[2*i+1]

CA: mdrd r1,r1  --D<-A
```

```
CB: mard r1,r1  --C<-D
CC: mbrd r1,r1  --B<-C
CD: mcrd r1,r1  --A<-B

CA: loop encrypt
CB: nop
CC: nop
CD: nop
```

Figure 33 – RC6 encryption kernel loop

5.2 Performance Evaluation on Xilinx Virtex 4 FPGA Devices

This section focuses on evaluating CCproc's performance while processing the AES round two finalists. Until now there is only a first prototype built on Virtex 4 FPGAs, which has been successfully verified in post-place and route simulation level. In order to evaluate its performance, first the total number of processing clock cycles needed for each cipher was measured and the results are shown in Chart 11.

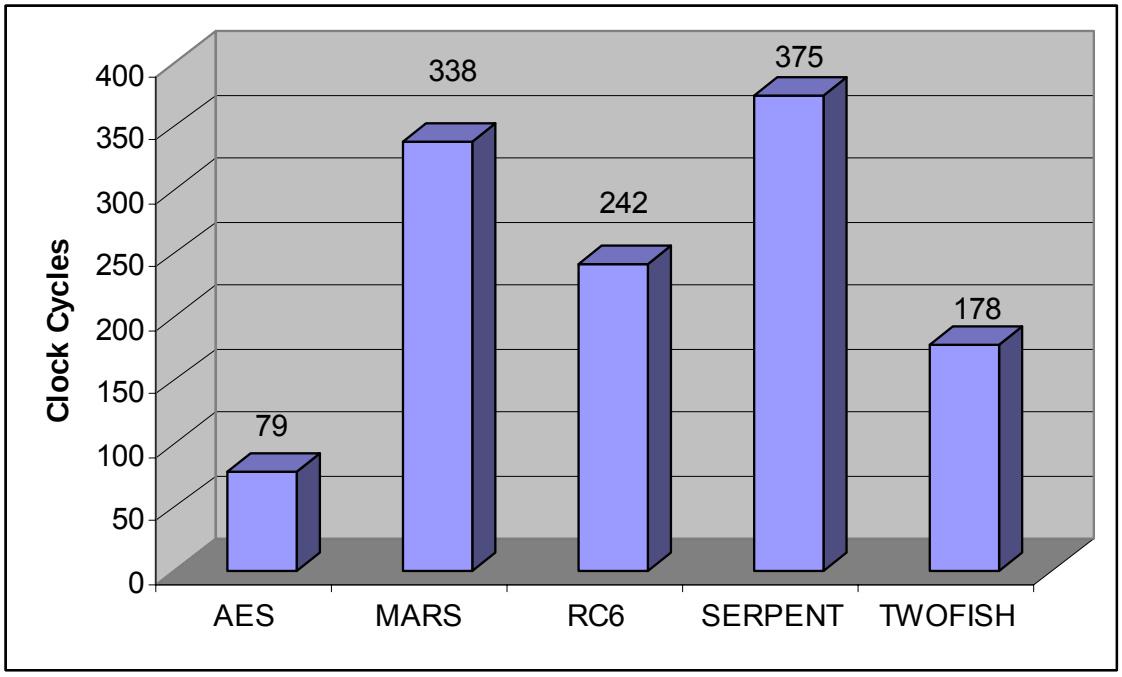

Chart 11 – CCproc's performance in clock cycles for the AES round two finalists

Xilinx XST (Xilinx Synthesis Tool) and ISE 7.1i reported the results shown in Table 10. The XC4VLX40 FPGA is the third smallest in the Virtex 4 series, a fact showing that CCproc is a compact design (275452 gates), capable to fit into today's smaller Virtex 4 FPGAs. The complete set consists of 1, 3 and 4-core implementations mapped

on XC4VLX40, XC4VLX100, XC4VLX160 and XC4VLX 200 FPGAs. It should be noted that devices with speed grade equal to -12, create the fastest implementations.

