CRYPTONITE – A Programmable Crypto Processor Architecture For High-Bandwidth Applications

Rainer Buchty

Institut für Informatik
der Technischen Universität München

Lehrstuhl für Rechnertechnik und
Rechnerorganisation

Cryptonite – A Programmable Crypto Processor Architecture For High-Bandwidth Applications

Rainer Buchty

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. Arndt Bode
	2. Univ.-Prof. Dr. Eike Jessen, em.

Die Dissertation wurde am 16. September 2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 27. November 2002 angenommen.

CRYPTONITE – A Programmable Crypto Processor Architecture For High-Bandwidth Applications

Rainer Buchty

Abstract

Cryptography was and still is one of the most interesting fields in Computer Science-related research. Where its origin lies in military and governmental use, today cryptography is widely used in everyday life. Cryptography secures communication between smart cards and card readers. It scrambles transmissions between DECT telephones and their base stations, it even entered the living rooms through digital Pay TV channels which use cryptographic methods to make sure that only their subscribers can watch their transmissions – and only what they have payed for. Similarly, each DVD player contains cryptographic techniques which were implemented to prevent unauthorized copying and playback.

Also for network infrastructures, company intranets, or the global internet with its shared resources, cryptography is essential to secure transmitted data against snooping. This is especially vital to so called virtual private networks (VPNs), where a (virtually) private network is spanned using shared network resources meaning that although a shared medium is used, the spanned network behaves like a private network in terms of data security and connectivity. Even the underlying physical network infrastructure is hidden.

One of the biggest problems with modern cryptography is data throughput. Combined video/audio data as on DVDs or broadcasted via Pay TV stations easily needs – depending on the quality – 2 to 10 MBit/s and more which means that networks carrying this data have to be able to provide at least the same bandwidth: A smaller bandwidth would lead to visible and audible artifacts.

Such applications require network devices which are not only capable of transporting the incoming data stream but also encrypt or decrypt them without becoming a bottleneck. For this reason, a number of dedicated hardware

solutions exist which support one or more crypto algorithms. Naturally, such solutions do not allow to change the supported algorithms. Change of algorithm automatically means change of hardware; similarly, supporting a number of algorithms requires to have several of dedicated architectures each supporting one algorithm.

A programmable architecture is a solution to this dilemma. General purpose processors seem to be the ideal candidates for this task, however, these did not supply the needed computation power in the past – and today, where they do, they need too much electric power and produce too much heat. Also, it is not economically sensible to use processors worth several hundred Euro but using only a fraction of their potential.

In this thesis a programmable architecture dedicated to typical needs of cryptographic algorithms is presented. Starting with the analysis of several major cryptographic algorithms key parameters and hardware requirements were identified. Based on these, an architecture was designed which not only suits these requirements but also satisfies economic factors like cost of production and power consumption. A number of algorithms were implemented on an architectural simulator and dedicated parts of the architecture were realized using VHDL to measure hardware parameters such as logic use and routability. These numbers were compared against existing solutions; the comparison has proven that a programmable architecture like the one proposed within this work can achieve performance similar to or even better than existing dedicated hardware solutions while still retaining modest hardware requirements.

Acknowledgements

This thesis would not have been possible without the support of many people. First of all, I would like to thank my advisor Professor Dr. Arndt Bode for the great freedom he gave me to conduct my research and the excellent research environment at LRR-TUM. I also want to express my gratitude to my co-referee Professor Dr. Eike Jessen for his valuable comments and hints.

The head of the Architecture Group, Dr. Wolfgang Karl, deserves credit for supporting my work. I want to especially thank him for making it possible for me to attend certain conferences which opened up more than fruitful opportunities.

Here, I would especially express my gratitude to Dr. Nevin Heintze (Agere Systems) for arranging an internship, and his colleague Dr. Dino Oliva (Agere Systems). Both of them deserve credits for their invaluable input and ongoing support. I further want to thank Ed Walters (Agere Systems) for taking the time to review my thesis so that it now will look not too awkward to native English speakers.

The internship would not have been possible without Dr. Tor Jeremiasson (Texas Instruments), who made contact between between Nevin Heintze and me on MICRO'33. Lastly, I want to thank Keith Bauer for providing a comfortable home during my stay in New Jersey.

Furthermore, I thank all of my former and current colleagues at LRR-TUM, in particular Georg Acher and Deti Fliegl, for valuable technical and non-technical discussions and the great time we had and still have. Deti deserves further credits for administering our firewalled Intranet network infrastructure and providing a reliable and secure working environment. Besides, I want to thank my former project mates of the EPP and HotSwap projects, Jürgen Jeitner, Dr. Carsten Trinitis and Max Walter for yielding dis-

cussions. I would also like to thank all my other colleagues for the friendly and familiar athmosphere at LRR-TUM which makes working there such a great pleasure.

Finally, I want to express my gratitude to my parents for their priceless support and sponsorship which made it possible for me to fully concentrate on my studies and research. I want to thank them especially for *not* allowing me to buy that video game back in the early 80s but rather encouraging and sponsoring my interest in computers, computer science, and electrical engineering. My brother Martin I want to thank for sharing the interest in "ancient" electronics.

Last not least, I want to thank my love Nina for her great understanding and support.

Rainer Buchty
September 2002

Contents

Figures

Tables

1

Introduction and Motivation

1.1 What is Cryptology?

When talking about cryptology in general the terms *cryptology* and *cryptography* are often used very loosely. To avoid confusion this document follows the definitions given by the American National Standard for Telecommunication (ANST) as part of the Alliance for Telecommuncations Industry Solutions (ATIS) [6] which are cited below.

> **Cryptology** is both *"the science that deals with hidden, disguised or encrypted communications"* and *"the field encompassing* cryptography *and* cryptanalysis*"* [123].
>
> The term **cryptography** describes *"the art or science concerning the principles, means and methods for rendering plain information unintelligible, and for restoring encrypted information to intelligible form"* [122]. It also refers to this term as *"the branch of cryptology that treats the principles, means, and methods of designing and using cryptosystems"* [122].

Measuring the quality of a cryptographic method is done by means of **cryptanalysis**, namely *"operations performed in converting encrypted messages to plain text without initial knowledge of the crypto algorithm and/or*

key employed in the encryption" [121]. This definition can be condensed to *"study of encrypted texts"* [121] [1].

1.2 Use of Cryptology

Today, cryptology is widely used and has already pervaded everyday life in many ways. So called smart cards use encryption for securing the communication between its host – for example a public telephone – and the smart card processor. Digital pay TV stations scramble their signals in a way which allows only the subscribers to descramble the transmissions. ID cards attached to a PC are used for authentication and digital subscription.

As these examples show, the role of cryptography is manifold; cryptographic algorithms perform the following tasks which directly relate to the above examples:

- **Securing a point-to-point connection** making it impossible to snoop into the communication to gain information about both, the transmitted data and the protocol used for data transmission. Two flavours of reversible data encryption exist which are symmetric and asymmetric encryption. In the first case, the same key is used for encryption and decryption; in the second case different ones are used. This will be explained in more detail when discussing asymmetric algorithms in Section 2.1.

- **Securing a one-to-many broadcast** ensuring that a group of individual recipients (and only this group) is able to decipher the transmitted data. This requires special encryption techniques which allow the use of group key(s) for encryption where the decryption is done using individual (subscriber, for instance) keys.

- **Authentication of data**; this can mean either proving the integrity of the transmitted data by providing some kind of checksum or authenticating the origin of a message. Data integrity is ensured by performing a one-way hash function over the received data and comparing the

[1] Since this work focuses on the development of an efficient programmable architecture for crypto algorithms this work will not deal with cryptanalysis issues beyond introductory information to each algorithm.

computed value against a reference value provided by the sender. A commonly known use of this technique is storage and verification of passwords under the various Unix and Unix-like operating systems which only store the hash value of the password instead of the password itself.

Sender authentication comes virtually for free when using asymmetric Public Key encryption techniques which will be introduced in Section 2.1.

1.2.1 Secure Data Transfer

This is the basic application of cryptography: Information has to be encrypted in a way that only the recipient (or a group of recipients as for the Pay TV example) can access the original, unencrypted data.

As mentioned in Section 1.2, two basic encryption methods exist: symmetric encryption and asymmetric encryption. In Section 2.1 these methods will be discussed more detailed showing how this class of algorithms can be used for digital signing of data.

Also, the way data is transmitted needs to be taken into account. Algorithms can either work on data chunks of same size (*block ciphers*) or continuous data streams (*stream ciphers*). This will be discussed in Chapter 2 which will also include an introduction into chained ciphering where preceding data is included into the encryption process of following data.

1.2.2 Authentication of Documents

Securing data transfer does only guarantee (to a certain degree) that no unauthorized person is able to read the encrypted message. However, it does not prove authenticity. Besides securing against so-called snooping, the recipient has also to be sure that the transmitted data is authentic and does indeed originate from the denoted sender.

One example might be online banking: The bank has to ensure that outgoing money transfers are initiated by the account holder only. Just using login and password for authentification would be highly insufficient; thus, every transaction is not only secured by the **P**ersonal **I**dentification **N**umber (PIN) but also by a per-transaction individual **T**rans**a**ction **N**umber (TAN).

Used TANs are stored in a blacklist ensuring that each TAN can be used only one time.

Another example comes from civil service: Recent approaches target an entire ”electronic service“ allowing citizens to fill out necessary forms for services like passport renewal, announcing movements, or car licensing on-line. Security for this systems needs to be very tight to prevent possible identity frauds. Both, the abuse of another person's account, and pretending a different identity must be prohibited. Furthermore, all transfers must be encrypted in a way that only the denoted recipient is able to decrypt the transmissions to avoid disclosure of sensitive or personal data.

The already mentioned asymmetric encryption also known as public key encryption offers a solution to this problem.

1.2.3 Data Distribution

In certain cases it wanted to address a group of recipients rather than allow only one single recipient to decrypt an encrypted message. The most common use of this technique is Pay TV where all subscribers of a channel have to be able to decrypt the scrambled TV transmission. It is also applicable to other means of *Digital Rights Management* (DRM) where access to information is restricted to a set of subscribers, possibly with varying access rights.

Since this is mainly a mathematical problem dealing with key generation algorithms further discussion of this field are omitted from this work. An introduction into the field of DRM can be found in [64] and [77].

1.2.4 Virtual Private Networks

Instead of expensively creating their own physical network many companies use existing network infrastructures. These can be either private networks shared among several companies but also public ones like the Internet. In either case sensitive corporate data needs to be secured from external observation - despite the shared medium the transported data still needs to be kept private, thus the term *Virtual Private Network* (VPN).

Several software solutions exist which create a secured so-called tunnel between two endpoints spawning a virtual network structure over an existing

network. One of the most common software packages used is the IPSEC system as incorporated in the FreeS/WAN project [124].

It is not uncommon that two VPN nodes connecting entire internal networks (hereafter referred to as intranets as opposed to the Internet) need to process an enormous workload of several 100 MBit/s. The connecting nodes have to cope with these data streams and process them immediately so they do not become a bottleneck and slow down the network interconnection. This would eventually disrupt ongoing real-time data streams like video conferencing, for example.

1.2.5 Other Cryptological Applications

Plenty of additional applications for cryptological algorithms exist such as electronic voting: Here it must be ensured that every person is able to vote only one time, but on the other hand the identity of that person has to be kept secret. For a detailed introduction into secret voting the interested reader is referred to pages 149 to 159 of [106]. The same source also lists many other applications such as distributed secret sharing where a message is split up into a number of pieces where each piece can be proven to be authentic but the message can only be decrypted when all pieces are put together. This encryption method is ideal for companies selling a unique product such as the Coca Cola company which obviously have to ensure that never a single person knows the receipe for their soft drinks.

Other examples of cryptological applications are immediate exchange of secrets, immediate signing of contracts, exchanging data through a trusted third party like a notary, or approving data by a third person. All of these are interesting mathematical questions concerning algorithm construction and key generation; the interested reader might find an absorbing introduction into the various applications of cryptology in [106]. For this work, however, this short introduction should be sufficient.

1.3 Key Generation

Keys needed for asymmetric cryptographical algorithms (see Section 2.1) are based on huge prime numbers. To find such numbers fast is still one of the greatest challenges in cryptology. An introduction into this field would

surely go beyond the focus of this work; the interested reader will find an overview over the various methods of prime number generation and proving primality in [29] and [38].

For about 15 years polynomial methods based on elliptic curve equations [12, 129] have been proven to be among the most efficient methods; for that reason a number of elliptic curves are recommended by the National Institute of Standards and Technology (NIST) [78] and numerous work was done on the field of implementing so-called elliptic curve processors, both in hardware [98, 71, 75, 76, 87, 130, 117] and software [14, 49, 111, 88]. However, the problem of generating large prime numbers is quite different from the cryptographic algorithms used for en- or decryption, which is why key generation will not be included in this work.

Whenever the term *key generation* is used it refers to round key generation[2]. The term *round* denotes one iteration of the algorithm's inner loop. For example the DES algorithm consists of 16 rounds as explained in Section 3.3. Unlike the cipher key, which has been computed by aforementioned methods, the round key is generated using independent rules as defined within the chosen cryptographic algorithm. Roughly said, it is a special permutation based on the cipher key and an integral part of its referring cryptographic algorithm. The methods of key generation will be discussed along with the respective cipher algorithms in Sections 3.3 and 3.4.

1.4 Need for speedy and secure cryptography

The use of cryptography is an emerging market. The growing demand for cryptography arises from the desire to secure networks and data against potential intruders and also the marketing of intellectual property.

In the past the data media itself was adequate copy protection: it is impossible (or better: economically not sensible) to create an exact copy of a book or a magazine, the same goes for audio vinyl records or movies on VHS tape. Whenever one tries to copy these media by means available at reasonable prices the copy is of lower quality than the original. The digital age has changed this dramatically: Data stored on digital media can be copied without loss of quality and at nearly no cost.

[2] Another term for round key generation often found in literature is *key setup*.

With increasing network performance and growing storage capacities the type of transmitted data changed. Whereas text data such as HTML pages usually does not require much bandwidth or, in case of text documents, does not have any real-time demands, requirements for so-called multimedia data like streaming video and audio are quite different. Audio streams in CD quality usually have a bit rate of 128 kBit/s to 256 kBit/s which is already twice to four times the data rate provided by a single ISDN data channel. Even more resource-greedy is video transmission which can easily use 10 MBit/s – an amount which was quite sufficient for entire companies' internal networks (intranets) just a decade ago.

This shows that especially for securing multimedia streams (usually on a per-user base as being used for pay-per-view services) a huge amount of data needs to be encrypted with minimum latency and appropriate throughput to meet the strict real-time requirements such streams have.

But it is not only multimedia which has high demands for bandwidth and data throughput. As mentioned before, companies often connect their local intranets over the internet using secure tunnels. These VPNs have to transport the entire communication between physically separated local nets. Companies with great network bandwidth demands use 155 MBit/s to 622 MBit/s lines to realize the intranet linking already requiring dedicated network routers. En- or decrypting these enormous data streams in real-time requires further hardware since this is far beyond the capabilities of any PC-based software solution.

Such solutions already exist. Back in the 1980s dedicated DES processors were built; today the first chips targeted towards the Advanced Encryption Standard (AES) have hit the market. However, these processors can only perform one single cryptographic algorithm – even worse with AES where common hardware solutions usually support only one out of 9 possible configurations as shown in Chapter 7.

Interestingly, until now almost no universally programmable crypto processors are known to the public. The available solutions, mostly for smartcard based cryptography, are designed for low-bandwith applications like exchange of keys or challenge-response operations. As of today, only one freely programmable and algorithm-independent crypto processor has shown up in literature which is capable of processing at least medium bandwidth data [131]. In addition, a microprogrammable solution [67] exists

which – although it was tailored towards the IDEA algorithm – is not necessarily restricted to that algorithm.

1.5 Research Contribution

The research contribution of this work is the development of a novel crypto processor architecture targeting high-bandwidth applications. It is based on exhaustive algorithm analysis with respect to parallelism, memory, and register use as well as support of non-standard functions. Currently existing solutions are mainly hardwired logic which means that these chips perform only one or a very limited set of algorithms. In contrast, the proposed CRYPTONITE architecture is fully programmable and is not tailored towards a single algorithm: The flexibility of the CRYPTONITE architecture is not achieved by a collection of dedicated processing units where each unit is responsible for a single algorithm; instead, the computation units are designed to be as general as possible[3].

To determine the proper base architecture a set of crypto algorithms was selected and analyzed to determine the architectural needs of each algorithm. This analysis started with a straight-forward implementation of the selected algorithms to firstly determine exploitable parallelism by analysis of data and control flow. A more fine-grained analysis was then performed to determine the number and type of functional units as well as supporting instructions. The data was collected and processed to determine an architecture which would be suitable for all investigated algorithms. Based on current and expected usage some algorithms were weighted over others (AES/DES).

The aim of this work was to create a programmable architecture suitable for a broad range of cryptographic algorithms used for encryption, decryption and fingerprinting (authentication) of data with respect to processing speed and manufacturing costs. To achieve this the algorithms were implemented in many ways to determine the optimal compromise between speed and complexity of the architecture as presented through Chapters 3 to 5; Section 5.2 lists reasons why a dedicated crypto architecture was preferred over using a high-performance general purpose processor.

[3] In case of DES, however, it was vital to provide a specialized DES unit to achieve high throughput. See Section 5.8.4 for further discussion of this topic.

Performance numbers as listed in Chapter 6 have been calculated based on software simulation and VHDL model synthesis. For software simulation a cycle-exact simulator was written to allow implementation and testing of selected algorithms. To determine hardware requirements selected parts of the CRYPTONITE architecture were modelled using VHDL; these models were then synthesized and fitted into the Xilinx Virtex-IIpro FPGA family.

Depending on the algorithm, the CRYPTONITE architecture presented within this work shows a raw crypto performance of 250 to 780 MBit/s *including* round key calculation. Is is remarked that usual software and also some hardware implementations do not include round key generation embedded into the ongoing encryption or decryption process but rather operate on precomputed round keys. The achieved throughput is not only in the range of comparable dedicated hardware solutions but even outperforms a number of these as shown in Chapter 7. In addition, the proposed architecture shows also superior performance within the sparse field of truly programmable solutions. It shall be remarked again that the algorithm implementations as realized on the CRYPTONITE architecture include round key generation.

This very promising performance is resulting from the general architectural concept which was tailored towards the demands of typical cryptographic algorithms and a special memory access technique, the S-Box lookup as explained in Section 5.8.1, and specialized arithmetic operation supporting certain algorithms as listed in Section 5.7.3.1.