FPGA	Speed Grade	CCproc Cores	Freq (MHz)	Utilization	Memory Blocks	Xtreme DSP
XC4VLX40	-12	1	108	95%	18	12
XC4VLX100	-12	1		36%	18	12
XC4VLX160	-12	3		77%	54	36
XC4VLX60	-11	1	95	19.6%	18	12
XC4VLX200	-11	4		78.6%	72	48

Table 10 – CCproc's performance statistics

Based on the above performance results, Chart 12 shows the achieved throughput for all AES round two finalists in ECB mode, in each multi-core CCproc implementation. The formula that is used to extract results for 1-core implementations is the one below, where F is the design's operating frequency and cc are the processing clock cycles:

$$Throughput = \frac{128 \cdot F(Mhz)}{cc} Mbits/\sec$$

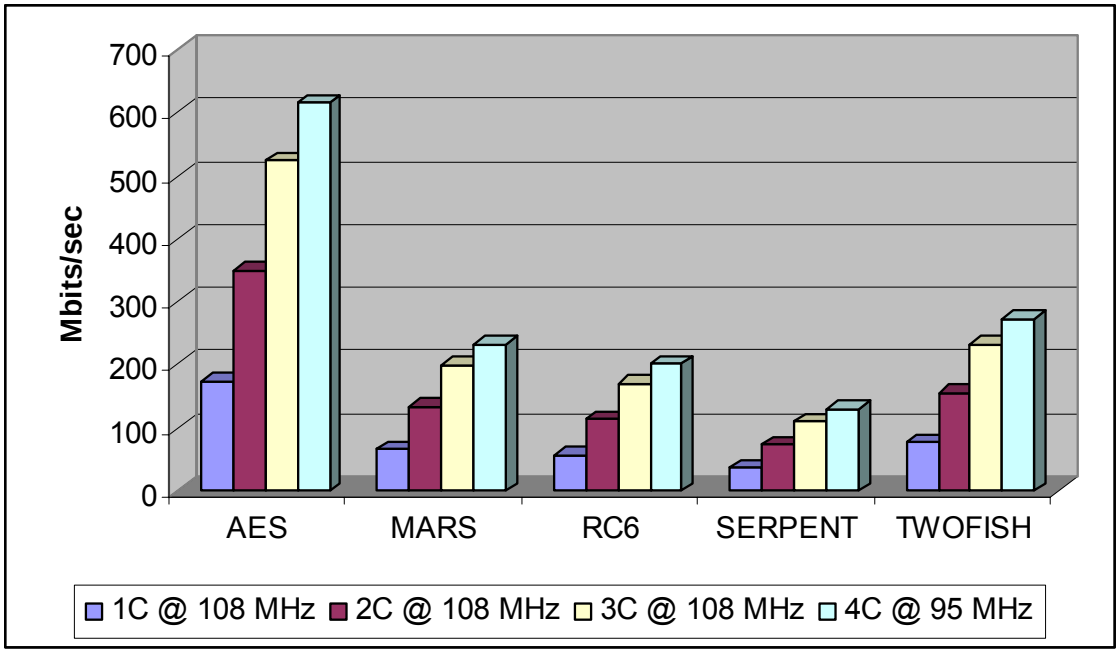

Chart 12 –CCproc Multi-core throughputs in ECB mode

5.3 Performance Comparison with Other Implementations

After presenting CCproc's performance results, this section focuses on making a small comparison between CCproc and the other designs mentioned in section 3.3. However

none of these designs was mapped on an FPGA device, but implemented as ASICs. Consequently CCproc yields a lower operating frequency than them, except from the COBRA design, as shown in Table 11.

Type	Design	Max. Freq. (MHz)
ASIC	Cryptomaniac 4W [50]	360
	Cryptonite [52]	400
	16-SMP [48]	1000
	COBRA [51]	102
FPGA	CCproc	108

Table 11 – Maximum frequencies

As it is confirmed from [61], probably there is no other related design so far (2005), which is not concluded in this comparison. As the authors report and also can be confirmed from Chart 13, Cryptomaniac is the most flexible design, because it supports most of the current ciphers compared to the other designs and has the most aggressive parallelization.

Cryptonite is capable to support one-way hash functions, such as MD5 [63] and SHA-1 [62]. Also in its paper are reported results regarding only AES from 128-bit ciphers, making the least flexible design of all. However there has been developed a specific assembly language, which is demonstrated in its paper, unfortunately only for AES.

COBRA is a design that yields the best results for AES, RC6 and Serpent. However they are valid only when the used COBRA atomic-units are equal to a cipher's rounds and also placed in parallel. For example, when there is one COBRA atomic-unit specifically reconfigured for RC6, it requires 145 clock cycles to complete, but with 20 such atomic-units, clock cycles are reduced to 2 and throughput is increased to 3.9 Gbits/sec. It should be noted that this throughput is valid for ECB mode, i.e. assuming 20 atomic-units working in parallel.

SMP is an approach orthogonal to the above ones, because it demonstrates how many crypto processors can be used to increase cipher processing throughput, by using the ICBC mode. As it can be easily comprehended, this approach can be combined with every other technique that has been presented for the inner cipher specific architecture, resulting to an even better final throughput.

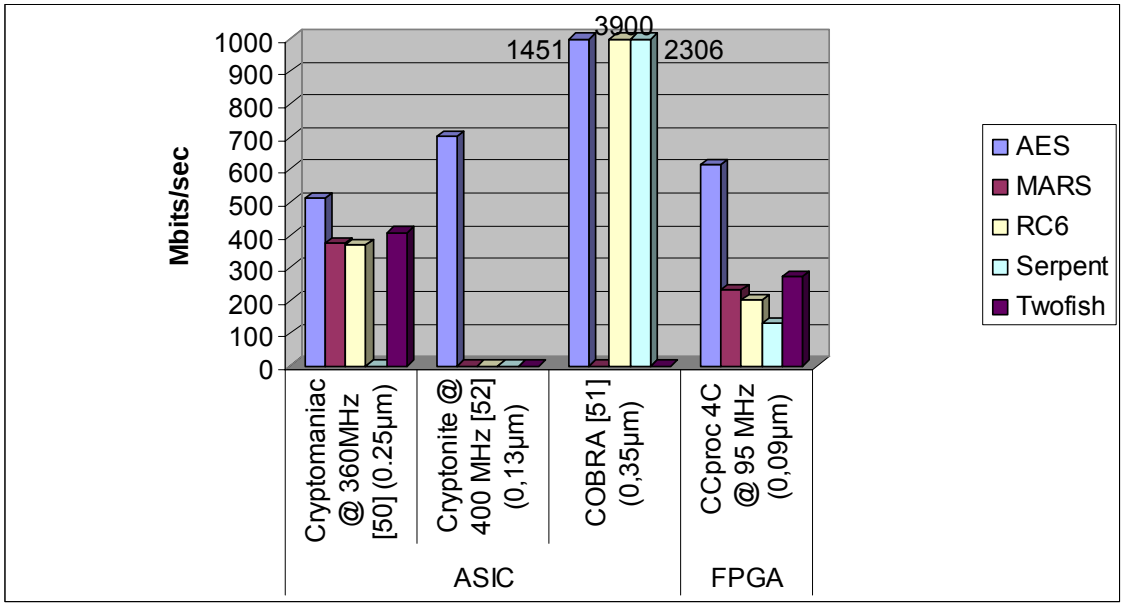

Chart 13 – Performance comparison between CCproc and other designs. This chart shows the best cases for each design.

As it can be seen from Chart 13, CCproc is the only one that supports so far all AES round two finalists. Cryptomaniac is the design that has many similarities with CCproc, so it is natural to make an immediate comparison between them. CCproc yields a better performance for AES cipher comparing to Cryptomaniac, while comparing to other ciphers, where applicable, it achieves a slightly decreased throughput, yet more than capable to saturate any up to 130 Mbits/sec connections, such as the wide used IEEE 802.11g wireless protocol and IEEE 802.3y 100 Mbits/sec Ethernet protocol. Cryptonite yields better results than CCproc and Cryptomaniac where applicable, however it lacks of flexibility. Also it should be reminded that CCproc is the only one mapped on FPGA devices, while all other have been implemented as ASICs. In addition, throughput is by far not the only characteristic that can evaluate a cryptography processor's value, because there many cases where there is no need for increased throughput, but for flexibility or power consumption. CCproc is believed to be a very flexible architecture, because of its efficiently designed assembly, as it was demonstrated in section 5.1. In addition, as Huffmire in [61] mentions, today's cryptography co-processors, among others should have the ability to process a cipher rapidly, support multiple ciphers and have the capability to be reconfigured and upgraded in case a cipher is broken. These aspects are the most important ones, and CCproc is superiorly offering them.

So far we have completed CCproc's first version mapped on various FPGA devices. It would be very useful to estimate its performance as an implemented ASIC in 0.25 μm process. First of all we located CCproc's critical path and found that it passes through the ALU core 1. We then used Synopsys Design Compiler (DC) [I17] to implement it as an ASIC and found that its latency is 3 ns, meaning a maximum frequency at 333 MHz. This fact leads us to the conclusion that a carefully designed VLSI CCproc implementation could achieve an even higher frequency. Chart 14 shows its performance when running at frequencies up to 500 MHz.

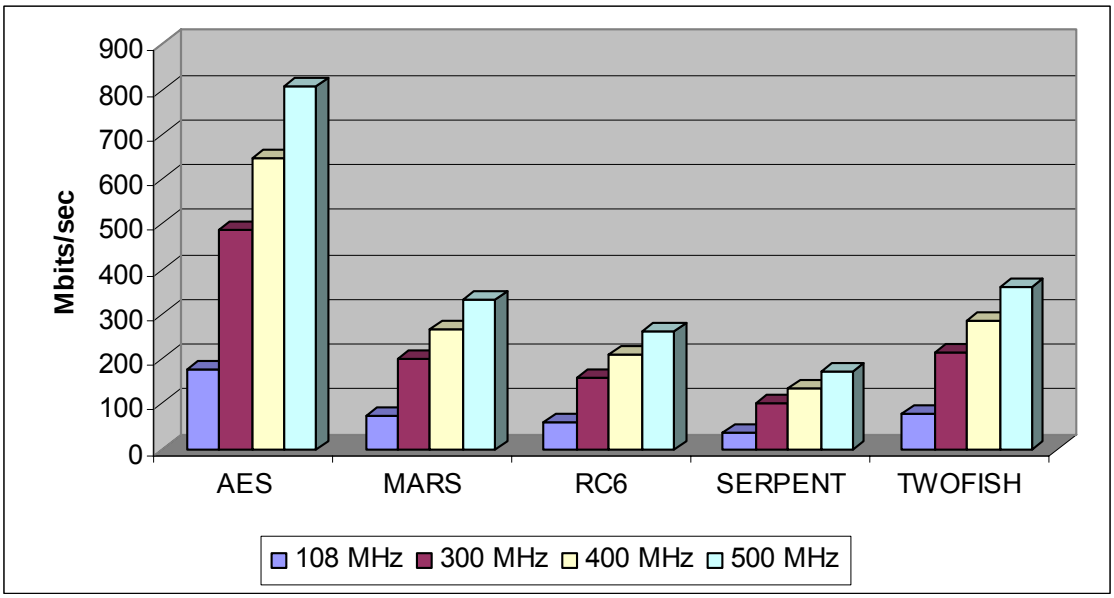

Chart 14 – CCproc's estimated throughput when implemented as VLSI

Chart 15 compares CCproc with other design when there is no parallel cipher processing available, i.e. CBC mode. As we can see, implementations that run at frequencies of 400 MHz and above achieve better throughputs in most cases comparing to other designs. However once again we note that, if a cryptography co-processor reaches a throughput level that enables its usage from a protocol, the next important fact is its flexibility, in order to provide higher security levels. CCproc's first version is already a very flexible cryptography co-processor, which, once a VLSI implementation is built, it can also provide almost the best throughputs.