1.6 Thesis Organization

The thesis is organized as follows: Chapter 2 will give an overview over the field of cryptology and basic definitions such as operation modes of crypto algorithms. After this introductionary part, the following Chapter 3 will present selected algorithms used for architectural demand analysis together with their relevance within the field of crypto systems.

Based on this analysis, the architectural requirements are summed up in Chapter 4 and the resulting architecture is presented in Chapter 5. Implementational aspects of both, the software and hardware side, are shown in Chapter 6 which also gives performance estimations based on software sim-

ulation and VHDL model synthesis. This topic is completed with Chapter 7 which compares the proposed architecture against existing solutions.

The thesis is closed with Chapter 8 giving conclusion and summary followed by Chapter 9 offering a brief outlook on future work and closes with some final remarks. It is finished with an overview of the instruction set in Appendix A and a description of the instruction word format described in Appendix B. Finally, the Assembly Language Format for CRYPTONITE is presented in Appendix C together with sample code.

2

Crypto Algorithms – Terms and Definitions

This chapter will elaborate on certain crypto algorithms: Those used for reversable encryption, which hereafter will be called cipher algorithms, and those for the generation of fingerprints or *hashes*, the hash algorithms.

Cipher algorithms are bijective mathematical constructs: They allow encryption of a plain text message into cipher text but also the reverse operation, decryption of a cipher text into the original plain text message. The group of hash algorithms, however, is injective. For every input message they create a distinct hash code of a typical size (512-bit for MD5) which then, for instance, can be used for authentication of messages.Traditionally, Cipher algorithms can be as simple as the *Caesar chiffre* which is just a permutation of the alphabet by a certain permutation factor. Instead, modern crypto systems make use of so-called cipher keys. These keys are derived from a special cipher key space which has to adhere to certain rules.

For systems which use the same keys for encryption and decryption equations 2.1 and 2.2 apply. Here, E_K and D_K represent the respecting encryption and decryption functions based on a key K. M and C denote original message and cipher text. The functions E_K and D_K are constructed to be the reverse functions of each other so that equation 2.3 is valid.

$$C = E_K(M) \tag{2.1}$$

$$M = D_K(C) \tag{2.2}$$

$$M = D_K(C_K(M)) \tag{2.3}$$

Algorithms following equations 2.1 to 2.3 use the same key for encryption and decryption. Apart from these, a second class of algorithms exists. These algorithms use different keys K_e and K_d for encryption and decryption instead of a uniform cipher key K. For these algorithms equations 2.4 to 2.6 apply.

$$C = E_{K_e}(M) \tag{2.4}$$
$$M = D_{K_d}(C) \tag{2.5}$$
$$M = D_{K_d}(C_{K_e}(M)) \tag{2.6}$$

2.1 Symmetric and Asymmetric Algorithms

As just discussed, cipher algorithms can be divided into two distinct classes which are symmetric and asymmetric (or public key) algorithms.

Symmetric algorithms use the same key for encryption and decryption of a message as discussed in the previous section. Consequently, this common key has to be kept secret between the communication partners under all circumstances. The class of symmetric algorithms divides further into *stream ciphers* and *block ciphers*. As the name suggests, stream ciphers operate on a stream of data; depending on the context such a stream can be either be a stream of bits or bytes. Block ciphers, in contrast, operate on data chunks of certain size which means that a message distributes over a number of blocks. The size of these blocks are determined by the selected algorithm and algorithm configuration. In case of DES, which is the first standardized crypto algorithm, block size is 64-bit. Using certain operation modes as discussed in Section 2.2, block ciphers can also be applied to streaming data.

Asymmetric algorithms – more figuratively called *public key algorithms* – use different keys for encryption and decryption. One key is called *public key* and can be handed out to all communication partners without compromising security. The second key, however, must be kept private and is therefore called *private key* .

Public key algorithms are designed in a way that a message encrypted with someone's public key can only be decrypted with that person's private key – and vice versa. This makes it possible to encrypt a message in

a way that it can be only deciphered by the addressee. Together with message authentication based on hash algorithms – which provide information about whether a data transmission was modified during transportation or not – public key algorithms ensure the authenticity of a message where the addressee can be sure that the message originates from the denoted sender (provided by the public key algorithm) and that the message content has not been altered (provided by the hash algorithm).

2.2 Modes of Operation

When discussing cipher algorithms in the context of developing a suitable crypto architecture, both, the algorithm itself and its mode of operation must be taken into account. These operation modes influence the way an algorithm can be implemented and especially affect efficiency and possible exploitation of parallelism. Besides, certain modes of operation allow the use of block-based cipher algorithms for character-based data streams which, for instance, are used for remote terminal applications (remote shells).

Numerous operation modes and combinations of these exist which can be reduced to one of the "primitive" modes which are *electronic codebook mode* (ECB), *cipher block chaining* (CBC), *cipher feedback mode* (CFB), *output feedback mode* (OFB), and *counter mode* (CM). Some of these modes employ shift registers into which computation results are fed back; another group of operation modes uses sequence numbers instead of computation results and are therefore called *counter modes*. For modes being derived from these modes the dedicated reader is referred to [103].

For certain crypto algorithms various operation modes are specified; Sections 2.2.1 to 2.2.5 will shortly introduce these operation modes prior to discussing the examined cipher algorithms in detail. For a more detailed discussion on operating modes please refer to [105].

2.2.1 Electronic Codebook Mode (ECB)

ECB is a straightforward way to use block cipher: a plain text block is directly encoded into a cipher text block. Assuming the use of identical keys plain text blocks of same content will be encrypted into cipher text blocks of same content. This would allow the creation of a codebook or dictionary

showing the correlation between plain and cipher text block. On one hand, this is only a theoretical consideration since with cipher algorithms working on n-bit blocks such a codebook would have 2^n entries, furthermore with a key size of k bit, 2^k of these codebooks would be needed. On the other hand a professional cryptanalyst would be able to attack this mode by creating such codebooks and do a statistical analysis: Recent experiments have shown that it indeed is possible to identify language and even the author by statistically analyzing the codebook of compression algorithms which use codebook methods similar to ECB-based encryption against known language parameters [13, 50].

Since plain text blocks can be processed independently, ECB ensures linear coarse-grained increase in throughput by simply increasing the number of computation units and distributing a message of arbitrary length (which has to be a multiple of the block size n, of course) evenly among all computation units present in the system. This is unlike the feedback modes discussed in the following sections which inherently inhibit this coarse-grained parallelism.

ECB is resistant against bit errors (bit value changes) since only the block containing one or more error bits is affected; there is no error propagation. However, ECB is very sensitive to synchronization errors: Once the synchronization is lost, for example by accidental bit insertion when receiving a encrypted message, the remaining message will be unreadable.

2.2.2 Cipher Block Chaining Mode (CBC)

With cipher block chaining the previously generated cipher text block block is used as another input value for the en- or decryption process. This means, that processing a single data block is not independent like in ECB but depends on *all* previously computed data blocks.

The cipher text block of the previous computation round is XORed to the current round's plain text block. While encrypting plain text blocks this XOR function takes place prior to the encryption function, for decrypting cipher blocks it takes place after the decryption function. The method is depicted in Figure 2.1 and can be formulated mathematically as follows:

Encryption:	$C_i = E_K(P_i \oplus C_{i-1})$
Decryption:	$P_i = C_{i-1} \oplus D_K(C_i)$

C_i and P_i denote the cipher and plain text blocks, E_K and D_K represent the encrypt and decrypt functions.

By feeding back the previously generated cipher text, CBC circumvents the major problem of ECB which transforms plain text blocks of same content into cipher text blocks (result blocks) of same content. Using CBC, $messages$ of same content will be processed into *result messages* of same content. This can be circumvented by initializating CBC using appropriate initialization vectors.

Unlike ECB, CBC mode shows error propagation since in case of a received cipher block containing an error not only is this block affected but also the following one before it recovers; much like ECB, however, it will not recover from synchronization errors [104].

There are certain security issues with CBC mode allowing attacks based on cryptanalysis which can be circumvented by slightly modifying the feedback pattern. Since security discussions are beyond the scope of this document, the interested reader is referred to [104].

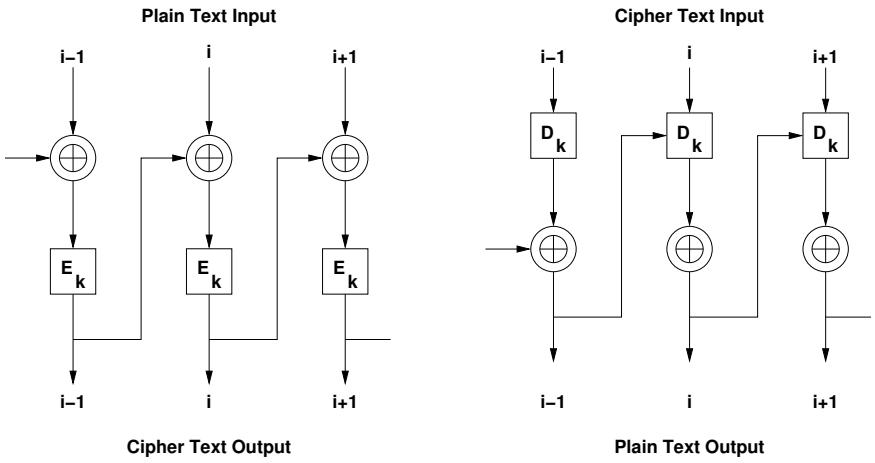

Figure 2.1: Encryption and Decryption using CBC

2.2.3 Cipher Feedback Mode (CFB)

One of the problems with ECB and CBC modes is that they operate on block size rather than single characters or bits as it would be necessary for certain applications such as securing a data stream from or to a terminal: In this example an immediate reaction to single characters is necessary rather than collecting a complete block of characters prior to processing. The straightforward idea of padding a block – even with cryptographically sensible values – is not resonable since a potential attacker would know that the data payload of a complete block is only a single character. Instead of simple padding, a scheme as illustrated in Figure 2.2 is used.

The idea behind this scheme is easy to understand: Since a block algorithm can only work on entire blocks rather than symbols of arbitrary size where simple padding to block boundaries obviously is not reasonable a shift register is used to generate blocks of required length. This register is initialized by a cryptographically sensible value, the so-called initialization vector. Assuming a byte-wise encryption or decryption based on a 64-bit algorithm this shift register contains 8 positions. Whenever a character is sent out, the register content will be (de)scrambled using the (de)cipher key; the leftmost byte of the result of this operation is then XORed with the input byte and the register is shifted by one position to the left.

In case of encryption, the plain text byte is fed into the XOR operation and the resulting cipher byte is stored at the righmost position of the shift register. For decryption, the incoming cipher byte is stored there and is fed into the XOR operation; the result then is the plain text byte. This equals

Encryption:	$C_i = P_i \oplus E_K(C_{i-1})$
Decryption:	$P_i = C_i \oplus D_K(C_{i-1})$

and Operation starts with index $i = 1$ and an initialization vector c_0 is used as a "seed" for the process.

Of course, the feedback mechanism makes CFB less resistant against bit errors and more susceptive to error propagation: A packet containing an error bit will not only affect the current but also the n following rounds where n is the size of the shift register; the maximum boundary for error propagation is $\frac{n}{m} + 1$ with n being block and m being symbol size. For the aforementioned 9 byte example, the current plus eight following rounds will

retain an error. However, it will recover from synchronization errors since the error will age out of the shift register after $\frac{n}{m}$ steps.

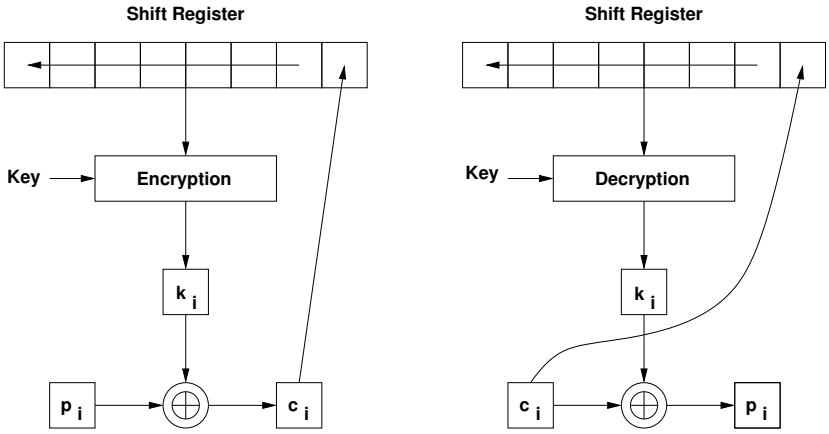

Figure 2.2: Cipher Feedback Mode (CFB)

2.2.4 Output Feedback Mode (OFB)

This mode is very similar to CFB. However, the symbol being fed back into the shift register is the cipher symbol k_i generated from the shift register. Thus, the feedback mechanism and the contents of the shift register S_i are completely independent from the cipher data stream but only depend on the cipher key [30] as illustrated by Figure 2.3. Hence, its formula can be written as:

Encryption	$C_i = P_i \oplus S_i$	$S_i = E_K(S_{i-1})$
Decryption	$P_i = C_i \oplus S_i$	$S_i = E_K(S_{i-1})$

The advantage of OFB over CFB and CBC is that feedback computation can be done in advance and independently from the message if the size of message blocks sharing the same key is known in advance. Like in ECB mode this allows a more coarse grained parallelization since k_i and S_i are independent from the message itself.

Also, for the same reason errors in transmission do not propagate as in the previously discussed feedback modes. However, there are various

security issues concerning the use OFB modes [68, 42, 43, 48] which the dedicated reader is referred to.

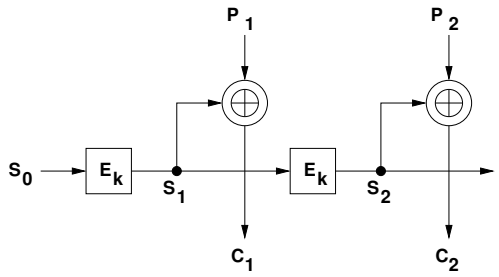

Figure 2.3: Output Feedback Mode (OFB)

2.2.5 Counter Mode (CM)

The security issues of OFB regarding block and feedback size (which have to be of same number for maximal security) as discussed in the previous section can be solved by using the Counter Mode (CM): Block ciphers using CM create their internal state from sequence numbers rather than output of the cipher algorithm itself [44, 51, 69] where the counter is incremented by a constant value after each round. This mode has the same attributes as OFB concerning error propagation and loss of synchronization but allows the generation of output smaller than block size.

The nature of the used counter does not matter; it can be either a simple up-counter, be incremented by a certain offset, or constructed around pseudo-random number generators. Based on this counter, the cipher key, and a rather complicated output function, the cipher symbol K_i is computed [51, 44] which will then be used for encryption or decryption of a message block.

2.3 Summary

Within this chapter a basic introduction into the field of cryptography was presented and definitions of the encryption and decryption process were given. Also, the difference between symmetric and asymmetric algorithms

was worked out. The foci of this chapter are the various operation modes of crypto algorithms.

For security reasons, today mostly chained operating modes are used. For example, IPSEC defines each packet to have its own initialization vector which means that within a packet the blocks are "chained" together. As the name suggests, these modes include results of previous calculations. This already makes one basic decision about the processor architecture: Speeding up algorithm processing has to be achieved by single, powerful processor cores rather than a SIMD-like architecture with a certain number of processor cores working on different data chunks. Doing so would gain speed-up only for ECB mode. For chained modes a SIMD architecture would not necessarily lead to higher per-channel computation power, but provide multiple channels sharing the same crypto algorithm multiplying the overall throughput. Looking at the IPSEC example, such a SIMD architecture would allow multiple channels to be processed in parallel.

Similarly, also a MIMD approach would not provide higher per-channel throughput. Like a SIMD architecture it would offer multiple channels, but – contrary to SIMD – an individual crypto algorithm can be applied to each channel.

3

Algorithm Discussion and Analysis

3.1 Exploitable Parallelism

Network communication exposes many levels of possibly exploitable parallelism that can be used to improve throughput. This can happen on a per-connection base, for example by assigning one processing unit to one single connection. It also can happen on a per-packet base; each packet consists – depending on the selected packet size for transmission and the used crypto algorithm – of several data blocks. With IPSEC in CBC mode, for example, each packet will use different initialization vector which means that packets can be processed individually.

Within one packet, the amount of exploitable parallelism is influenced by the operation mode used and the algorithm itself. Using ECB mode, each block of a packet can also be processed individually; this is not possible with feedback modes where the previously processed block is fed into the current block's encryption (or decryption) process. As mentioned in Section 2.2.1 ECB for security reasons is hardly used anymore which usually means that blocks cannot be processed individually because a data dependency exists between consecutive blocks of a packet.

On the block level, parallelism is dictated by data dependencies within the used algorithm and number of functional units. The following sections concentrate on selected cryptographic algorithms, possibly exploitable parallelism and additional requirements such as dedicated functional units.

3.2 Selection of Algorithms

For this work an initial set of algorithms was chosen consisting of DES/3DES [26], AES/Rijndael [40], and MD5 [93].

DES was chosen since it is the very first standardized crypto algorithm. Despite its age it is still widely used, although software realization of DES is quite problematic since it is very hardware oriented containing many single-bit permutations rather than operations on complete data blocks. Additionally, it operates on a variety of data sizes which are 64-bit, 56-bit, 48-bit, 32-bit and 28-bit which are usually not ideally supported by standard architectures.

AES/Rijndael on the other hand is the new encryption standard and was especially designed to run on current general purpose architectures. Here, the challenge is developing an architecture which supports AES/Rijndael best in all configurations but also does not contain too many AES-specific elements.

The third candidate, MD5, is a widely used hash algorithm which was chosen to demonstrate the flexibility and universability of the architecture described in this work. It is very similar to other hash algorithms like its predecessor MD4 [92] or SHA-1 [27, 108] which are also included within this chapter.

Also, two further algorithms, IDEA [73, 101] and RC6 [96] were added; IDEA was developed in the beginning 1990s as an algorithm easily and efficiently implementable on 16-bit architectures and gained popularity through the PGP tool mainly used for securing email privacy. RC6 is a recent algorithm and was one of the finalists in the AES competition.