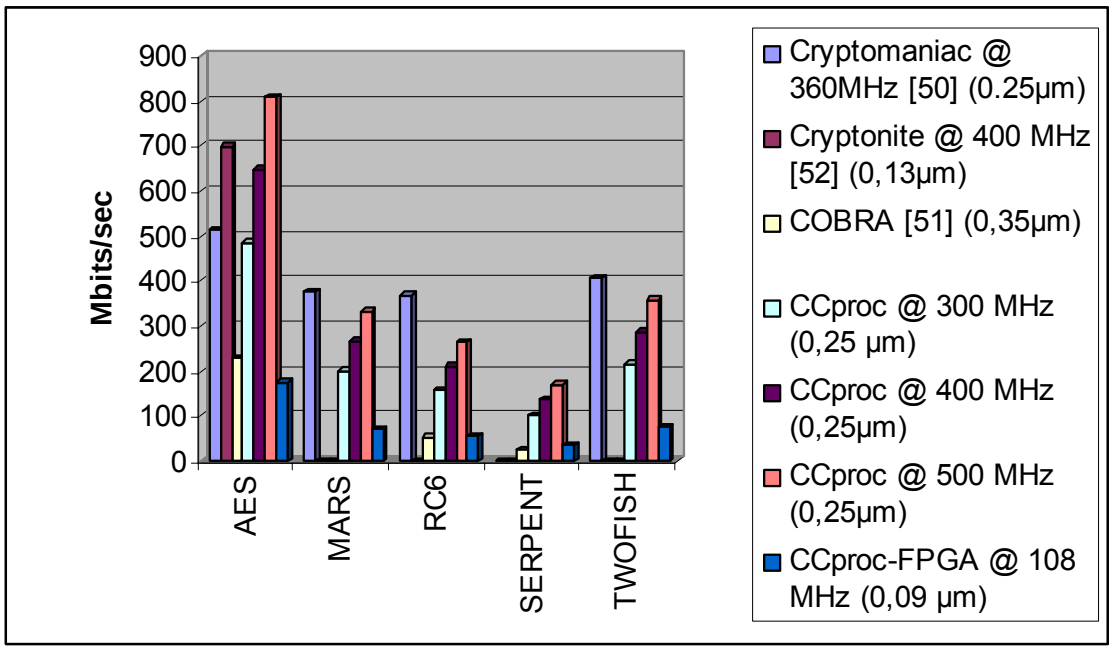

Chart 15 – CCproc performance comparison with other designs without parallel process. This chart shows performance of the current CCProc and the estimated VLSI implementations.

6. Conclusions and Future Work

As we mentioned in the beginning of this thesis report, internet is growing larger over the years that pass. Also various embedded processors are more and more used in many wireless communication devices, such as cell phones, palmtops, PDAs (Personal Digital Assistants), televisions and automobile navigation systems. Consequently, secure communication and confidentiality are crucial, in order to avoid virus infection, privacy loss, and stop digital crime activities from malicious users.

Cryptography is a major issue, which all computer architects should take account when designing new processors that will be used for communication and data exchange. This project was focused on analyzing many cryptographic symmetric ciphers and designing from scratch a VLIW RISC co-processor, in order to efficiently support them in a hardware process level in very competitive speeds. In summary our design has the following characteristics:

- Efficient and flexible ISA capable to support many symmetric 128-bit ciphers
- 4-wide VLIW processor using 128-bit instructions with RISC datapath structure
- Fits in small FPGAs, while multiple CCproc cores can be placed in larger ones to improve cipher performance
- Supports all AES round two finalists
- Achieves an AES performance up to 616 Mbits/sec at 95 MHz in ICBC mode using a 4-core CCproc implementation
- A 1-core CCproc VLSI implementation estimated running at 500 MHz, yields an AES throughput of 800 Mbits/sec
- Capable to saturate wide used protocols such as the 801.11g wireless and 802.3y 100 Mbits/sec Ethernet

This task was pretty difficult, because it demanded to combine many very different algorithms in one common design and also to try and foresee potential needs from future algorithms.

CCproc's first version results are very competitive in comparison with others, as it has been seen from section 5.3. Note also that, once a cryptography co-processor's throughput meets the level that we need, flexibility plays a very significant role to the

security level it provides. CCproc achieves very good cipher speeds, while its ISA offers a high level of flexibility. Also, a very important aspect is that all other designs have been implemented as ASICs, while CCproc mapped on a FPGA device.. In addition, as in every first version, there are a few potential improvements that can be made:

- *More flexible Sboxes structure*. As it was mentioned in previous sections, static specific cipher Sboxes could be replaced by a more flexible and equally fast structure. This improvement may reduce CCproc's hardware utilization, resulting even in an increased operating frequency.