3.3 The Data Encryption Standard (DES)

DES is the first standardized crypto algorithm. It was developed by IBM in the 1970s and published in the Official Gazette of the United States Patent and Trademark Office in 1975 [126] and 1976 [127]. In 1977 DES became a Federal Information Processing Standard (FIPS) [26]. It fulfills the later defined *Security Requirements for Cryptographic Modules* [28] which specify security requirements for protecting classified information within computer and telecommunication systems.

The DES algorithm can be used in four different modes [70] which are Electronic Codebook mode (ECB), Cipher Block Chaining mode (CBC), Cipher Feedback mode (CFB), and Output Feedback (OFB). A discussion of these operation modes and their architectural impact was already given in Section 2.2.

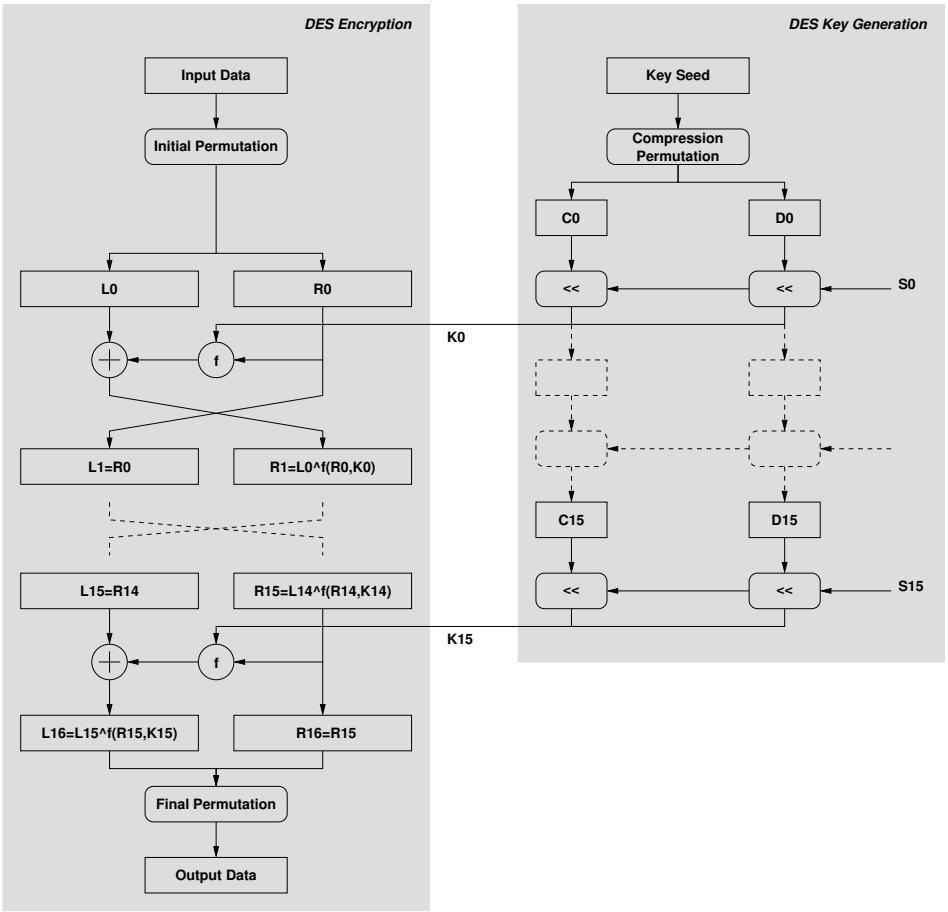

Figure 3.1: Workflow of the DES Algorithm

3.3.1 Design of the DES Algorithm

The DES algorithm basically consists of three parts which are initial permutation, 16 rounds of encryption and final (inverse initial) permutation as shown in Figure 3.1. It operates on 64-bit chunks of data, hence any message which needs to be encrypted using DES has to be padded with appropriate filling data to the next 64-bit boundary. Also the key provided for en- or decryption needs to be 64-bit, however, every eighth bit is dropped during the key permutation resulting in an effective bit length of 56-bit. The remaining 8 bits can be used for parity checks but do not contribute to cryptographic strength.

Within each round, key generation and encryption take place. Key generation is fairly easy: The 56 bits of key data are split into halves; each half is fed into a 28-bit cyclic left-shift register which according to a round constant rotate the key halves up to two positions.

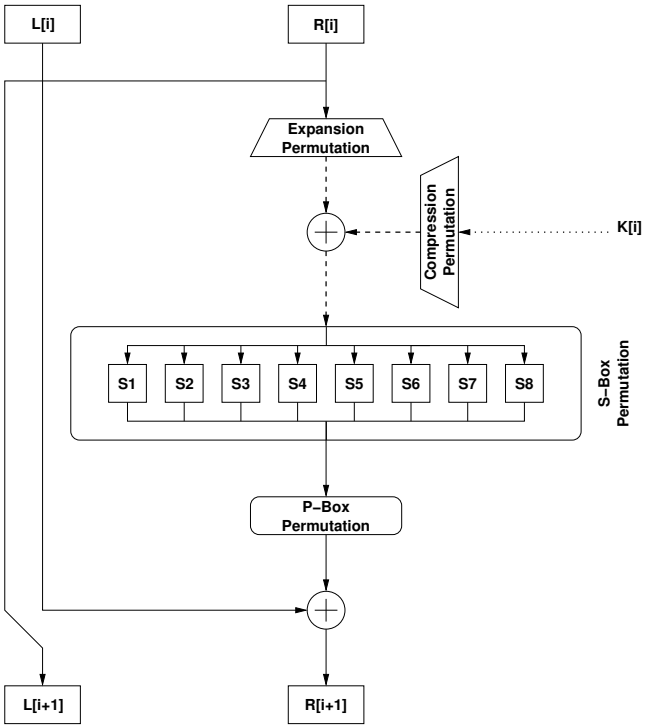

Figure 3.2: A single DES round

The encryption process itself (see Figure 3.2) is slightly more complicated: The input data is split into two chunks of 32 bits each where only the right half is included into the encryption function. Through an expansion permutation these right 32 bits are expanded to 48 bits; similarly, the 56-bit key is compressed to 48 bits. Both values are then XORed, the result of this operation is fed into a lookup table implemented as 8 S-Boxes. These S-Boxes provide the main encryption (or decryption) function within the DES algorithm. All other permutations employed are used to evenly scatter the input data among a 64-bit chunk ensuring that no conclusions about message and key can be drawn by analyzing multiple chunks of encrypted data.

Functionally, each S-Box is a ROM with 32 memory locations: It translates a 5-bit input value into a corresponding unique 4-bit output value. No two S-Boxes are the same. The 48-bit result of the previously performed XOR operation between compressed round key and expanded right half of the input data is fed into the array of S-Boxes producing a 32-bit lookup value. This value undergoes another permutation, the P-Box permutation, before it is XORed with the left half of the input data.

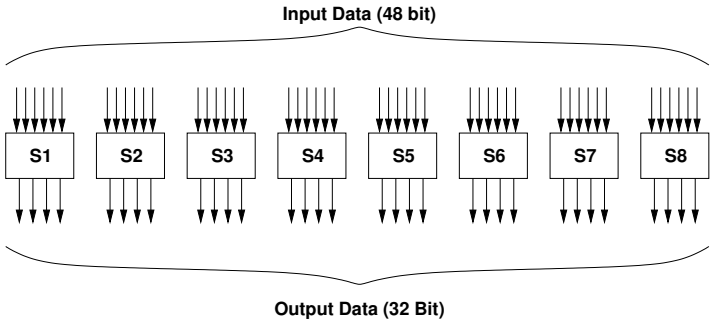

Figure 3.3: The DES S-Box

For the next round, the computed result becomes the left half of the input data where the previous left half is swapped with the right half. This swapping is omitted in round 16, instead the 32-bit values are concatenated and sent through the final permutation which is the reverse operation of the initial permutation.

Since DES is a symmetric algorithm decryption works just the same as encryption, only the round keys have to be applied in reverse order.

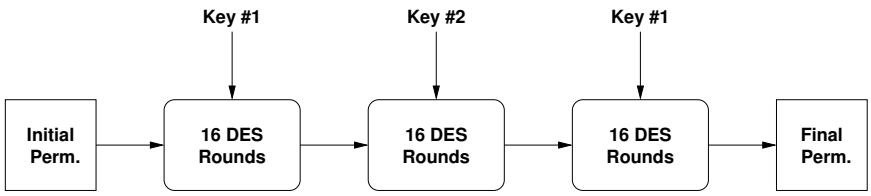

Figure 3.4: Triple-DES

Today, plain DES usually is not used anymore for security reasons since with modern machines it can be broken in comfortable time using brute-force methods (for example, the DES-II challenge was won on July 15, 1998 using the DES Key Search Machine after 56 hours [1]; half a year later, the DES-III challenge was won on January 18, 1999 after less than 24 hours [2]). Instead, Triple-DES – usually referred to as 3DES – is used: This means, that each data packet is DES encrypted three times using two different keys as shown in Figure 3.4.

3.3.2 Analysis of the DES Algorithm

Figure 3.1 shows very clearly that the DES algorithm divides into three steps which are *input permutation* together with *key transformation*, *encryption*, and *final permutation*. The main time consuming factor of DES is the encryption since it is performed 16 times (48 times with 3DES).

3.3.2.1 Speeding up Encryption

Recalling Figure3.2 displays the encryption process consisting of five major steps; assuming these can be computed in 5 cycles this leads to 80 cycles in total for a complete DES encryption, 240 cycles for 3DES which are itemized in Table 3.1.

It is obvious that each permutation is followed by an accompaning XOR operation. This XOR/permutation compound takes two inputs, one of which is permutated prior to the XOR operation, the other is fed in directly. With this observation DES encryption can be reformulated as follows needing only 48 (144 for 3DES) cycles as dissected in Table 3.2.

1.	Expansion-Permutation of R_{i-1}, transfer R_{i-1} to L_i
2.	XORing the result from step 1 with the generated round key
3.	Perform an S-Box lookup based on the XOR result
4.	Do a P-Box Permutation on the lookup value
5.	XOR that result with the value of L_{i-1} and store the result into R_i

Table 3.1: Naive DES Implementation

1.	Expand/XOR R_{i-1} with the actual round key, transfer R_{i-1} to L_i
2.	Perform an S-Box lookup based on the Expand/XOR result
3.	P-Box permutate/XOR the lookup value with L_{i-1} and store the result into R_i

Table 3.2: A faster DES Implementation

1.	Initial expansion/XOR function followed by an S-Box lookup (2 cycles)
2.	14 iterations of the monolithic function followed by an S-Box lookup (2 cycles, 28 cycles in total)
3.	A final P-Box/XOR function (1 cycle)

Table 3.3: An even faster DES implementation (excluding key generation)

1.	One cycle from the initial permutation and key transformation
2.	31 (3DES: 93) from the encryption (key switch within 3DES can be happen in parallel to the P-Box/XOR function)
3.	One cycle from the final permutation

Table 3.4: Ideal DES implementation including Round Key generation

Since 16 consecutive rounds of encryption (or decryption, which is basically the same algorithm) occur, P-Box/XOR and Expansion/XOR happen sequentially for the inner fourteen rounds and can be further condensed to a monolithic three-input function which takes the result of an S-box lookup, L_{i-1} and the round key and produces the index value for the S-box lookup. Within this function also the transfer from R_{i-1} to L_i also takes place.

Using such a function, the 16 rounds of encryption can be further shrunk bringing the complete DES encryption down to 31 (93 for 3DES) cycles as explained in Table 3.3 leading to an ideal (3)DES implementation listed in Table 3.4.

3.3.2.2 Round Key Calculation

Round key calculation consists of two steps which are key shifting and key compression. Since this computation is not dependent on any other input data besides the shift value and the content of the key register this process can happen in parallel to the encryption. Ideally, round key calculation is a combined operation like expansion/XOR or P-Box/XOR and takes only one cycle.

On an ideal parallel architecture calculation of the round key can take place during an S-Box lookup for rounds 2 to 16; the first round's key calculation can happen in parallel to initial permutation needing key transformation prior to rotation. In that case the round key generation can be completely overlapped with the main encryption algorithm without incurring extra cycles.

3.3.3 Proposed Computation Speed

Following the above assumptions based on an ideal architecture, the encryption of a complete 64-bit data block takes 33 (3DES: 95) cycles resulting in 0.52 (1.48) cycles per bit or a throughput of 775 (3DES: 270) MBit/s assuming that the architecture is running at 400 MHz and offers capabilities as listed in Table 3.5.

Parallelism	• Arithmetic operations can happen in parallel with memory lookup (round key generation together with S-Box lookup) • Two arithmetic operations can be processed in parallel (round key generation together with input permutation)
Instruction Set	• DES Input Permutation • DES Final Permutation • combined DES expansion/XOR operation • combined DES P-Box permutation/XOR operation • method to combine the above two operations into one monolithic operation • monolithic round key generation operation consisting of switchable key transformation, key rotation, and key compression
Data Size	• Input/Output: 64 bit • Intermediate Results: 28, 32, 48, and 56 bit

Table 3.5: Architectural requirements for efficient (3)DES implementation

3.4 Advanced Encryption Standard (AES / Rijndael)

When this work was started the Rijndael algorithm [39] was one of the participiants in the contest for becoming the new encryption standard. At the end of 2001 it won against its competitors and became the so-called Advanced Encryption Standard (AES) [40] which over time will replace DES and possibly many other, not officially standardized crypto algorithms, at least in the commercial and governmental area.

Unlike DES which was designed to easily fit into 1970s hardware technology AES/Rijndael was developed to efficiently run in software on modern general purpose processors. Where DES is very easy to implement in hardware it tends to be quite slow in software due to exhaustive bit shuffling and internal changes in data chunk sizes (56-bit to 48-bit, 48-bit to 32-bit, 4-bit to 6-bit, etc.); in AES/Rijndael data chunks are either 8-bit or 32-bit and all internal operations work on these chunk sizes. For block and key size usually multiples of 32 bits are given; in this context, a size of 4 means $4 * 32 = 128$ bits.

A further advantage of the AES/Rijndael algorithm over DES is its flexibility to operate on different key and data block sizes[1]. Encryption strength can be adjusted to the actual needs as shown in Table 3.6.

		Block Size		
		4	6	8
Key Size	4	10	12	14
	6	12	12	14
	8	14	14	14

Table 3.6: Number of Rounds as a function of block and key size

The AES/Rijndael algorithm as realized in the reference implementation can be divided into three phases: Round key generation and initialization, first round and remaining rounds. For decryption, the latter two phases become “first rounds” and “last round”. The phases are illustrated in Figures

[1] It must be noted, however, that usually the data block size is 128 bits. This means, that only three of nine possible configurations are used.

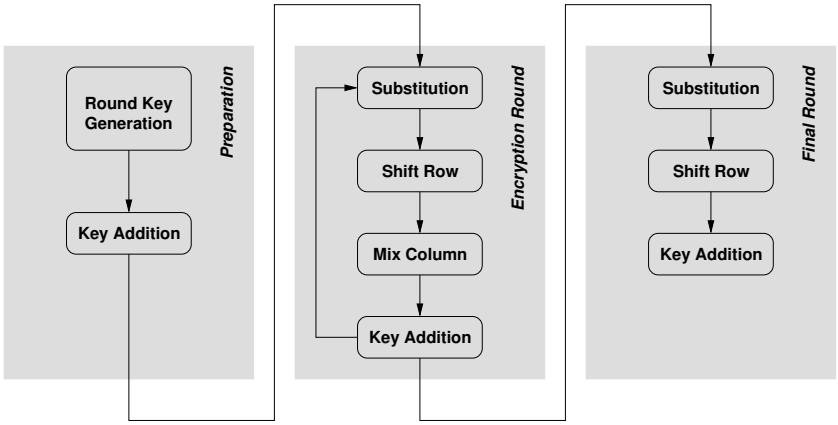

Figure 3.5: AES Encryption

3.5 and 3.6. The rounds themselves split into round key addition, substitution, row shifting, and column mixing or their inverse operation for decryption. Again, the main cryptographic operation is a non-linear transformation based on an S-Box lookup which happens during the substitution stage.

3.4.1 Key Scheduling

AES/Rijndael does not directly operate on the cipher key or a primitive permutation of it. Instead, a sequence of round keys is generated according to the following rules:

- The number of round key bits must equal the block length times the number of rounds plus one. Assuming a block length of 128 bits and a key of same size this results in 10 rounds or $128 \cdot (10 + 1) = 1408$ needed round key bits.
- For this purpose the cipher key is expanded into an expanded key. From this expanded key the round keys are taken in blocks of N_b words where N_b denotes the block size in bits divided by 32.

The expansion is done recursively using rotation and exclusive-or operations where the expansion algorithm is dependent from the key size N_k which is the key size in bits divided by 32. This algorithm is explained

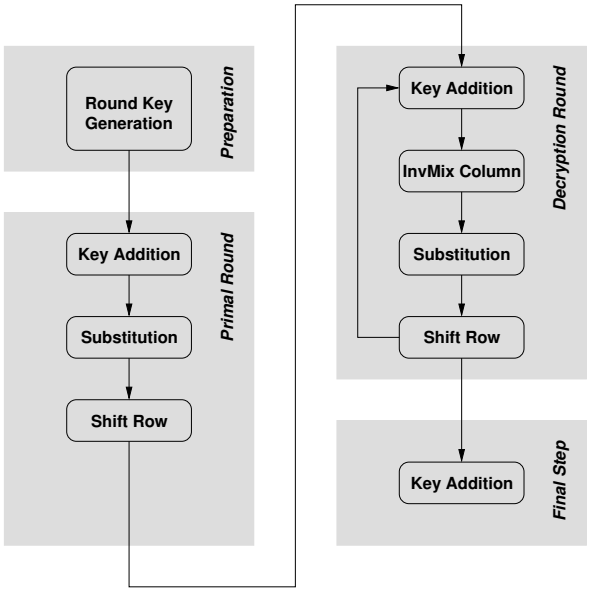

Figure 3.6: AES Decryption

on pages 14 to 15 of [39] and generates the expanded key array $W[i]$ from which round keys of desired block size are taken.

Although the reference implementation – for the sake of simplicity and intuition – precomputes all round keys, the algorithm is designed in a way that round key generation can be done on the fly and interwoven with encryption or decryption. This was done in a fast AES-128/128 implementation [45].

3.4.2 Inner architecture of AES/Rijndael

As explained before, AES/Rijndael encryption or decryption divides into four different steps which are explained below. For the following explanations the nomenclature of the Rijndael description using the term *state* applies [40, 39]:

> *The different transformations operate on the intermediate result, called the* state*. This state can be pictured as a rectangular*

array of bytes with four rows and a variable number of columns equal to the block length divided by 32.