- *Increased throughput with parallel thread support*. Although there are four clusters, which are available for cipher process, as it has been shown in Chart 10, most of the ciphers do not take full advantage of it, resulting in many "nops". These "empty slots" could be used efficiently by the same or another cipher to encrypt a different data block, increasing significantly total throughput more closely to 1-core's theoretical that can be achieved (128 bits/clock cycle). The left portion of Figure 34 shows CCproc's utilization while processing a data block with Twofish. Blue (bright) rectangles indicate a useful instruction, while the red (dark) ones show "nops". With an addition of extra hardware resources and slight modification in current ISA, CCproc may process simultaneously two different data blocks, from the same or different users, by using the same or completely different symmetric ciphers. In Figure 34, in its right portion the blue (bright) rectangles are instructions that process one block, while the green ones (striped) process another block. This is a feature that none from the previously mentioned related works currently supports.

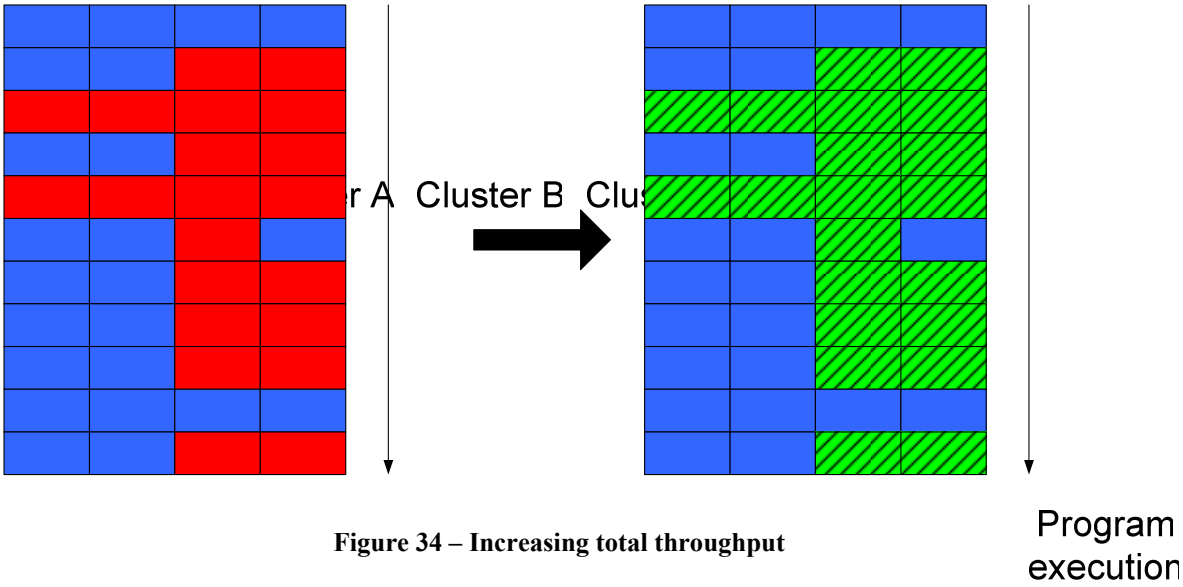

Figure 34 – Increasing total throughput

- *CCproc's ASIC implementation*. So far CCproc prototype has been tested, as mentioned in section 5.2, only in Virtex 4 FPGA devices. It would be very useful if an ASIC implementation were built using Synopsys DC, in order to obtain additional performance and throughput information.

In summary, Cryptography, as it was mentioned in chapter 1, had been used from ancient years to hide important information. Until today, many algorithms have been developed, and always will be, while concurrently many ways are being discovered to unlock even the securest ones. Consequently there are many cases where people have lost or charged a large amount of digital money, while confidential information has been intercepted. That is why computer architects and engineers should always try to protect people from malicious users, by developing new designs resistant against as many as possible types of digital attack.

7. References

[1] Xilinx Corporation, "*Virtex 4 User Guide*", April 11, 2005

[2] Joan Daemen, Vincent Rijmen, "*AES Proposal: Rijndael*", Document Version 2, March 9, 1999

[3] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O'Connor, M. Peyravian, D. Safford, N. Zunic, "MARS - *a candidate cipher for AES*", IBM Corporation, 1999.

[4] Bruce Scheier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson, "*Twofish: A 128-bit Block Cipher*", 15 June 1998, Counterpane Systems.

[5] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, "*The RC6 Block Cipher*", August 20, 1998.

[6] Ross Anderson, Eli Biham, Lars Knudsen, "*Serpent: A Proposal for the Advanced Encryption Standard*", 5th workshop on Fast Software Encryption, 1998.

[7] B. Schneier, "*Description of a new variable-length key, 64-bit block cipher (Blowfish)*", Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994.

[8] Ronald L. Rivest, "*The RC5 Algorithm*", October 1996

[9] MediaCrypt, "*International Data Encryption Algorithm*", Technical Description

[10] B. Preneel, "*Modes of Operation of a Block Cipher*", K.U. Leuven, Belgium

[11] J. L. Smith, "*The design of Lucifer: A cryptographic device for data communications*, " Technical report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., 10598, U.S.A., 1971.