So whenever the term *State* is used in the following explanation the intermediate result of an ongoing encryption or decryption is meant.

- **Key Addition**
  During this operation a round key k is applied to the State a by a bitwise exclusive-or operation. The round keys must have been computed from the cipher key by the key scheduler prior to addition and equals the block length in length. Mathematically expressed, key addition performs the following operation:
  $$\boxed{\forall i, j : b_{i,j} = a_{i,j} \oplus b_{i,j}}$$

- **Substitution**
  This operation replaces each element of the State a by its referring entry in the S-Box table and is performed by a simple table-lookup and write-back algorithm:
  $$\boxed{\forall i, j : a_{i,j} = sbox[a_{i,j}]}$$
  Unlike DES where the entries of the S-Box tables were designed by the NSA giving no further insight into the construction of those tables the creation of the AES/Rijndael S-Box is well documented and can be found in Chapter 4.2.1 of [39].

- **Row Shifting**
  In this stage, rows 1 to 3 of the State are cyclically shifted over different offsets where the shift offsets are individual per row and are dependend on the block length N_b. Row 0 is not shifted. Table 3.7 shows the shift offsets depending on block lengths.

N_b	Row 1	Row 2	Row 3
4	1	2	3
6	1	2	3
8	1	3	4

Table 3.7: Shift offsets for different block lengths

- **Column Mixing**
  In this stage, the columns of the State are considered as polynomials over GF(2^8). They are multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x)$ given by

  $$c(x)=\text{‘03’}x^3+\text{‘01’}x^2+\text{‘01’}x+\text{‘02’}$$

  Because this polynomial is coprime to $x^4 + 1$ and therefore invertible, it can be written as a matrix multiplication following the formula

  $$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

3.4.3 Implementational aspects of AES/Rijndael

For the other algorithms discussed in this chapter due to their simplicity there is hardly a difference between optimized and reference implementation especially when considering hardware implementation. AES/Rijndael is different since it has been designed to run especially well in software. The algorithm itself is well described both in the NIST standard paper and the reference implementation, for evaluating its architectural demands it must be assumed that there is a noticeable difference between an evaluation based on the reference implementation and an optimized version. For this reason, the reference implementation of this algorithm has been reverse-engineered to get the “naked” (not optimized for software implementation) algorithm.

This work is based on an algorithm analysis and evaluation documented in [45]. The implementation based on this analysis is able to perform AES-128/128 encryption within 80 cycles including round key generation as opposed to 60 cycles excluding round key generation for the straight-forward implementation documented in [46].

3.4.4 Architectural Impact

[46] already demands supporting operations which are further extended by [45]. Unlike DES, the needed operations are not explicitly AES-specific and can be used for other algorithms as well.

AES/Rijndael was created with 32-bit architectures in mind. All operations within the algorithm operate on 32-bit (for computation) or 8-bit (for table lookups) quantities, the latter can be avoided by aligning the 8-bit table contents to 32-bit borders. Unlike with RC6, which will be discussed later in this chapter, this behaviour remains consistant throughout all AES configurations. Data and key size do not influence operand sizes, only computation effort. On the other hand, with RC6 the computation effort stays constant and the operand sizes vary. Unlike (3)DES, intermediate results always have 32 bits.

Since DES favours a 64-bit architecture, [46] and [45] were developed having a hypothetical 64-bit architecture in mind which led to the requirements as listed in Table 3.8.

Parallelism	• Two parallel arithmetic operations per cycle sufficient for encryption
Instruction Set	• S-Box lookup
	• Swap-Operation for swapping 32-bit halves of 64-bit registers
	• Splitted 32-bit shift/rotate allowing two 32-bit halves of one register to be shifted by two individual amounts
	• Fold-Operations (see 5.7.3.1)
	• Bit-“Butterfly”-Operations
Data Size	• 32-bit for arithmetic operations
	• 8-bit for table lookups (see text)

Table 3.8: Architectural requirements for efficient AES/Rijndael implementation

3.5 The MD5 Message Digest Algorithm

The MD5 message digest algorithm [93, 107] generates a 128-bit hash value from an input message. Due to the algorithm's workflow the message size has to be a multiple of 512 bits (minus 64 bits) which is achieved by padding the message with a single one bit followed by null bits until the next "512 bits minus 64"-boundary is reached. The final 64-bit field is then filled with a representation of the original message length. This padding mechanism is illustrated in Figure 3.7. The overall length divided by 512 determines the number of MD5 iterations where each iteration processes one 512-bit block of the padded message.

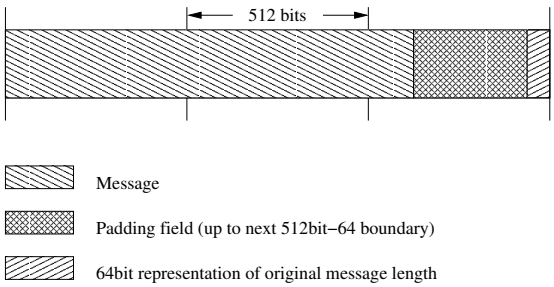

Figure 3.7: Message Padding for MD5

A	$=$	$0x01234567$
B	$=$	$0x89abcdef$
C	$=$	$0xfedcba98$
D	$=$	$0x76543210$

Table 3.9: Chaining Variables Init Values

The hash value is based on four so-called chaining variables , each being 32-bit wide. They are named `A` to `D` and get initialized with the values shown in Table 3.9.

An MD5 iteration as illustrated in Figure 3.8 is based on four rounds of identical architecture. The current 512-bit block is split up into four 128-bit chunks each being fed into one round which consists of 16 steps. Within each round step as shown in Figure 3.9 the values of `B`, `C`, and `D` are fed into

a non-linear function. The result of this function is XORed with a 32-bit excerpt of the j-th 128-bit chunk M_j of the message block being currently processed, and the resulting value is then XORed with a 32-bit value taken from a lookup table t_i[2]. This first intermediate result is then circularly shifted left by s bits where s is one of four values taken from the actual round table. Finally, the computed value is XORed with `B`, the chaining variables are rotated clockwise so that `B` becomes `A`, `C` becomes `B`, `D` becomes `C` and into `A` the result of the aforementioned computation is stored.

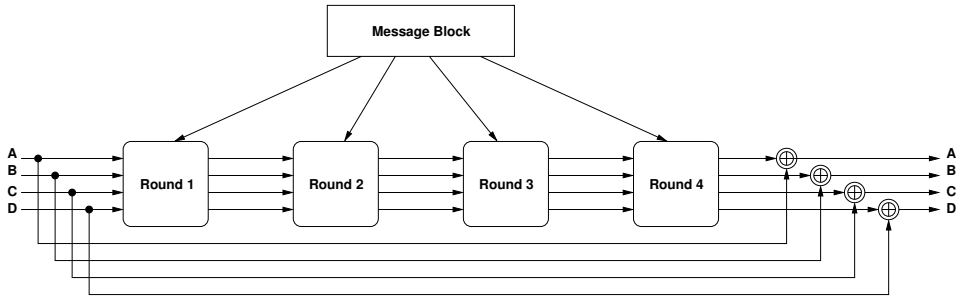

Figure 3.8: Workflow of the MD5 Algorithm

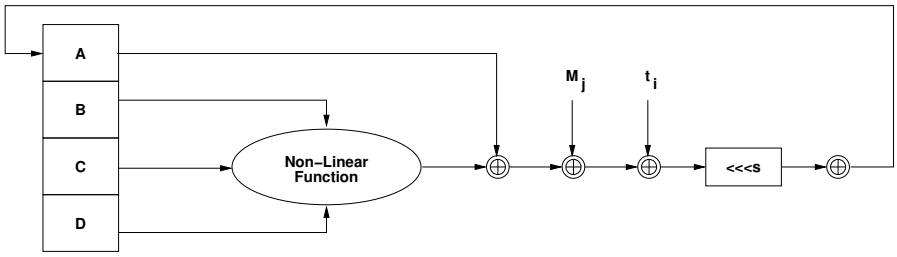

Figure 3.9: A single MD5 round

The hearts of the MD5 hash generation are non-linear functions employed in each round and round step. Each round uses an individual function which is listed in Table 3.10. These functions are designed in a way that the result's single bits are independent from each other and evenly distributed (assuming that this is also valid for the input values X, Y, and Z).

[2]The values of this table are based on the formula $2^{32} * abs(sin(i))$ [107].

Having processed all 512-bit message chunks the resulting MD5 hash is the concatenation of the four chaining variables.

Round 1:	$f(X,Y,Z)$	=	$(X \wedge Y) \vee ((\neg X) \wedge Z)$
Round 2:	$g(X,Y,Z)$	=	$(X \wedge Y) \vee (Y \wedge (\neg Z))$
Round 3:	$h(X,Y,Z)$	=	$X \bigoplus Y \bigoplus Z$
Round 4:	$i(X,Y,Z)$	=	$Y \bigoplus (X \vee (\neg Z))$

Table 3.10: The non-linear functions

3.5.1 Analysis of the MD5 Algorithm

As shown in Figure 3.8 MD5 consists of four different rounds with 16 iterations each. Since there is not much outside the round iterations besides chaining variable initialization and final addition, the main computation time is used within the four rounds which therefore determine computation speed.

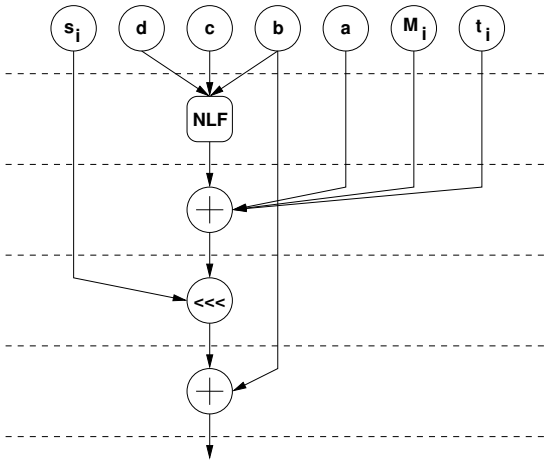

Figure 3.10: MD5 Data Dependencies

3.5.1.1 Round Analysis

A generic round has a structure as depicted in Figure 3.10 and initially consists of two parallel paths which are the computation of the non-linear func-

tion (NLF) plus the addition of a, M_j, and t_i. The results of these operations are then added and shifted, finally b is added to this result. The final sum is stored in the chaining variable a, then the registers are rotated and the next round begins. After the final step instead of the register rotation the initial values of the chaining variables are added onto their computed values.

3.5.1.2 The non-linear functions

Looking at the non-linear functions as sketched in Figure 3.11 shows an almost identical data flow for these: One parameter is negated, combined with another parameter and this result is in term combined with the third parameter.

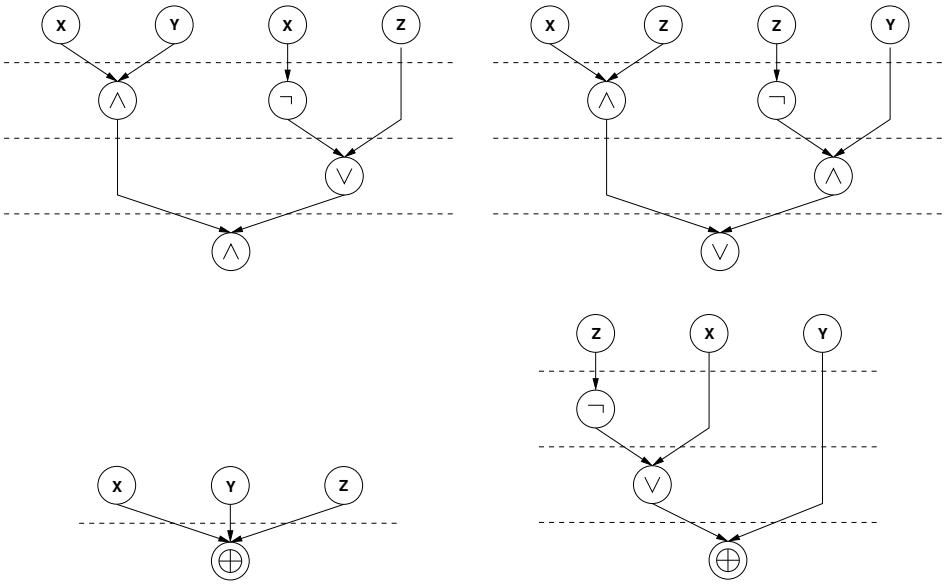

Figure 3.11: The MD5 non-linear functions F, G, H, and I

3.5.2 Similar Hash Algorithms

There are some hash algorithms which are closely related to MD5 and similarly wide-spread in use: These are namely the MD4 Message Digest algorithm [92, 91], and the Secure Hash Algorithm (SHA) [27, 108]. The latter

is superficially similar to MD4 and MD5. However, a closer look reveals that SHA is quite different regarding parallelism which will be discussed in 3.6.

Round 1:	$f(X,Y,Z)$	=	$(X \wedge Y) \vee (\neg X) \vee Z$
Round 2:	$g(X,Y,Z)$	=	$(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
Round 3:	$h(X,Y,Z)$	=	$X \oplus Y \oplus Z$

Table 3.11: MD4 non-linear functions

Since MD4 is the direct predecessor of MD5 it is not surprising that it is a less complex version of MD5. It consists of only three functions and also misses the final addition of the chaining variable b to the result of the shifting operation. Besides, MD4 uses slightly different non-linear functions as listed in Table 3.11. It also uses a different message block indexing for M_j and a different look-up table t_i.

For a discussion of computation speed and architectural impact of MD5 see Section 3.6.1 which will also cover SHA.

3.6 The Secure Hash Algorith (SHA)

SHA [27, 108] was designed for use with the Digital Signature Standard [41] and is also very similar to MD5, but unlike that algorithm it employs five 32-bit chaining variables (producing a 160-bit hash) instead of four (resulting in a 128-bit hash) and 20 iterations per round instead of 16. Also, it applies rotate functions to those chaining variables not employed in the current non-linear function. However, where it is slightly more complex on the computational side as it only uses four round constants, one for each round. MD5, in contrast, uses individual constants for each iteration. SHA uses the same three non-linear functions as MD4 (see Table 3.11 where $g(X,Y,Z)$ is applied to rounds 2 and 4); their data flow is shown in Figure3.13.

However, as Figure 3.12 reveals, there are lesser data dependencies within an SHA round. Where MD5 has 3 sequential steps following the computation of the non-linear function, only one exists with SHA. This means that based on the data dependency graph the non-linear function (NLF) is the main time consuming factor within SHA with 50 to 75 per-

cent, where with MD5 the NLF's contribution to the overall computation is between 25 to 50 percent.

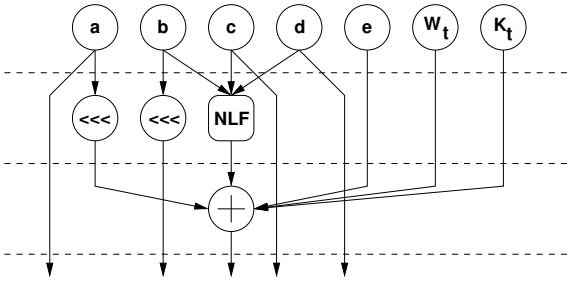

Figure 3.12: SHA Data Dependencies

Round 1:	$f(X,Y,Z)$	=	$(X \wedge Y) \vee (\neg X) \vee Z$
Rounds 2 and 4:	$g(X,Y,Z)$	=	$X \oplus Y \oplus Z$
Round 3:	$g(X,Y,Z)$	=	$(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$

Table 3.12: The SHA non-linear functions

3.6.1 Proposed Computation Speed and Architectural Impact

The time determining factors within SHA are the non-linear functions; on an ideal architecture these vary between 1 and 3 cycles which means that this architecture needs to perform both shifts in parallel to the NLF computation resulting in 3 parallel ALUs. It will also need 3-input logical operations ($\wedge$ and $\oplus$) and a 5-input addition; when limiting to 3-input logical operation a fourth ALU would be required since the 5-input addition needs to be split into two 3-input additions.

Assuming an ideal architecture we get the following numbers for all 4 rounds per 20 iterations: $1 + 20 * (4 + 2 + 3 + 2) + 1 = 222$ cycles per 512-bit chunk equalling 0.43 cycles/bit or 2.31 bits/cycle for an ideal SHA processor as opposed to $1 + 16 * (6 + 6 + 4 + 6) + 1 = 354$ cycles per 512-bit chunk which equals 0.69 cycles/bit or 1.44 bits/cycle for an ideal

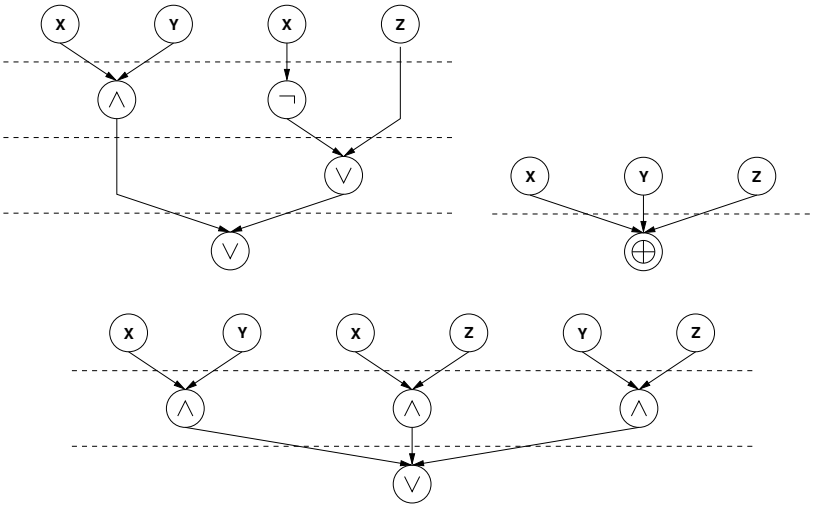

Figure 3.13: The SHA non-linear functions for rounds 1, 2/4, and 3

Parallelism	• up to 5 parallel arithmetic operations
	• up to 4 input variables needed per computation step
Instruction Set	• special monolithic functions not necessary
	• need for indexed addressing to access round tables
Data Size	• Input: 512 bit arranged in 32-bit chunks
	• Output: 128/160 bit (MD5/SHA) arranged in 32-bit chunks
	• Intermediate Results: 32 bit

Table 3.13: Architectural requirements for efficient MD5 and SHA implementation

MD5 processor. Applied to the 400 MHz example this means a throughput of 924 MBit/s for SHA and 576 MBit/s for MD5.