[12] Howard M. Heys, "*A tutorial on linear and differential cryptanalysis*", Memorial University of Newfounland, 2002

[13] David A. Patterson, John L. Hennessy, "*Computer Architecture: A quantitative approach: Second edition*", Morgan Kaufman Publishers, Inc, 1996

[14] John Worley, Bill Worley, Tom Christian, and Christopher Worley, "*AES Finalists on PA-RISC and IA-64: Implementations & Performance*", AES Candidate Conference, 2000, New York, USA

[15] Intel Corporation, "*Intel Itanium Processor*", August 2001

[16] Intel Corporation, "*Intel Pentium Pro family*", 1996

[17] Intel Corporation, "*Intel Pentium II processor at 350MHz, 400Mhz, 450 MHz*", August 1998

[18] Intel Corporation, "*Intel Pentium III processor based on 0,13 micron process up to 1.33 GHz*", December 2001

[19] David L. Weaver, Tom Germond, "*The SPARC architecture manual, version 9*", Prentice Hall Inc., 1994

[20] Compaq Computer Corporation, "*21164 Alpha Microprocessor*", December 1998

[21] Hewlett Packard Corporation, "*The Hewlett Packard PA-RISC 8500 processor*", October 1998

[22] R. Weiss, N. Binkert, "*A comparison of AES candidates on the Alpha 21264*", Third Advanced Encryption Standard Candidate Conference, April 2000

[23] Intel Corporation, "*Intel Pentium 4 processor with 512-KB L2 cache on 0.13 micron process*", February 2004

[24] R.E. Kessler, E.J. McLellan, D.A. Webb, "*The Alpha 21264 microprocessor architecture*", Compaq Computer Corporation, January 1999

[25] AMD Corporation, "*AMD Athlon processor model 4 datasheet*", November 2001

[26] Alireza Hodjat, Ingrid Verbauwhede, "*A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA*", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2004

[27] Xilinx Corporation, "*Virtex-II Pro and Virtex-II Pro X Platform FPGAs*", March 2005

[28] Maire McLoone, John McCanny, "*Rijndael FPGA Implementations Utilizing Look-up tables*", Journal of VLSI signal processing 34, 261-275, 2003.

[29] Xilinx Corporation, "*Virtex-E 1.8V Field programmable gate arrays*", July 2002

[30] P. Chodowiec, P. Khuon, and K. Gaj. "*Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining*", In Symposium on

Field Programmable Gate Arrays – FPGA 2001, pages 94–102. ACM Press, 2001.

[31] Xilinx Corporation, "*Virtex 2.5V Field programmable gate arrays*", April 2001

[32] Alireza Hodjat, David D. Hwang, Bocheng Lai, Kris Tiri, Ingrid Verbauwhede, "*A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18-μm CMOS technology*", Proceedings of the 15th ACM Great Lakes symposium on VLSI, April 17-19, 2005, Chicago, Illinois, USA

[33] Sumio Morioka, Akashi Satoh, "*A 10-Gbps full-AES crypto design with a twisted BDD S-box architecture*", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v.12 n.7, p.686-691, July 2004

[34] Jean-Luc Beuchat, "*FPGA Implementations of the RC6 Block Cipher*", In P.Y. K. Cheung, G.A. Constantinides, and J.T. de Sousa, editors, Field-Programmable Logic and Applications, number 2778 in Lecture Notes in Computer Science, pages 101-110. Springer, 2003

[35] Elbirt et al, "*An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists*", IEEE Trans. of VLSI Systems, 9.4, pp.545-557, August 2001

[36] Ichikawa T., Kasuya, T., Matsui, M., "*Hardware Evaluation of the AES Finalists*", Proc. 3rd Advanced Encryption Standard (AES) Candidate Conference, New York, April 13-14, 2000

[37] J. Elbirt , C. Paar, "*An FPGA implementation and performance evaluation of the Serpent block cipher*", Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays, p.33-40, February 10-11, 2000, Monterey, California, United States

[38] Bora, P. and Czajka, T., "*Implementation of the Serpent Algorithm Using Altera FPGA Devices*", 1999

[39] Altera Corporation, "*Flex 10K embedded programmable logic device family*", January 2003

[40] Pawel Chodowiec, Kris Gaj, "*Implementation of the Twofish Cipher Using FPGA Devices*", ECE, George Mason University, July 1999

[41] Xilinx Corporation, "*XC4000E and XC4000X series field programmable gate arrays*", May 1999

[42] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "*New Results on the Twofish Encryption Algorithm*", Second AES Candiate Conference, April 1999

[43] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "*Performance Comparison of the AES Submissions*", Proc. Second AES Candidate Conference, NIST, March 1999, pp. 15-34

[44] K. Gaj and P. Chodowiec, "*Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Array*" Proc. RSA Security Conf. - Cryptographer's Track, San Francisco, CA, April 8-12, 2001, pp. 84-99

[45] Dandalis, V. K. Prasanna, J. D. Rolim, "*A Comparative Study of Performance of AES Final Candidates Using FPGAs*", Proc. Cryptographic Hardware and Embedded Systems Workshop, CHES 2000, Worcester, MA, Aug 17-18, 2000