Neither MD5 nor SHA need specific instructions although they would greatly benefit from monolithic non-linear functions or even monolithic round functions, but considering the already high throughput such specialized functions would clearly be overkill. Both algorithms show up a high level of parallelism and could use up to 5 parallel ALUs plus appropriate memory units providing the input values. The algorithms' requirements are summed up in Table 3.13.

3.7 The International Data Encryption Algorithm (IDEA)

The IDEA algorithm appeared in 1992 [73] and was the result of an ongoing improvement of the original Proposed Encryption Standard (PES) which was presented in 1990 and its successor the Improved Proposed Encryption Standard (IPES) [74, 72]. Due to patent reasons it never was as widely used as DES but gained popularity through the well-known and widely used PGP software [139, 138, 102] which is mainly used for email encryption and authentication.

Like DES, the IDEA algorithm is a symmetrical algorithm and operates on 64-bit data chunks. Contrary to DES, key size was doubled to 128 bit. Also round key generation for decryption is more complex. Figure 3.14 gives an overview of the IDEA algorithm.

The 64-bit input data is split up into four chunks of 16 bits which are fed into the first round of calculation. After 8 rounds the data undergoes an output transformation and the resulting four 16-bit chunks are concatenated to form the 64-bit output data.

A single computation round is depicted in Figure 3.15. Unlike DES which is mainly based on static permutation and XOR, IDEA was designed using *"a mixture of operations from different algebraic groups"* [101]. These operations are chosen to be efficiently implementable, both in hardware and software. Since the data chunks used for computation are 16 bits in size, IDEA is efficiently implementable on 16-bit architectures and does not prefer 32-bit architectures like AES/Rijndael or MD5.

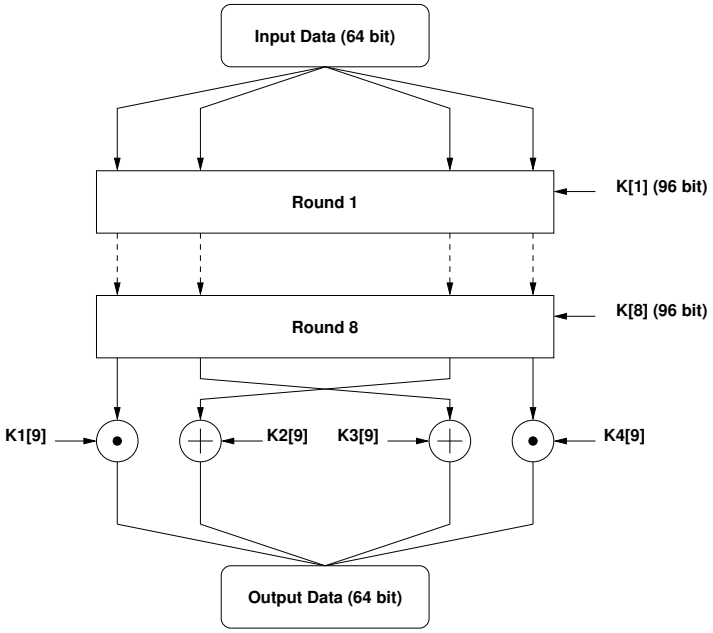

Figure 3.14: Workflow of IDEA

3.7.1 Round Key Generation

Round key generation for IDEA is fairly easy. The 128-bit key is divided into 8 sub-keys of 16 bits each. From these 8 sub-keys, the first six are applied as round keys named $Z_1^{(1)}$ to $Z_1^{(6)}$ to the first round, the last two are used as $Z_2^{(1)}$ and $Z_2^{(2)}$ for the second round. Now all 8 sub-keys of the first iteration are applied and the 128-bit key is rotated left by 25 bits and again divided into 8 sub-keys. The first four of this second iteration will be applied to round 2 as round keys $Z_2^{(3)}$ to $Z_2^{(6)}$, the remaining four go to round 3 as keys $Z_3^{(1)}$ to $Z_3^{(4)}$. Now the 128-bit key is again rotated left by 25 bits and the distribution of sub-keys starts over again. This scheme is applied to all 8 rounds, the very last four sub-keys are used with the output transformation which makes a total of 52 sub-keys (cipher keys) to be used within IDEA.

Decryption is similar, however the keys are applied in reverse order; furthermore the round keys are partly different: For the even numbered round keys the additive inverse $-Z_j^{(i)}$ is used, for odd numbered round keys the

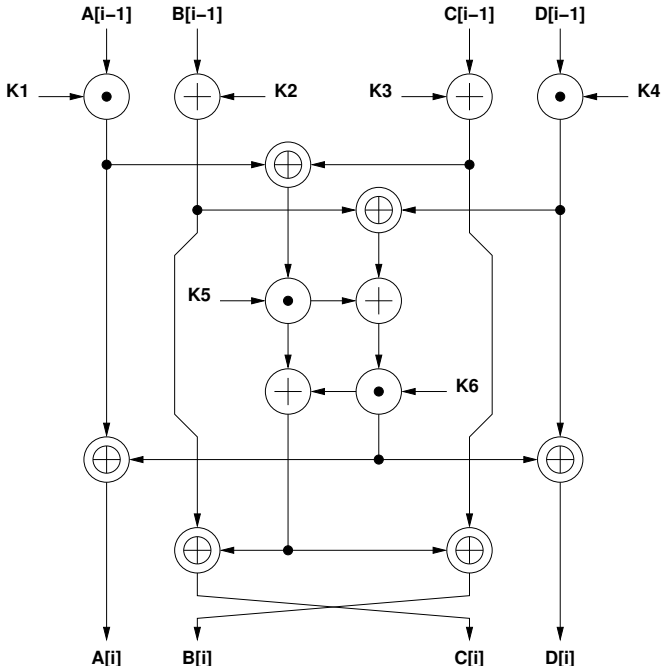

Figure 3.15: A single round of IDEA

multiplicative inverse $Z_j^{(i)-1}$ is applied. Like all computations within IDEA also the inversion is based on modulo $2^{16} + 1$ arithmetics.

Table 3.14 shows the context between round number and round key generation for both, encryption and decryption.

3.7.2 Algorithm Analysis

IDEA was designed to run especially well on 16-bit hardware. Hence it is not surprising that all data chunk sizes used during computation are plain 16-bit – much unlike DES with its many different chunk sizes (28-, 32-, 48-, and 56-bit), AES/Rijndael, or MD5 which were designed having today's 32-bit architectures in mind and operate only on 32-bit (long/int) and 8-bit (byte) quantities.

Much unlike these algorithms, IDEA makes heavy use of multiplications. To ideally support IDEA an architecture would need a 16x16-bit $modulo_{16}$ multiplier. Furthermore, for decipher key (decryption round key)

Round	Encryption Round Key						Decryption Round Key					
1	$Z_1^{(1)}$	$Z_2^{(1)}$	$Z_3^{(1)}$	$Z_4^{(1)}$	$Z_5^{(1)}$	$Z_6^{(1)}$	$Z_1^{(9)-1}$	$-Z_2^{(9)}$	$Z_3^{(9)-1}$	$-Z_4^{(9)}$	$Z_5^{(9)-1}$	$-Z_6^{(9)}$
2	$Z_1^{(2)}$	$Z_2^{(2)}$	$Z_3^{(2)}$	$Z_4^{(2)}$	$Z_5^{(2)}$	$Z_6^{(2)}$	$Z_1^{(8)-1}$	$-Z_2^{(8)}$	$Z_3^{(8)-1}$	$-Z_4^{(8)}$	$Z_5^{(8)-1}$	$-Z_6^{(8)}$
3	$Z_1^{(3)}$	$Z_2^{(3)}$	$Z_3^{(3)}$	$Z_4^{(3)}$	$Z_5^{(3)}$	$Z_6^{(3)}$	$Z_1^{(7)-1}$	$-Z_2^{(7)}$	$Z_3^{(7)-1}$	$-Z_4^{(7)}$	$Z_5^{(7)-1}$	$-Z_6^{(7)}$
4	$Z_1^{(4)}$	$Z_2^{(4)}$	$Z_3^{(4)}$	$Z_4^{(4)}$	$Z_5^{(4)}$	$Z_6^{(4)}$	$Z_1^{(6)-1}$	$-Z_2^{(6)}$	$Z_3^{(6)-1}$	$-Z_4^{(6)}$	$Z_5^{(6)-1}$	$-Z_6^{(6)}$
5	$Z_1^{(5)}$	$Z_2^{(5)}$	$Z_3^{(5)}$	$Z_4^{(5)}$	$Z_5^{(5)}$	$Z_6^{(5)}$	$Z_1^{(5)-1}$	$-Z_2^{(5)}$	$Z_3^{(5)-1}$	$-Z_4^{(5)}$	$Z_5^{(5)-1}$	$-Z_6^{(5)}$
6	$Z_1^{(6)}$	$Z_2^{(6)}$	$Z_3^{(6)}$	$Z_4^{(6)}$	$Z_5^{(6)}$	$Z_6^{(6)}$	$Z_1^{(4)-1}$	$-Z_2^{(4)}$	$Z_3^{(4)-1}$	$-Z_4^{(4)}$	$Z_5^{(4)-1}$	$-Z_6^{(4)}$
7	$Z_1^{(7)}$	$Z_2^{(7)}$	$Z_3^{(7)}$	$Z_4^{(7)}$	$Z_5^{(7)}$	$Z_6^{(7)}$	$Z_1^{(3)-1}$	$-Z_2^{(3)}$	$Z_3^{(3)-1}$	$-Z_4^{(3)}$	$Z_5^{(3)-1}$	$-Z_6^{(3)}$
8	$Z_1^{(8)}$	$Z_2^{(8)}$	$Z_3^{(8)}$	$Z_4^{(8)}$	$Z_5^{(8)}$	$Z_6^{(8)}$	$Z_1^{(2)-1}$	$-Z_2^{(2)}$	$Z_3^{(2)-1}$	$-Z_4^{(2)}$	$Z_5^{(2)-1}$	$-Z_6^{(2)}$
Output Xform	$Z_1^{(9)}$	$Z_2^{(9)}$	$Z_3^{(9)}$	$Z_4^{(9)}$			$Z_1^{(1)-1}$	$-Z_2^{(1)}$	$Z_3^{(1)-1}$	$-Z_4^{(1)}$		

$Z_j^{(i)}$ means the j-th sub-key of Round i

Table 3.14: IDEA sub-keys used for encryption and decryption

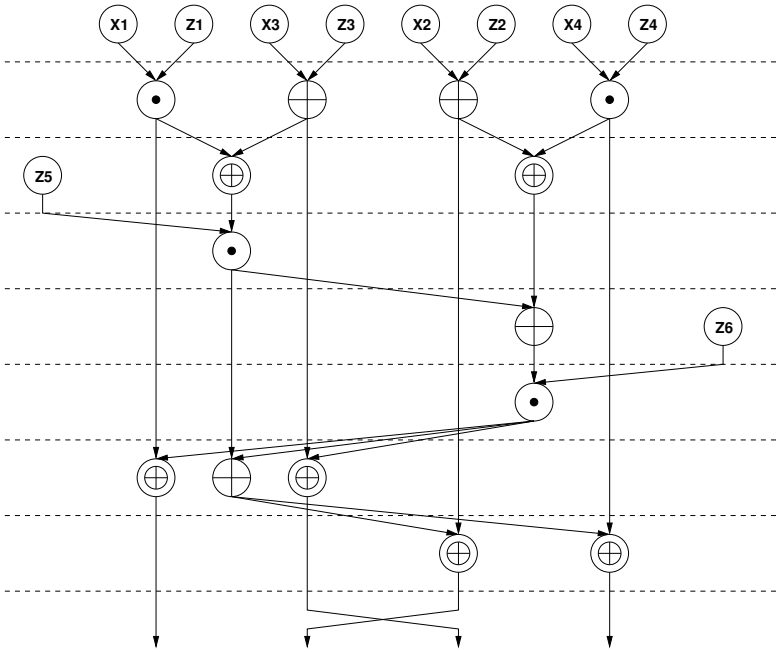

Figure 3.16: Data flow within an IDEA round

calculation division and modulo arithmetics based on modulo $2^{16} - 1$ are needed. Although these operations can be emulated using just addition, subtraction, and Boolean operations, doing so would highly impact computation speed.

Assuming that the needed functions are present, data flow analysis of IDEA shows a quite high register pressure. Not less than 8 input variables are fed into the first calculation step where four of the results have to be kept for further processing in steps 6 and 7. Similarly, the results of step 2 and 3 have to be queued for later use.

Regarding parallelism, IDEA gives a quite unbalanced view as shown in Figure 3.16. For the very first step four algorithmical operations can happen in parallel; this drops down to two in step 2 and reaches strict sequentiality for steps 3 to 5. The following two steps again incorporate 3 and 2 parallel operations, finally a swap operation between two results takes place and the next round begins. For the eighth round the final swapping is omitted,

Parallelism	• Maximum parallelism of 4 instructions per cycle; ideally, either 4 memory operations, 4 algorithmic operations or mixtures of both
	• Round key generation can be embedded into sequential part of round calculation
Instruction Set	• 128-bit rotation for round key generation
	• single-cycle 16x16-bit multiplication
	• single-cycle 16:16-bit division and modulo operation for decipher key generation
Data Size	• Input/Output: 64 bit (Data), 128 bit (Key)
	• Intermediate Results: 16 bit

Table 3.15: Architectural requirements for efficient IDEA implementation

instead the output transformation, four parallel arithmetic operations, take place.

Thus, an ideal IDEA processor could process a complete encryption or decryption within $1 + 8 * 8 = 65$ cycles which equals a processing rate of $\frac{64bits}{65cycles} = 0.99$ bits per cycles or 1.01 cycles per bit which is about twice the numbers as calculated for DES, which also operates on 64-bit data chunks. Assuming that this IDEA processor runs at 400 MHz it will reach a throughput of about 394 MBit/s as opposed to 775 MBit/s for DES.

3.8 The RC6^{TM} Block Cipher

RC6[3] was one of the competitors for becoming the new AES standard and is a successor to the RC5 Block Cipher[94, 95, 109]. It was not only designed to meet the AES committee's requirements but also to be ideally suited for today's 32-bit microprocessors.

[3]RC6 is a registered trademark of RSA Laboraties

Like RC5, RC6 is parametrizable and should correctly be written RC6-$w/r/b$ where w denotes the size of data chunks in bits, r number of rounds and b the length of the encryption key in bytes which can be any number between 0 and 255. For AES submission, RC6 was configured with $w = 32$, $r = 20$ and $b = 16$. With these parameters, key and block size of RC6 correspond to Rijndael-128/128. Regardless of configuration data, RC6 always employs four working registers of w bits size and six basic operations which are listed in Table 3.16.

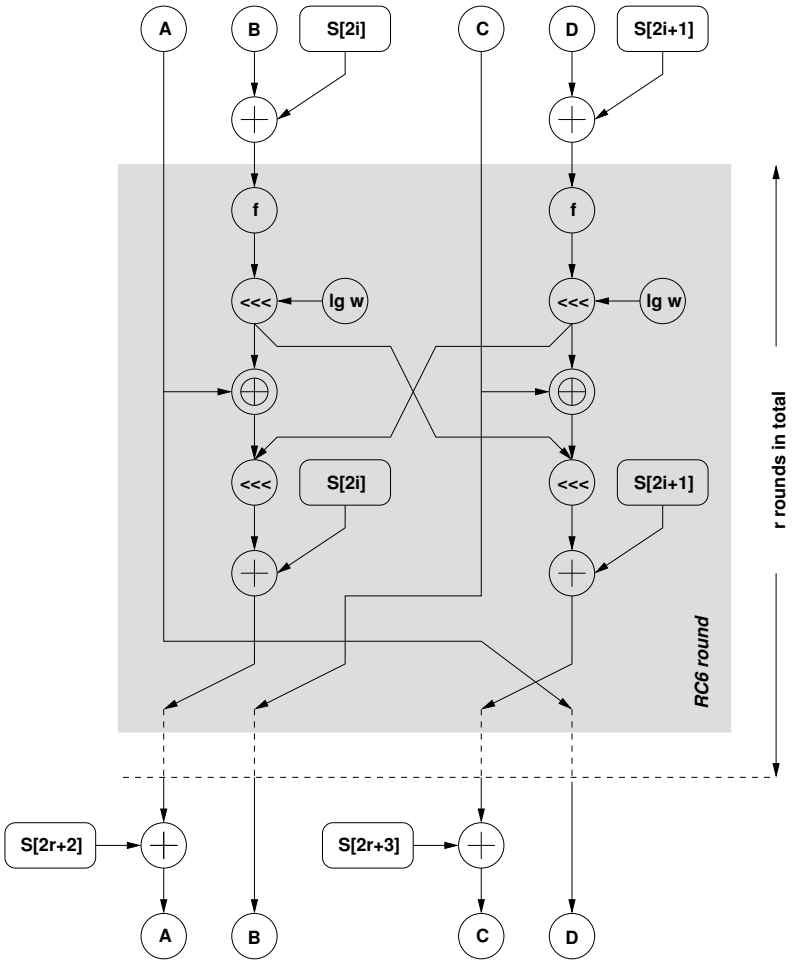

Figure 3.17: Workflow of the RC6^{TM} Block Cipher

$a + b$	integer addition modulo 2^w
$a - b$	integer subtraction modulo 2^w
$a \times b$	integer multiplication modulo 2^w
$a \oplus b$	bitwise exclusive-or of w-bit words
$a \lll b$	rotate the w-bit word a to the left by the amount given by the least significant $\log_2 w$ bits of b
$a \ggg b$	rotate the w-bit word a to the right by the amount given by the least significant $\log_2 w$ bits of b

Table 3.16: RC6 Basic Operations

3.8.1 Key Scheduling

Key scheduling of RC6 is practically identical to key scheduling of RC5: As said before, any key b consists of $0 \leq b \leq 255$ bytes. From this key, $2r + 4$ words of w bits each are generated and stored in the round key array $S[0, \ldots, 2r + 3]$.

The key scheduling consists of two loops, an initialization loop which presets the round key array based on distinct initialization values, and the main computation loop which generates the round key array by severe mixing of the supplied user key [96]. Compared to other algorithms, RC6's key generation is rather heavyweight and makes it sensible to precompute the round key array. This eventually causes performance issues as illustrated in [97]. Implementations on smaller processors or in hardware might suffer from the use of multiplication and modulo operations for round key generation.

3.8.2 Algorithm Analysis

Like all other algorithms discussed so far also RC6 is based on rounds. As depicted in Figure 3.17, RC6 consists of an initial addition of the first two round keys to the input variables B and D followed by r rounds. Finally, the A and C results are added to the $2r + 2^{nd}$ and $2r + 3^{rd}$ round key.

For decryption the algorithm is just processed backwards starting with keys $2r + 2$ and $2r + 3$ being added to A and C followed by r rounds and ending with the addition of key 0 and 1 to B and C. In this manner it

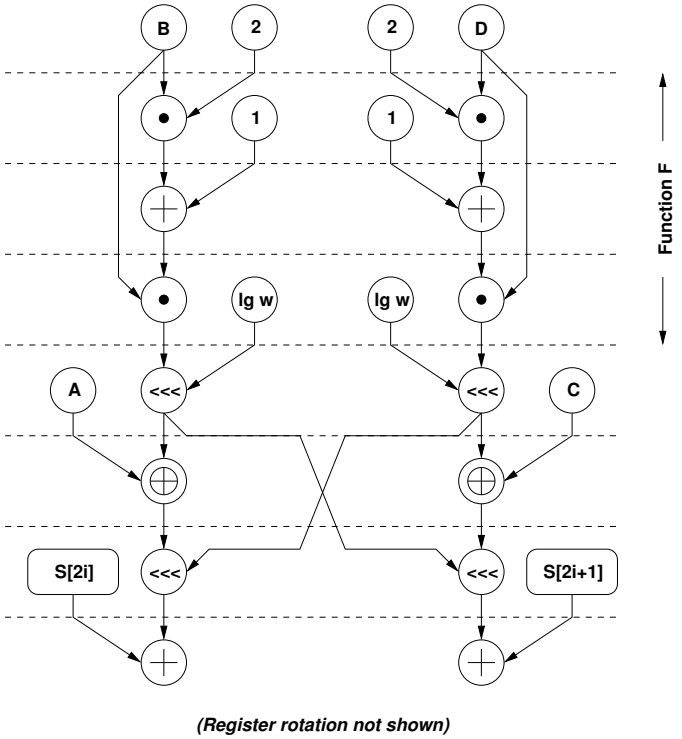

Figure 3.18: Data flow within one RC6^{TM} round

resembles much of DES if you consider these key additions to be input and output transformation.

Basically, each round consists of two identical ▶strands which results are intermixed after the shift operation. There is a strong data dependency within each strand which makes it impossible to combine or precompute intermediate results as shown in Figure 3.18 which means, that there is – again much like DES – not much parallelism to exploit. The round key generation is pretty trivial as shown in 3.8.1 and can be reduced to memory lookups.

This leads to technology requirements as shown in Table 3.18 with the speed estimations listed in Table 3.17: For AES configuration (w=32 bits, r=20 rounds, b=16 byte=128 bits) this should give a performance of $2 + (20 * 7) + 1 = 143$ cycles equalling 0.90 bits per cycle or 1.12 cycles per bit without having the monolithic RC6-specific function. On an estimated

400 MHz clock rate this will lead to a throughput of 358 MBit/s. If the RC6-specific function is present, the numbers are $2 + 20 * 4 + 1 = 83$ cycles equalling 1.54 bits per cycle or 0.65 cycles per bit leading to a throughput of almost 617 MBit/s at 400 MHz.

Initialization	2 cycles
Round Processing	7/4 cycles
Postprocessing	1 cycle

Table 3.17: Speed estimations for RC6 on an ideal architecture

Parallelism	• Arithmetic operation can happen in parallel to memory lookup (round key)
	• RC6 consists of two strands with sequential nature; thus, two arithmetic operations can be processed in parallel
Instruction Set	• Addition and Multiplication modulo 2^w
	• Exclusive-OR on operands of w bits
	• Rotate-Left by $\log_2 w$ positions on w-bit operand
	• Register Swapping
	• Special instruction $f(x) = x*(2x+1) \lll log_w$ would save 3 cycles per round
Data Size	• Input/Output: $w*4$ bit, typically 128 bit (w=32)
	• Intermediate Results: w bit (32, 48, or 64-bit operand size)

Table 3.18: Architectural requirements for efficient RC6^{TM} implementation

3.9 Summary

In this chapter, a selected set of crypto algorithms was presented. These are namely DES and 3DES, AES/Rijndael, IDEA and RC6. Also, a number of hash algorithms used for message authentication were investigated: The widely used MD5 and its somewhat improved version SHA-1 were selected; furthermore, the relationship to their common predecessor, MD4, was illustrated. Also, the influence of selected algorithm and operation mode on possibly exploitable parallelism was explained.

The algorithms were selected for many reasons. DES is not only the first standardized crypto algorithm but is also very hardware-oriented, In contrast, AES/Rijndael, the new standard, was designed with respect to actual 32-bit architectures but can be also realized on less powerful architectures. IDEA stands somewhat in between these two algorithms since it was designed for 16-bit architectures but requires fast modulo-multiplication and -division units. Finally, RC6 was selected, since it was one of the most promising candidates in the AES competition. The hash algorithms were selected to include a different algorithm class into this investigation to work out a most general architecture.

Where this chapter concentrated more on the needs of the single algorithms and how they can be implemented in an ideal manner, their impact on the CRYPTONITE architecture will be elaborated in Chapter 4.

4

Architecture Impact resulting from Algorithm Analysis

In the previous chapter a set of well known and widely used algorithms was discussed. These algorithms were selected to cover a broad range of possible applications and included crypto algorithms designed for hardware implementation (represented by DES and 3DES), and these developed to run efficiently on 16-bit (IDEA) and 32-bit (AES/Rijndael, RC6) general purpose processors. Also hashing algorithms, MD5 and SHA, were included.

Each algorithm has certain architectural needs. Some algorithms would greatly benefit from three-input functions, others require special monolithic functions for high performance. However, in a programmable environment algorithm-specific functions have to be reduced to an absolute minimum since the idea of such an architecture is certainly not building a container around dedicated hardware solutions.

Based on the results of the previous chapter the influences on the architecture will be investigated by determining the average requirements of all explored algorithms. This examination will then provide information about the basic architectural concepts like number and type of functional units, register size and count, and the necessary instruction set.

4.1 Functional Units

Analyzing the algorithms with respect to used operations results in the list printed in Table 4.1. Concerning the XOR functions two observations can

- Boolean operations like logical AND, OR and XOR; also a NOT operation for inverting an operands bit pattern is needed as a result of the hash algorithm analysis
- simple $modulo_w$ algorithmic operations like addition, subtraction
- complex $modulo_w$ algorithmic operations like multiplication, division, and modulo as needed by the IDEA algorithm
- single indexed memory access for loading round constants and storing results
- simple load/store memory access for keeping and reloading intermediate results
- multiple parallel indexed memory accesses as needed for S-Box lookups
- conditional looping of code snippets for repeating computation rounds without the need of loop unrolling which would lead to bloated code size

Table 4.1: Operations employed in analyzed algorithms

be made: First, there are certain circumstances where especially the XOR function could use more than one input, for instance within one of the non-linear functions of MD5 and SHA which ideally make use of a 3-input XOR function. Second, the XOR function often is the final step after a series of computations. Ideally, this final step could happen in parallel to other algorithmic operations to speed up computation. This consideration leads to a somewhat partite ALU consisting of a register file for parameter storage, an XOR unit and an arithmetic unit as described in Section 5.7.

For memory access a very basic architecture was considered: No immediate memory access is possible, but memory access is always register-based which means that a memory address has first to be loaded into an address register; memory access can then happen based on the content of a selected address register. Extending this model by a simple adder logic

enables combining two address registers; this leads to indexed memory addressing. This scheme can be modified to allow parallel S-Box access as described in Section 5.8.1 when discussing the address generation unit. The complete memory unit is presented in Section 5.8.

Since some of the crypto algorithms clearly separate into two strands as shown with DES (encryption and key generation), IDEA or RC6 (two parallel strands interchanging values within encryption) it seems to be somewhat natural to realize two individual sets containing one ALU and memory unit. This reproduces the observed two-strand behaviour in hardware. This basically means just doubling ALUs, associated memory units and data memories which leads to a structure as described in Section 5.3. This structure also allows the architecture to exploit parallelism found within the other algorithms and leads to fast execution times as shown in Chapter 6.

This split architecture, however, creates a problem of exchanging values between the two strands as needed for all discussed algorithms. Here, a bipartite approach has been taken making it possible to forward computated results or register values from one ALU to the other as shown in Section 5.7

4.2 Register File

Registers are needed to hold input and output data as well as intermediate results. For that reason they need to satisfy a wide range of constraints such as data size and number of values to hold.

Not supporting certain data sizes would eventually result in additional cycles needed for modulo arithmetics or masking out unused bits. Supporting too many data sizes would unnecessarily add complexity to the design resulting in slower processing speed and higher power consumption.

Similarly, implementing too few registers will create a cycle penalty for intermediate results to be stored to or fetched from memory; too many registers, however, will increase the processor's die size, cause higher power consumption and potentially slower processing speed.

4.2.1 Register Size

Looking at the algorithms, the common data chunk size is 32-bit which is found within many of the discussed algorithms. Both hash algorithms use it,

also AES/Rijndael and RC6 in AES configuration work on 32-bit entities. IDEA, however, is strictly 16-bit based.

A second common data chunk size is 8-bit which is commonly used for table (S-Box) lookup e.g. within AES/Rijndael and DES, the latter being quite special here since it takes 64-bit quantities but internally operates on a broad variety of bit sizes like 56-bit after input and before output transformation, 28-bit for key generation, 48-bit for S-Box lookup and 32-bit for intermediate round results. However, a closer look reveals, that these sizes can be mapped either to 32 or 64 bits by rearranging the "odd" sizes to 8- or 32-bit boundaries. That way the 28-bit value would refer to an aligned 32-bit value, similarly the 48-bit data size as appearing together with the S-Box lookup can be rearranged to a 64-bit value where the 6-bit chunks are aligned to 8-bit boundaries.

A software implementation of DES following these alignment schemes and hence requiring only data sizes of 32- and 64-bit displayed no negative side-effects concerning registers. For this reason and since 8-bit data size is only employed within an S-Box lookup as mentioned above the decision was made to support only two data sizes: 64-bit (word), and 32-bit (half-word where upper and lower half-word of a 64-bit word are directly accessable). To select these data sizes a 2-bit control value is needed resulting in 4 possible states. The fourth, unoccupied, state is used to clear a register or zero its output ("muting´´ the register), allowing an easy reset of the register contents to zero as described in Section 5.7.2 and illustrated through Table 4.2.

4.2.2 Number of Registers

Looking at the algorithms shows that most of these incorporate just two-input functions; this is certainly true for DES, RC6 and IDEA. Also most operations used for hashing only have two input parameters. Additionally, a straightforward implementation of AES/Rijndael shows that two-input functions are sufficient.

This means, that for a simple calculation step only two source registers are needed; some algorithms like IDEA or RC6 need to keep intermediate results for later computation steps so having an intermediate result register like an accumulator placed inside the ALU would come in quite handy.

Control Value	Operation
00_2	clear or "mute" register
01_2	load 64-bit value into register
10_2	load lower 32-bit half of register
11_2	load upper 32-bit half of register

Table 4.2: Register Size Control

Three registers already need two address lines for access control which means that a fourth register comes in virtually for free in terms of system control. Such a fourth register will be quite useful on algorithms with great register pressure like AES/Rijndael or IDEA and can be used for storing multiple intermediate results or accelerate operations by pre-loading needed values.

Following this argument the number of registers per register file has been set to four as a compromise between register pressure caused by register-hungry algorithms like IDEA or AES/Rijndael and on-chip space requirements caused by the registers itself and their address and data paths.

Releasing this pressure by the possibility of exchanging values between registers of the two strands is a beneficial side effect of the interlink structure already described in Section 4.1. In addition, pressure is further eased by the possibility to use the Memory unit's data output register, which is needed for synchronizing data, as a supplementary source and destination registers for arithmetic operations as described in Section 5.8.

4.3 Special Instructions

A programmable device should not be a collection of specialized hardware. Instead, it should be based on ideally primitive and reusable hardware functions.

For some algorithms, however, the fully programmable approach does not make sense when it comes to computation speed: DES is a good example of an algorithm which has to be clearly based on specialized hardware. Of course, all DES operations can be reduced to a series of primitive operations such as Boolean algebra functions or shift operations, but this leads to

a highly ineffective code. As a result of the DES analysis there clearly has to be a specialized DES unit to support this algorithm at reasonable speed as discussed in Section 5.8.4. AES/Rijndael is another example since both, its basic [46] and fast [45] implementations clearly need supportive operations to enable the desired speed requirements. However, unlike DES these operations as described in Section 5.7.3.1 do not need to be Rijndael-specific but could possibly be used for other applications.

It can be observed, that for efficient implementation especially the two standardized crypto algorithms show great need for specialized functions. In contrast, the other algorithms can easily be realized using Boolean and basic mathematical operations. Due to the continued use of (3)DES in many real-time applications such as (de)scrambling digital pay TV and the expected success of AES, operations supporting these algorithms must be implemented regardless of whether they can be re-used for other algorithms or not.

IDEA and RC6 use modulo-based multiplication which makes them quite different from all other discussed algorithms which only rely on table lookups, simple arithmetics, and boolean operations. Implementing a hardware multiplier would support these algorithms and enhance computation speed greatly. However, the problem arises that IDEA uses mod_{16} multiplication where RC6 favours mod_{32} to mod_{64} multiplication depending on the configuration. The solution to this dilemma with respect to the already taken decision towards a 64-bit architecture would be a configurable multiplier which either works as 64-bit*64-bit mod_{64}, mod_{32}, or mod_{16} multiplier. Integer division and modulo operator, however, should not be included: With the investigated algorithms these operations are only needed together with IDEA round key calculation for decryption. The construction rules of these keys, however, disable embedding the round key generation into the encryption routine. Instead, they should be precomputed by external means and applied together with the input data.

4.4 Summary

Where Chapter 3 focused on the algorithms and an ideal architecture for each algorithm, this chapter concentrated on shaping out an architecture which serves well for a broad variety of algorithms.

For this reason, the needs of all algorithms investigated in Chapter 3 was summed up and discussed with respect to their needs towards number and type of functional units, number and size of internal registers. Also, algorithm-specific instructions have been discussed.

Based on this discussion, an initial architecture partitioning into an arithmetic unit consisting of arithmetic-logical unit (ALU), XOR unit (XU) and associated register file, and a memory unit (MU) being responsible for data transfers from an associated data memory to the registers and vice versa was developed. A decision was made to incorporate two independent strands consisting of these units since many algorithms showed a certain amount of inherent parallelism which can be exploited by this two-strand model.

Furthermore, the register file has been defined as a set of four 64-bit registers with directly addressable 32-bit halves; to release register pressure and to allow exchange of values between the two strands, an interlink mechanism was created which enables forwarding of register values and computation results between the two independent strands.

Finally, a set of more or less algorithm-specific instructions was discussed. As for DES this resulted in a completely independent DES unit since DES operations are too special to be used for any other algorithm; using less specific operations, however, would lead to a inacceptable slowdown of DES performance. For AES/Rijndael a more unspecific set of supporting instructions has been defined. The presence of a mod_w multiplication unit is recommended for IDEA and RC6, but can be omitted fue to the low distribution and use count of these two algorithms.

Chapter 5 will now describe the resulting architecture in detail. Please notice that due to signed contracts the work presented there is intellectual property of Agere Systems, USA.

5

The CRYPTONITE Architecture

According to contracts signed by the author the work presented in this chapter is intellectual property of Agere Systems, USA, with patents pending. With permission of Agere Systems the author is allowed to publish this chapter publicly.

If you plan to use the architecture described in this chapter or parts of it commercially, please contact Mr. Nevin Heinze of Agere Systems. He can be reached as follows

Agere Systems
Processor Architectures & Compiler Research
Mr. Nevin Heintze
4 Connell Drive, Room 4P-739
Berkeley Heights, NJ 07922
U.S.A.

Alternatively, you can reach Mr. Heintze via e-mail (`nch@agere.com`).

5.1 Design Goals

To understand how this architecture was designed the following design criteria have to be taken into account which are

- **Technology limitations** The most critical design parameters for the CRYPTONITE architecture are **single-cycle access** to registers and on-chip memory and **single-cycle execution** of the implemented arithmetic operations to enable maximum processing speed. For technology reasons, namely single cycle access to memory, the operating frequency was limited to 400 MHz. Also, pipelining techniques have to be used since it is infeasible that instruction fetch, decoding, and execution can happen within one 400 MHz cycle.

- Due to **bandwidth demands**, however, an ideal architecture should be able to cope with a summarized data bandwidth of 4 GBit/s. Since this is definitely not achievable with a single, fully programmable processor element a per-element throughput of 500 MBit/s was defined. A production system should then achieve the denoted 4 GBit/s throughput by employing multiple processor elements serving independent data streams.

- These hard parameters, 500 MBit/s throughput at 400 MHz resulting in a relative speed of 1.2 bits per cycle or 0.8 cycles per bit, propagate a parallel architecture with very tight **timing constraints**. This means that functional units have to be designed in a way that the overall signal latency must still fit into a single cycle.

These targets are not only independent of the crypto algorithms, which in turn define which functional units are needed in what configuration, but also have higher priority which means that the proposed architecture is a trade-off between technological specifications and algorithm demands: For example, throughput issues dictate a rather high bits-per-cycle ratio demanding a single-cycle execution model to guarantee maximal throughput. Economical issues limit the frequency to 400 MHz where technology issues make a pipelined architecture mandatory to ease hardware parameters but still be able to provide necessary computation speed.

5.2 General Purpose Architectures – An Alternative?

Today's high-performance general purpose architectures (GPAs) like the Intel Pentium or Itanium families represented through Pentium 4 and Itanium-II are able to deliver an enormous computation power. The decision to use a dedicated architecture at first glance does not seem to be justified for the following reasons:

- Modern GPAs are superscalar architectures and offer computation speeds up to 2.5 GHz
- They are mass products resulting in comparably low prices; also, numerous development systems exist for these processors which programmers are used to.
- Similarly, readily developed interface hardware like PCI bridges, even complete hardware infrastructures – main or mother boards – exist in many configurations which decrease development time significantly.