[46] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, Edward Roback, "*Report on the Development of the Advanced Encryption Standard (AES)*", National Institute of Standards and Technology, October 2, 2000

[47] J. Burke, J. McDonald, T. Austin, "*Architectural Support for Fast Symmetric-Key Cryptography*", ASPLOS 2000

[48] Praveen Dongara and T. N. Vijaykumar, "*Accelerating Private-Key Cryptography via Multithreading on Symmetric Multiprocessors*", In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2003

[49] M. Jung, F. Madlener, M. Ernst and S. A. Huss, "*A Reconfigurable Coprocessor for Finite Field Multiplication in GF(2^8)*", Darmstadt University of Technology, Germany, IEEE Workshop on Heterogeneous reconfigurable Systems on Chip, Hamburg, April 2002

[50] Lisa Wu, Chris Weaver, and Todd Austin, "*Cryptomaniac: A Fast Flexible Architecture for Secure Communication*", ISCA 2001, June 2001

[51] A. J. Elbirt, C. Paar, "*An Instruction-Level Distributed Processor for Symmetric-Key Cryptography*", IEEE Transactions on Parallel and Distributed Systems, 16(5), pp. 468-480, May 2005

[52] D. Oliva, R. Buchty, and N. Heintze, "*AES and the Cryptonite Crypto Processor*", Proc. Int. Conf. Compiler, Architectures and Synthesis for Embedded Systems, pp. 198-209, Oct. 2003

[53] A. Murat Fiskiran and Ruby B. Lee, "*On-Chip Lookup Tables for Fast Symmetric-Key Encryption*", Proceedings of the IEEE 16th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 356-363, July 23-25, 2005

[54] A. Murat Fiskiran and Ruby B. Lee, "*Performance Impact of Addressing Modes on Encryption Algorithms*", Proceedings of the International Conference on Computer Design (ICCD 2001), pp. 542-545, September 2001

[55] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. "*Fast and portable parallel architecture simulators: Wisconsin Wind Tunnel II*", *IEEE Concurrency*, 2000

[56] Yedidya Hilewitz, Zhijie Jerry Shi, and Ruby B. Lee, "*Comparing Fast Implementations of Bit Permutation Instructions*", Proceedings of the 38th Annual Asilomar Conference on Signals, Systems, and Computers, November 2004

[57] Xilinx Corporation, "*Spartan 3 FPGA family: complete datasheet*", January 2005

[58] Xilinx Corporation, "*Xtreme DSP design considerations user guide*", February 2005

[59] Xilinx Corporation, "*Single port block memory core v6.2*", April 2005

[60] Xilinx Corporation, "*Distributed memory v7.1*", January 2005

[61] T. Huffmire, "*Application of cryptographic primitives to computer architecture*", University of California, Santa Barbara, March 2005

[62] R. H. Brown, A. Prabhakar, "*FIPS180-1: Secure Hash Standard (SHA)*", Federal Information Processing Standards Publication (FIPS), May 1993

[63] R. Rivest. "*RFC1312: The MD5 Message-Digest Algorithm*", April 1992

Internet Links:

[I1] **http://www.internetworldstats.com**

[I2] **http://www.stanford.edu/group/siqss/Press_Release/press_detail.html**

[I3] **http://wp.netscape.com/eng/ssl3/draft302.txt**

[I4] **http://www.codesandciphers.org.uk/enigma/**

[I5] **http://www.ssh.fi/support/cryptography/introduction/**

[I6] **http://www.xilinx.com/**

[I7] **www.fact-index.com/r/rc/rc4_cipher.html**

[I8] **www.aci.net/kalliste/des.htm**

[I9] **http://csrc.nist.gov/CryptoToolkit/aes/round1/round1.htm#algorithms**

[I10] **http://www.cs.ut.ee/~helger/aes/**

[I11] **http://csrc.nist.gov/CryptoToolkit/aes/round1/round1.htm#algorithms**

[I12] **http://www.apple.com**

[I13] **http://palms.ee.princeton.edu/PAX/**

[I14] **http://www.tsmc.com/**

[I15] **http://www.python.org**

[I16] **http://www.model.com**

[I17] **http://www.synopsys.com**

Appendix A: CCproc Complete Instruction Set

Format	Operation	Syntax	Description
R	add	add rdx,rsa,rsb	adds rsa with rsb and stores the result to rdx
	sub	sub rdx,rsa,rsb	subtracts rsb from rsa and stores the result to rdx
	shr	shr rdx,rsa,rsb	shifts right rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	shl	shl rdx,rsa,rsb	shifts left rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	xor	xor rdx,rsa,rsb	logic xor between rsa and rsb, and stores the result to rdx
	rol	rol rdx,rsa,rsb	rotates left rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	and	and rdx.rsa,rsb	logic and between rsa and rsb, and stores the result to rdx
	ror	ror rdx,rsa,rsb	rotates right rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx
	or	or rdx,rsa,rsb	logic or between rsa and rsb, and stores the result to rdx
	gfm	gfm rdx,rsa	galois field multiplication in GF(2^8) between rsa and GF operand x and the result is stored to rdx
	mmult	mmult rdx,rsa,rsb	modulo 2^{32} multiplication between rsa and rsb and the result is stored to rdx