However, despite these quite convincing facts, there are also reasons against using GPAs:

- Despite their computation power, GPAs still are not able to compete with dedicated processors for certain applications. This is especially true for cryptography. Based on the numbers given in [47] and [110] cycle count of software implementations based on GPAs is still an average factor of 7.55 higher than cycle count for the same algorithms implemented on the CRYPTONITE architecture. In worst case (AES/Rijndael decryption including round key generation) the performance gap reaches even a factor of 21.76.
- High-performance GPAs have an enormous power consumption. For example, members of the Pentium-4 family running at 2.53 GHz use nearly 70 Watts [35]. For this reason, these processors produce a tremendous amount of heat which requires either huge coolers or – in case space matters – complicated and expensive cooling systems.

The chipsets necessary to interface these processors to RAM and peripheral buses, the Memory Controller Hub (MCH) [33] and I/O Controller Hub (ICH) [32], consume another 11 Watts and therefore produce also a noticeable amount of heat. These two factors, power dissipation and heat generation, make these processors almost unusable for embedded systems.

- These processors only provide highest performance if the problem to be computed can be entirely held in cache. If accesses to external RAM are needed, they have to be reordered to hide RAM access latencies. As the investigated algorithms show very tight inner loops there is not much – depending on the algorithm even almost no – time to hide RAM access latencies. Due to the real-time character of the processed data streams it is neither possible to preload a bigger set of data in advance.

- Although the GPAs are comparably low in price, they need special interface chips (within the PC market referred to as"chipset´´) to make them work together with SDRAM or RDRAM and standardized peripheral buses like PCI. Typically, such a system consisting of main board holding the chipset together with a decent GPA running at 2.5 GHz will cost around 1000 Euro as of August 2002.

For the above reasons the development of a specialized architecture is justified: It will provide the necessary computation speed but will only consume a fraction of the power a current GPA needs and therefore produce much less heat. In addition, due to the lesser chip complexity the fabrication costs are expected to be much lower.

5.3 Architecture overview

CRYPTONITE as shown in Figure 5.1 can be divided into two strands each consisting of an Arithmetic Unit (see Section 5.7) and its corresponding Memory Unit (see Section 5.8) controlling the associated data memory. The Memory Unit itself separates into an Address Generation Unit (AGU) being responsible for address generation based on the selected addressing mode (Sections 5.8.1 and 5.8.2) and a Data Input/Output Unit (DIO) (Section

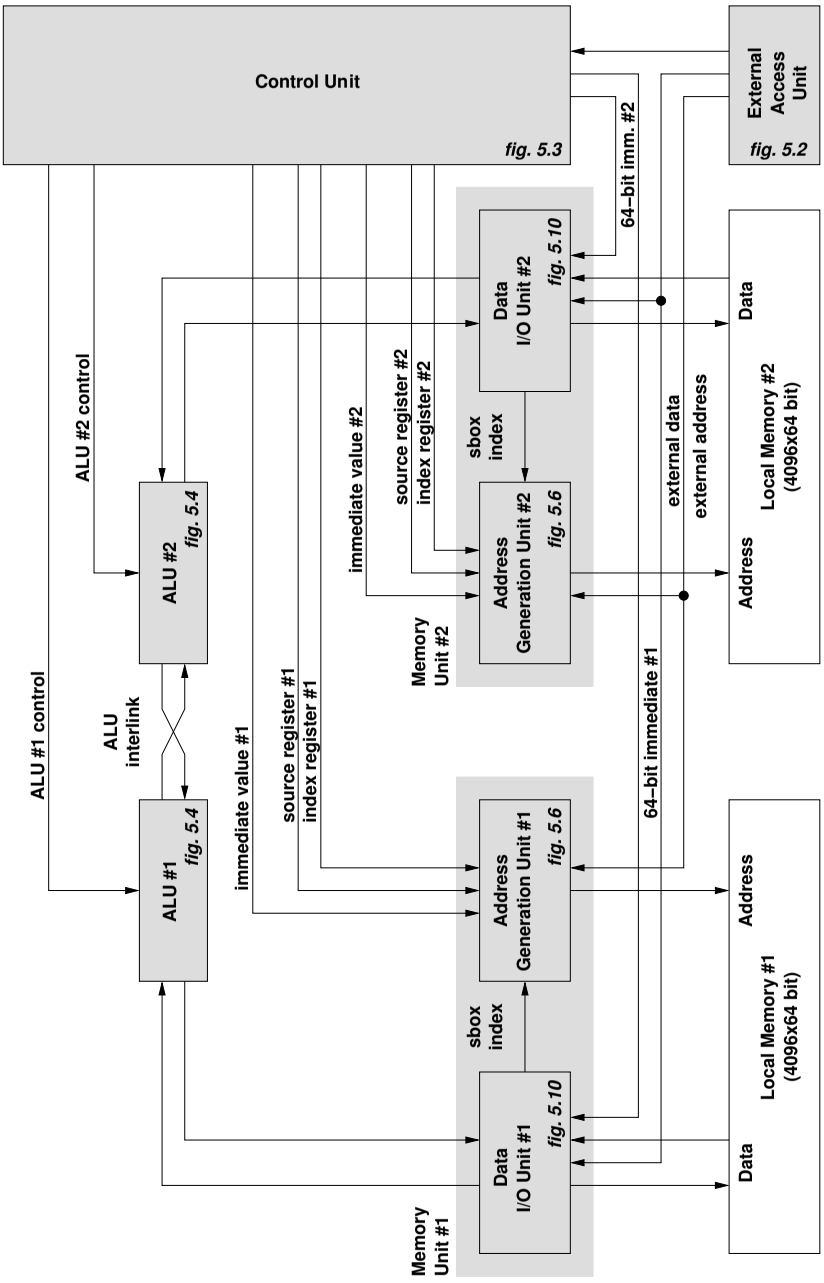

Figure 5.1: Overview over the CRYPTONITE architecture

5.8.3) where the latter holds also a DES-specific unit (Section 5.8.4).

The following architecture presentation will concentrate on single units. The reader should keep in mind that whenever the text refers to ALU registers, local address registers or data input/output registers these register sets are individually present within each strand.

5.4 The Pipeline

CRYPTONITE is targeted towards 400 MHz operating frequency providing single-cycle execution. For this reason, the architecture needs to be pipelined since it is not realistic to assume that instruction fetch, decoding and execution can happen within one 400 MHz cycle. Theoretically, the pipeline could be hidden if the 400 MHz clock is internally tripled resulting in an internal clock frequency of 1.2 GHz. Doing so would, however, cause too high requirements towards chip hardware and design.

Stage	Function	Function
1	Instruction Fetch	$addr \leftarrow PC$ $PC \leftarrow PC + 1$
2	Instruction Decode	create internal control signals $PC \leftarrow immediate\ address$ (PC assignment for immediate branches only)
3	Execution & Writeback	perform operations $PC \leftarrow immediate\ address$ (PC assignment for conditional branches only)

Table 5.1: Pipeline Stages of the CRYPTONITE Architecture

For this reason, a three-stage pipeline is employed as explained in Table 5.1. Besides releasing timing pressure from hardware, the proposed instruction pipeline motivates the use of delayed branches as also shown in Table 5.1. Immediate branches have a one-cycle delay and become effective after pipeline stage 2 (instruction decode); for conditional branches the delay is

2 cycles since these have to wait for the result of the execute stage which means that the new PC value – in case the conditional branch was taken – becomes effective after pipeline stage 3. Already fetched instructions will be processed during the delay cycles.

Branch delay is a characteristic which applies to all pipelined architectures and is therefore known from several architectures like the industrially manufactured AMD AM29K [4] or the educational DLX processor designed by Hennessy & Patterson [52]: Depending on the pipeline length and the branch delay caused through this, one or more instructions following a branch instruction are executed.

For typical crypto algorithms as discussed in Chapters 2 and 3 the control flow is usually not data driven which makes it easy to rearrange code avoiding increase in cycle count or code size. In most cases branch delay can be circumvented by code rearrangement, only in very few situations code increase applies: These are usually loops with bodies smaller than 3 instructions. To avoid padding `NOP` instructions – which negatively affect throughput – loops of this type have to be partially unrolled to allow code rearrangement.

5.5 Access from External Devices

External access to CRYPTONITE is necessary to enable updates of internal look-up tables as needed for external computation of round keys and also to feed in data chunks and key updates within a running computation as well as reading back result data. The physical interface between CRYPTONITE and an external device is realized through the External Access Unit (EAU) as depicted in Figure 5.2.

Whenever an external access to local data memory happens, the requested address plus the actual content of the external data bus is sampled; at the same time a `ready` flag is cleared to signal to the external unit that CRYPTONITE is now busy with the requested data transfer. This flag is also fed into the Control Unit (see Section 5.6). Internally, a maskable interrupt is triggered which stops current execution and performs the requested memory operation by inserting 3 cycles into the pipeline as shown in Table 5.2. This kind of external access interrupt is enabled by default but can

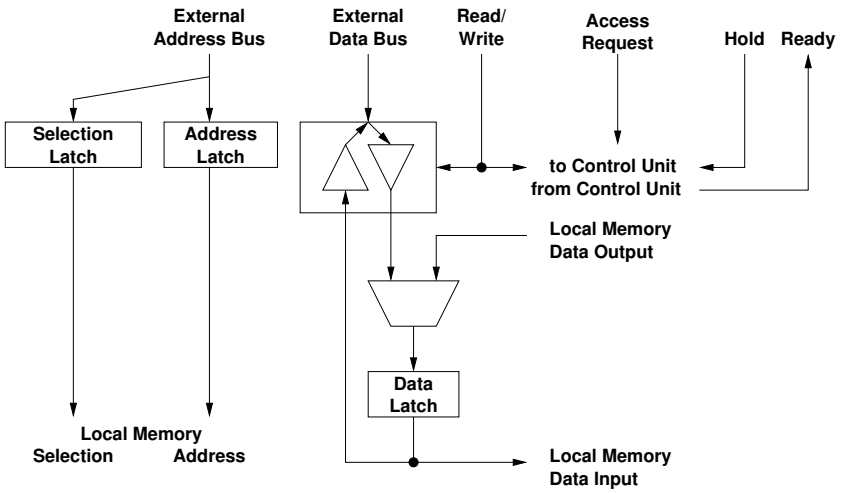

Figure 5.2: The External Access Unit

be disabled using the CU's `DI` command. To re-enable interrupts, the `EI` command is used.

Within the first cycle, the sampled address is put on the local memory address bus; in case of write access, the sampled data is transferred into the data input register, which in term is saved to a shadow register. During the second cycle, the requested data transfer happens and – in case of a read access – the return data is sampled in the data output register. In this case, the previous value of the data output register is saved to a shadow register. In the third cycle, the data input and output registers are restored while the content of the data input register is transferred to the external data register and the `ready` flag is risen.

To allow synchronous transfer, a `hold` input signal exists. Whenever the CU encounters a `HOLD` command, it puts further execution on hold by assigning `NOP` operations to all units and not incrementing the program counter until the hold input signal raises from low to high level. Due to the employed pipeline, `HOLD` will show the same delay as unconditional branches as explained in Section 5.6.2.

This approach was taken to provide a method of accessing local data memory and synchronizing CRYPTONITE to external devices with minimal impact to the core architecture. More heavyweight interface structures being

Cycle	Name	Function
1	Rescue & Transfer	$DIR \rightarrow DIR_{shadow}$ $data\_in_{latched} \rightarrow DIR$ $DOR \rightarrow DOR_{shadow}$ $address_{latched} \rightarrow local\ memory\ address$
2	Memory Access	$read\ access: \quad memory\ data \rightarrow DOR$ $write\ access: \quad DIR \rightarrow memory\ data$
3	Return & Restore	$DOR \rightarrow data\_out$ $DIR_{shadow} \rightarrow DIR$ $DOR_{shadow} \rightarrow DOR$ set `ready` flag

Table 5.2: Interrupt Cycles

essential to directly connect CRYPTONITE to the PCI bus or similar bus interfaces are not covered by this work.

5.6 The Control Unit

Architecturally, CRYPTONITE consists of several independent units being controlled by a central dispatcher which generates the appropriate control signals based on internal state and current instruction word.

This dispatcher is called the Control Unit (CU) and is depicted in Figure 5.3. It contains the instruction word decoder and additional units which are needed for proper program processing; it supports linear addressing and conditional branching as needed for conditional constructs like `IF/THEN` and loops. It also provides immediate values being encoded into the command stream. This command stream is fetched by the CU from program memory.

5.6.1 Supplying immediate values

The CRYPTONITE architecture supports two types of immediate values: Big immediates (64 bit) for initialization of ALU and MU registers plus 12-bit address/immediate values used for counter register initialization (8-bit)

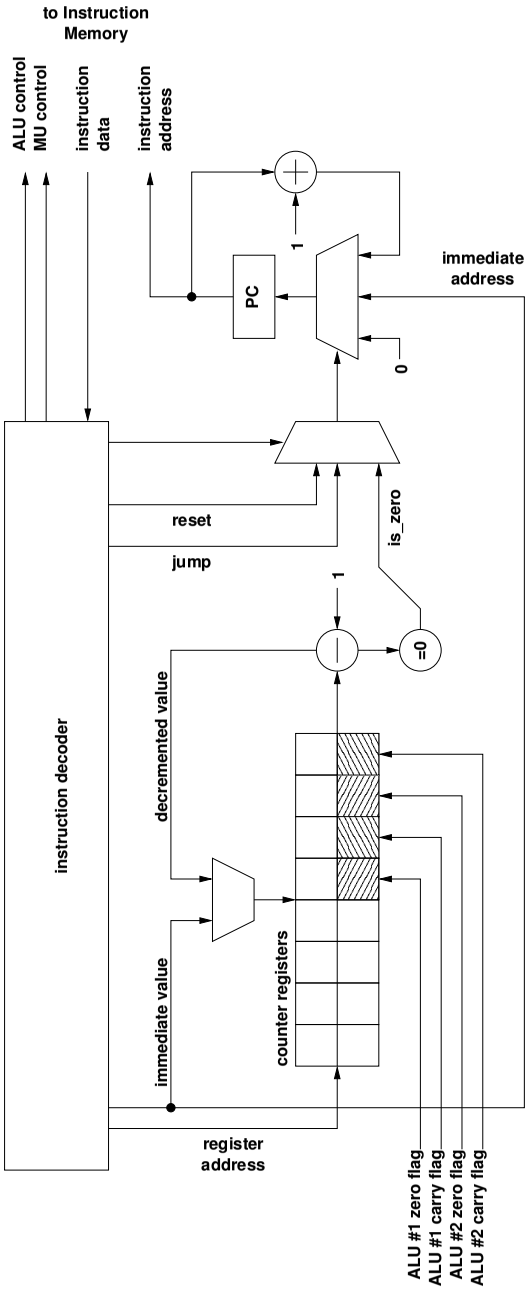

Figure 5.3: Overview over CRYPTONITE's Control Unit

and providing branch addresses for conditional and unconditional branches (12-bit). The 64-bit immediates are named global immediate values (GIV) since they are common for all addressable registers within a so-called strand consisting of ALU, MU and associated memory, where the 12-bit value can be used within the Control Unit and for register initialization within the MAU.

Since initialization with GIVs usually does not take place very often and happens only during an algorithm's initialization stage a special immediate-load operation `LDI` changes the interpretation of the VLIW-style instruction word as explained in Appendix B. During an `LDI` cycle both strands are put in a special load mode and normal operation of ALU and MU is disabled for this cycle.

This decision was made to allow the load of registers with values of up to 64-bit in size while not dramatically the size of the already wide instruction word. This load mechanism fits into the already existing architecture. For this reason, GIVs are internally treated as values being read from data memory.

5.6.2 Counters, looping, and conditional branching

The CRYPTONITE architecture only supports basic down-counting loops based on 12 counter registers `cr0` to `cr11` where each of these registers can be loaded with an immediate value for loop initialization. As implemented in the architecture, loops are pre-decrement which means a register's value is read out, decremented and based on the result a branch is taken or not. In terms of functionality it is similar to the `DJNZ` instruction as known from several processors like the Intel x86 processor family [34] or Zilog's Z80 [137].

While this minimal loop support is sufficient for almost all analyzed algorithms, the IDEA algorithm also needs conditional branching based on ALU results. To make this concept fit into the existing ALU and CU designs, the counter registers `cr12` to `cr15` will sample the current state of the two ALU's zero and sign outputs and serve as flags rather than being used as ordinary counter registers. This way the flag states can be checked by the equal-to-zero comparator as already used together with counter registers `cr0` to `cr11`.

5.6.3 System Control

As mentioned in Section 5.6, the Control Unit generates the necessary control signals from the current instruction word for parametrization of all on-chip units.

The instruction word has 10 parts and contains the coded control words for the five units which are Control Unit, two ALUs, and two Memory Units plus the address/immediate field. A description of the instruction word format is given in Appendix B.

5.7 The Arithmetic Logical Unit

Crypto algorithms usually employ a variety of basic arithmetic operations like Boolean operations, addition/subtraction, and bit-shifting or -rotation. Depending on the algorithm quite a number of intermediate results have to be stored which ideally can be held in registers rather than external memory to avoid latencies caused by external memory access.

However, a register file must not exceed a specific size to follow the technological specifications. Where a huge and massively interconnected register file, i.e. having many input and output ports, will certainly ease program realization it will easily exceed technology limitations.

In this chapter an arithmetic logical unit (ALU) will be described which is not only suited for fast and efficient computation of typical crypto algorithms and but will also fulfill the technological requirements as listed in Section 5.1.

5.7.1 Overview

CRYPTONITE's ALU consists of 3 functional units which are the Register File (RF), the Arithmetic Unit (AU), and the XOR Unit (XU) as depicted in Figure 5.4. The following sections will discuss these units in detail.