	ldgfopx	ldgfopx rsa	loads Galois Field operand x with rsa's value
	addadd	addadd rdx,rsa,rsb,rsc	adds rsa with rsb, adds the result to rsc and stores it to rdx
	subadd	subadd rdx,rsa,rsb,rsc	subtracts rsb from rsa, adds the result to rsc and stores it to rdx
	addsub	addsub rdx,rsa,rsb,rsc	adds rsa with rsb, subtracts rsc from the result and stores it to rdx
	subsub	subsub rdx,rsa,rsb,rsc	subtracts rsb from rsa, subtracts rsc from the result and stores it to rdx
	xoradd	xoradd rdx,rsa,rsb,rsc	logic xor between rsa and rsb, adds the result to rsc and stores it to rdx
	r2c/c2r	r2c / c2r rdx,rsa	toggles between rows and columns in a 128-bit data value
	xorsub	xorsub rdx,rsa,rsb,rsc	logic xor between rsa and rsb, subtracts rsc from the result and stores it to rdx
	addxor	addxor rdx,rsa,rsb,rsc	adds rsa with rsb, logic xor between rsc and the result and stores it to rdx
	subxor	subxor rdx,rsa,rsb,rsc	subtracts rsa with rsb, logic xor between rsc and the result and stores it to rdx
	ldgfmr	ldgfmr rsa	loads 8-bit mr register with rsa's value, which holds modulo polynomial in Galois Field multiplication
	xorxor	subxor	logic xor between rsa and rsb, logic xor

		rdx,rsa,rsb,rsc	between rsc and the result and stores it to rdx
	krfpaz	krfpaz	resets KRF's pointer to first address
I	addi	addi rdx,rsa,#a	adds rsa with #a and stores the result to rdx
	subi	subi rdx,rsa,#a	subtracts #a from rsa and stores the result to rdx
	shri	shri rdx,rsa,#a	shifts right rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	shli	shli rdx,rsa,#a	shifts left rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	xori	xori rdx,rsa,#a	logic xor between rsa and #a, and stores the result to rdx
	roli	roli rdx,rsa,#a	rotates left rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	andi	andi rdx.rsa,#a	logic and between rsa and #a, and stores the result to rdx
	rori	rori rdx,rsa,#a	rotates right rsa by the amount specified from #a's 5 LSBs and stores the result to rdx
	ori	ori rdx,rsa,#a	logic or between rsa and #a, and stores the result to rdx
	ld	ld rdx,#a(rsa)	loads from data memory address rsa+#a to rdx
	st	st rsb,#a(rsa)	stores to data memory address rsa+#a register rsb
	lui	lui rdx,#a	loads 16-bit value #a to rdx's 16 MSBs
	ldlc	ldlc #a	loads 6-bit lc register with #a
loop	loop	loop label	jumps to the beginning of a loop which starts at address “label”
pseudo-instructions	rd	rd rsa	reads rsa, i.e. adds rsa with r0, but does not store any result
	nop	nop	no operation, i.e. adds r0 with r0, but does not stores any result
move	mx	mx<instr> rdx,rsa,rsb,rsc	current cluster passes data to sbox stage from x cluster (x=a, b, c, d). <instr> will be performed between rsa, rsb and rsc, but result will not be stored in rdx. Instead it can be used from other cluster as well. <instr> may have the following values: add sub shr shl or rol and ror xor addadd addsub addxor

			subadd subsub subxor xoradd xorsub xorxor
cipher	aesX	aesX rdx,rsa	Sbox access during AES encryption or decryption (X=E,D) with rsa and the result is stored to rdx
	marsX	marsX rdx,rsa	Sbox access during MARS forward mode, backward mode, or E function (X=F,B,E) with rsa and the result is stored to rdx
	serX	serX rdx,rsa	Sbox access during Serpent encryption or decryption (X=E,D) with rsa and the result is stored to rdx
	tX	tsld rsa,rsb / tsbox rdx,rsa	during Twofish, loads to S0 and S1 rsa and rsb respectively / Sbox access with rsa and the result is stored to rdx

Appendix B: Setup and Usage of Assembler and CAD Tools

This appendix focuses on numbering the appropriate steps that should be taken, in order the user to perform a CCproc's simulation. As it was mentioned in chapter 5, the CAD (Computer Aided Design) tools that were used, are Xilinx's ISE Foundation Series 7.1i with service pack 3, plus Core Generator for IP core generation. Additionally, XST was used for synthesis and Modelsim SE 6.0a for simulation. Finally Python 2.4.1 was installed in order to develop CCproc's assembler.

The suggested steps that should be made are the following:

1. Launch Core Generator and create the "*.xco" files as there are in this project's CD.
2. Create a valid "imem256x128.caf" file consisting of a cipher written in CCproc's assembly language. Once this is done, use IDLE Python to create an "imem256x128.coe" file consisting of respective 0s and 1s.
3. Through Core Generator initialize instruction cache "imem256x128" with the above "imem256x128.coe" file.
4. Create a testbench for the top-level file "cryptium2.vhd" and inside ISE run Modelsim.

It should be noted that if the Python assembler raises an error, possibly there is a wrong syntax somewhere in "imem256x128.caf". User may check the respective message that IDLE created in order to find its location inside the text.