5.7.2 The Register File

The RF holds four 64-bit registers, `r0` to `r3`, which can be individually loaded from two internal buses called A and X. The A bus transports the

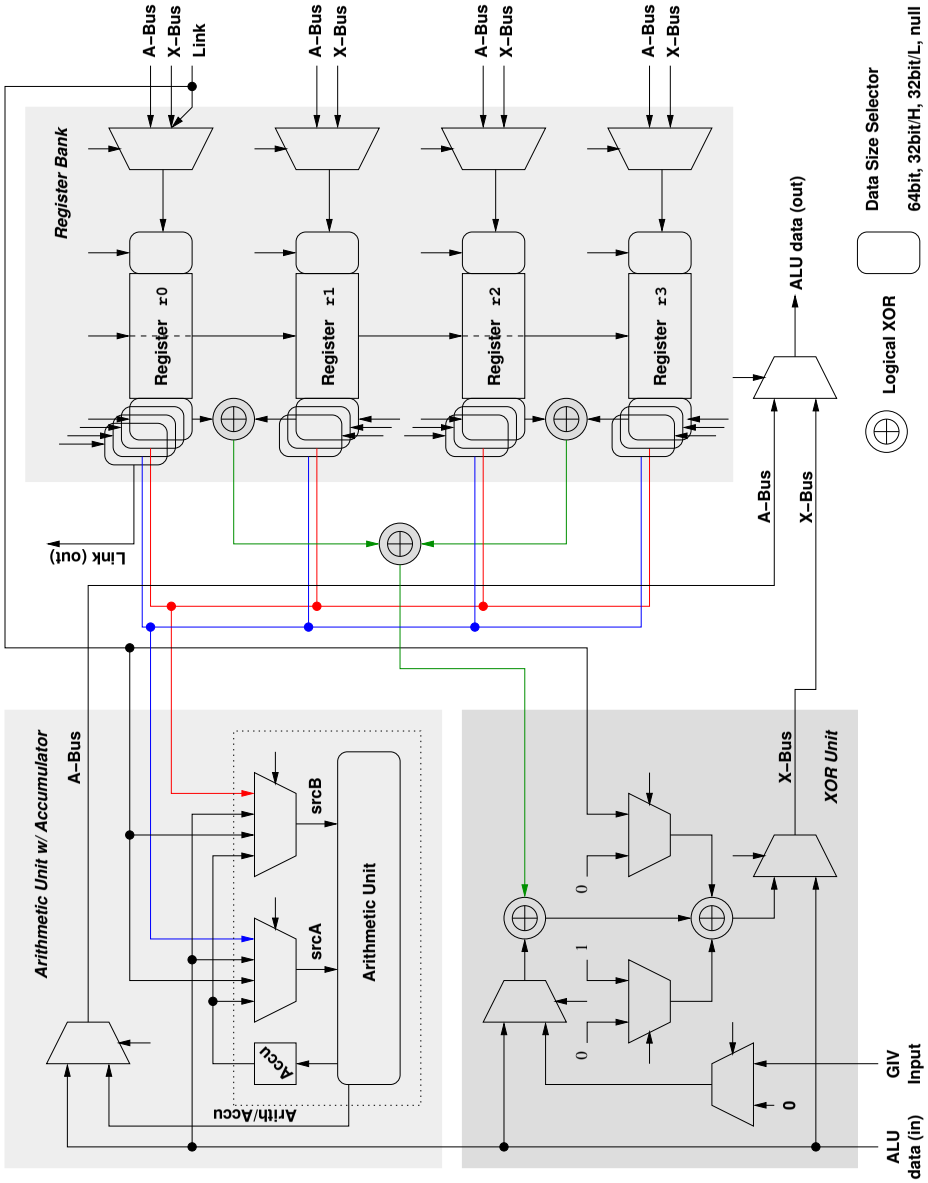

Figure 5.4: Overview over CRYPTONITE's ALU

result of a previous arithmetic operation where the X bus supplies the result of a previous XOR operation. For exchange of data between the two ALUs, register `r0` of each ALU register file can also be loaded from the ALU interlink bus.

Once a bus is assigned to a register the input data size selector determines whether the complete 64-bit data, or the high or low 32-bit portion should be loaded into the register. Alternatively, the data size selector can be used to simply apply a zero value for clearing the register as already mentioned in Section 4.2.1.

At the output port of the register a similar data size selector is employed which again enables selection of the entire 64-bit, or high or low 32-bit of a stored value, or to "mute" the register by outputting a zero value without changing the register's content. Each register has three output ports for addressing the XOR unit and the source inputs #1 and #2 of the ALU. There is no immediate way of storing a register value to memory (or the data input register, respectively). This is only possible for results of arithmetic or boolean operations. In case the unaltered register value needs to be stored, the used AU or XOR unit has to perform a NOP operation. Direct read-out of register values was omitted since this would need another output unit per register and four additional 64-bit datapaths within the chip.

Again, register `r0` can serve for interlink purposes and has a fourth output port for feeding the AU interlink.

5.7.3 The Arithmetic Unit (AU)

The AU is able to perform all standard arithmetic and Boolean operations plus specific functions needed to improve speed on AES/Rijndael calculation based on input values provided through the Register File registers, the associated Memory Unit's Data Output Register, and the AU's internal accumulator. The accumulator can be used for storage of intermediate results and can be loaded from either of the two source inputs. Its use is optional.

Besides basic arithmetic (addition & subtraction) and Boolean functions (`AND`, `OR` & `XOR`) the ALU employs a barrel shifter which can be used to shift or rotate either a 64-bit value or two 32-bit values by an arbitrary number of bits and enables special functions as explained in Section 5.7.3.1. The AU also offers a `swap` instruction by which the 32-bit halves of a 64-bit quantity can be swapped; as a result of this operation and the presence

of a barrel shifter, also combined `swrt` operations based on swapping and left-rotation are possible which are employed for fast AES/Rijndael implementation [45].

The analysis of IDEA and RC6 motivated the use of a parametrizable multiplier. For example, IDEA uses 16-bit∗16-bit mod_{16} multiplications, RC6 – depending on its configuration – employs 32-bit∗32-bit mod_{32} to 64-bit∗64-bit mod_{64}. Such a unit would greatly enhance the performance of these algorithms and also support asymmetric cryptography. For the CRYPTONITE architecture, this unit is optional because of the lesser relevance of IDEA and RC6 and since its focus is symmetric If implemented, only the modulo operation should be configurable with respect to the 64-bit architecture. The input data size is always 64-bit as with all other ALU units.

The result of an arithmetic operation is put on the internal A bus and can be routed to any of the four registers or the Data Input or Ouput Register of the associated Memory Unit.

5.7.3.1 Special Functions

As a result of the AES/Rijndael analysis a set of instructions was developed to speed up that algorithm. However, unlike the DES Unit which is only applicable to (3)DES, these instructions are not AES/Rijndael-specific and can be used for other algorithms as well.

These functions sometimes operate in 32-bit quantities. In such cases, indices h and l denote the referring 32-bit portion of a 64-bit quantity with h being the leftmost and l the rightmost one. Numeric indices refer to the corresponding byte (8-bit portion) of a 64-bit quantity with byte #0 being the rightmost one.

- The **upper64** operation takes two 64-bit quantities *x* and *y* and outputs the 64-bit quantity $x_7x_3y_7y_3x_6x_2y_6y_2$ where the indices denote the referring byte within the input values.
- Similarly, **lower64** generates the 64-bit quantity $x_5x_1y_5y_1x_4x_0y_4y_0$.

upper64	$f(x, y) = x_7x_3y_7y_3x_6x_2y_6y_2$
lower64	$f(x, y) = x_5x_1y_5y_1x_4x_0y_4y_0$

Table 5.3: The **upper64** and **lower64** functions

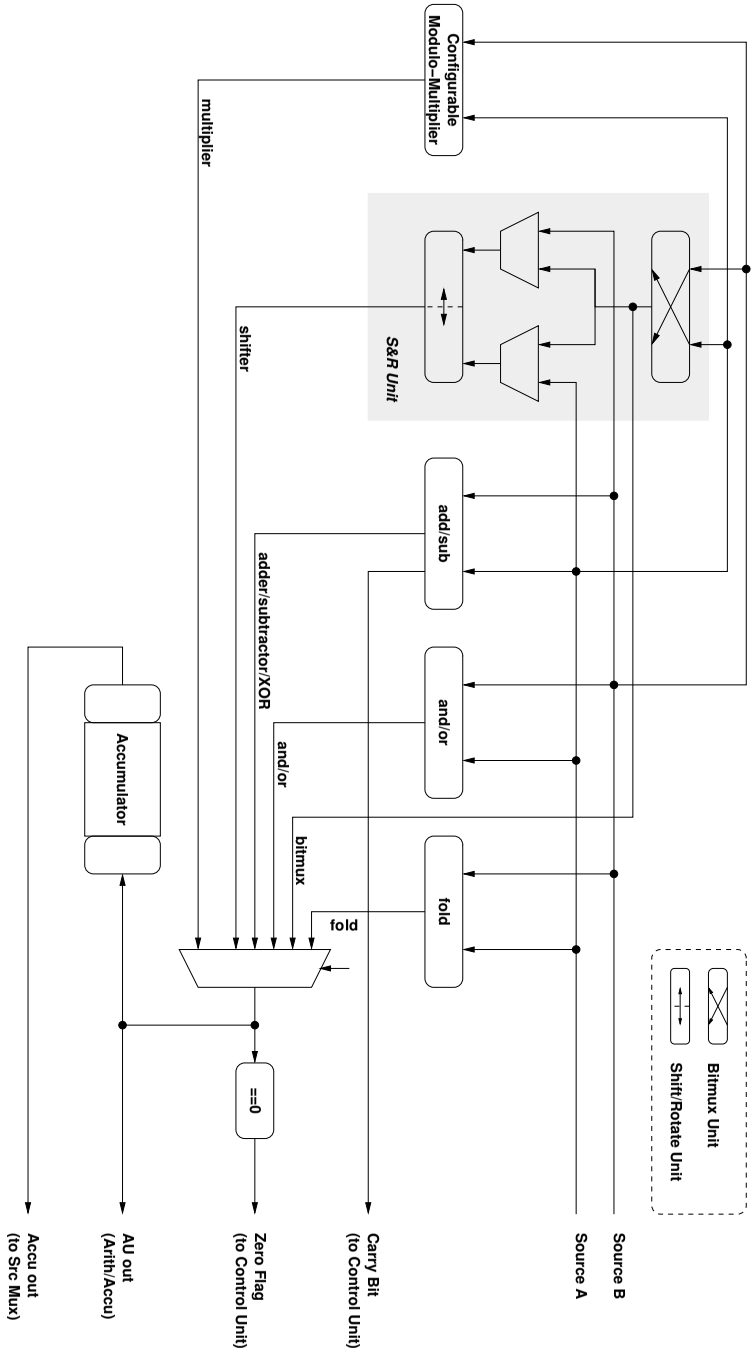

Figure 5.5: Detail of the arithmetic unit

- The **swap** instruction takes two 64-bit quantities x and y and generates the 64-bit result $x_l y_h$ (or $x_h y_l$ depending on the parametrization of the register output multiplexers). The indices denote the referring 32-bit half of an input value.

- The **swrt** instructions are based on the **swap** instruction and rotate operations; they take a two 32-bit quantities like the **swap** instruction but rotate these by individual amounts as shown in Table 5.4.

swap	$f(x,y) = x_l y_h$
swrt0	$f(x) = (x_h \lll 8) y_l$
swrt1	$f(x) = (x_h \lll 24)(y_h \lll 16)$

Table 5.4: The **swap** functions

- A set of **fold** instructions is included. These perform a series of XOR operations based on two input values and intermediate results as shown in Table 5.5.

foldb32	$f(x,y) = \forall 0 \leq i \leq 3 : \begin{cases} i = 0 : x_0 \oplus y_0 \\ i \neq 0 : f_{i-1} \oplus y_i \end{cases}$
foldb64	$f(x,y) = \forall 0 \leq i \leq 7 : \begin{cases} i = 0 : x_0 \oplus y_0 \\ i \neq 0 : f_{i-1} \oplus y_i \end{cases}$
foldw64	$f(x,y) = \forall 0 \leq i \leq 1 : \begin{cases} i = 0 : f_h = x_l \oplus y_h \\ i = 1 : f_l = f_l \oplus y_l \end{cases}$

Table 5.5: The **fold** operations

Within the AU most of these functions are performed by the so-called **bitmux unit** which performs the mentioned **swap**, **upper64** and **lower64** operations as shown in Figure 5.5. This unit also feeds the barrel shifter to allow complex swap and rotate operations. The **fold** operations are computed in a separate fold unit.

5.7.4 The XOR Unit

The exclusive-or (XOR) operation is the key operation of all cryptological algorithms which makes it vital to implement a fast and efficient method for performing a series of XOR operations.

In CRYPTONITE the XOR Unit not only performs the XOR operation on two to six operands consisting of the register file together with the content of the associated Memory Unit's data output register `dor` and an optional sixth value provided through the ALU interlink as described in Section 5.7.5, but also allows the negation of a bit pattern. For that reason the XOR unit has an internal switch to optionally invert the result of an XOR operation.

To minimize on-chip buses the XOR unit is partly embedded into the data path. The first XOR stage already sits within the register file and combines the values of register 1 and 2, or 3 and 4 respectively. The result of this first XOR stage is then combined in another stage before it is fed into the XOR unit's main part as depicted in Figure 5.4. This method of embedding an arithmetic function into the physical on-chip data path is currently filed for patenting.

Like register `r0` of the Register File also the XOR Unit has a special interlink input enabling to forward the value of the opposite ALU's register `r0` directly into the XOR Unit's main part to be concatenated with the result of the register combination if needed. Finally, the overall result can be inverted and the result is put onto the ALU's X bus to be routed to any of the four registers `r0` to `r3` or the associated Memory Unit's data input `dir` and data output registers `dor`. The XOR unit furthermore can distribute a GIV (provided by the CU as mentioned in 5.6.1) to the aforementioned registers.

5.7.5 Avoiding register pressure: Crosslinking the arithmetic units

The original design considered an 8-register file with full connectivity of registers to both ALUs. This concept was cancelled for speed reasons since full connectivity would mean more read and write ports per register and less optimal routing, especially long wires.

Instead, the concept of individual register files per ALU was taken with only four registers per file. This turned out to be enough for most cases; however, it limited the possibilities of parallel computation of independent

values since the computation of a single formula had to be fully assigned to one ALU. Also, some algorithms turned out to lay rather high pressure on the register file during a sequential calculation forcing the writeback and reload of intermediate values to and from memory.

For this reason an interlink mechanism was designed to allow forwarding of register contents to the opposite ALU. To minimize technological impact, mainly speed and delay issues, link exchange takes only place between register `r0` of each ALU. To allow immediate use of these values within ongoing computations, it is additionally possible to directly feed the current value of register `r0` into the opposite ALU's arithmetic and XOR units. Doing so avoids a one-cycle delay otherwise occuring for value transfer from one register `r0` to its opposite ALU's sibling.

5.8 The Memory Unit

Each ALU has a corresponding Memory Unit (MU) for data storage. The MU consists of an address generation unit (AGU) responsible for proper address generation based on a selected addressing scheme (see Section 5.8.1), and a Data Input/Output Unit (DIO) (see Section 5.8.3) which is basically used for data transport but also contains the DES Unit as explained in Section 5.8.4. Attached to the MU is the local data memory as depicted in Figure 5.1.

5.8.1 Address Generation Unit

The AGU as depicted in Figure 5.6 is responsible for generating addresses during data memory access and inherently supports direct and S-Box addressing; indexed and modulo addressing are also supported by the AGU, but computation of addresses takes one cycle. When using these addressing modes the programmer must remember this post-increment/post-modulo model (as opposed to pre-decrement as used within the control unit for loop support). The complete list of supported addressing modes is listed in Table 5.6.

To enable external access to data memory, it is also possible to supply an external address instead of the generated one. This allows an external processor, a network protocol processor for example, to directly read computed

results or feed in new data like plain-text chunks or key updates; similarly, look-up tables can be easily exchanged using an external processor.

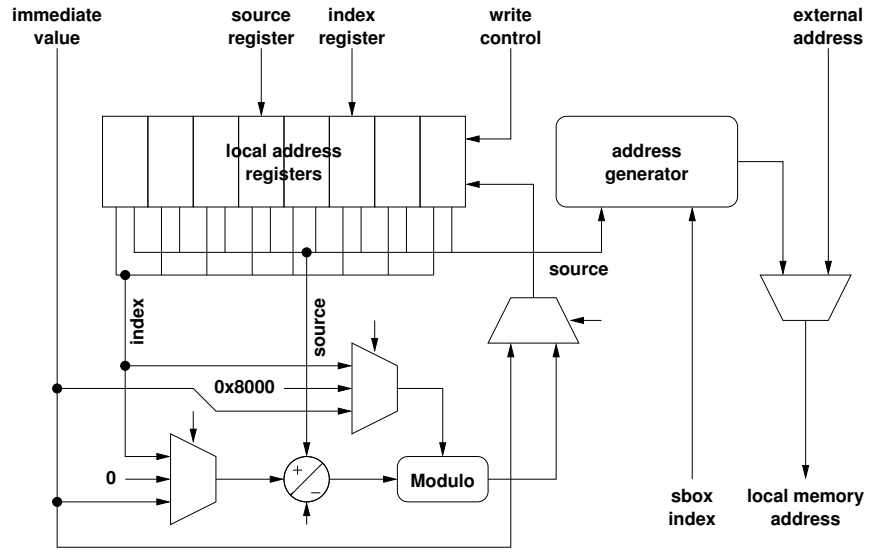

Figure 5.6: The Address Generation Unit

The basis of all address generation is the Local Address Registers (LAR). In the simplest configuration, direct addressing, a LAR holds an address pointer which is directly used as a memory address; indexed addressing is supported in two ways, using immediate index values as supplied by the control unit (which means that the index value is embedded into the opcode) or index values derived from other LARs. With both methods, the index value is added to an address supplied by a LAR. Also, as mentioned before these modes provide deferred results. The reason for doing so was to keep the memory address generation as fast as possible to be able to provide memory data right in time and minimizing the address setup time for the on-chip memory.

With these three modes the AGU also supports modulo addressing to automatically keep a generated address within a dedicated memory segment. This is especially useful for hash algorithms like MD5 or SHA-1 where a table of 16 (20 for SHA-1) entries is accessed using indexed addressing with an offset pointing into the table and an index increment greater than one. Figure 5.8 illustrates the effect of modulo addressing applied to indexed

Addressing Mode	Address Computation	LAR Update
direct	$addr = LAR$	
", w/ register modulo	$addr = LAR_x$	$LAR_x = LAR_x \,\%\, LAR_y$
", w/ immediate modulo	$addr = LAR_x$	$LAR_x = LAR_x \,\%\, idx$
S-Box	$\forall 0 \leq i \leq 7 : addr_i = (LAR \wedge \texttt{0x7f00}) \vee idx_i$	(LAR unchanged)
immediate-indexed	$addr = LAR$	$LAR = LAR + idx$
ditto, w/ register modulo	$addr = LAR_x$	$LAR_x = (LAR_x + idx) \,\%\, LAR_y$
register-indexed	$addr = LAR_x$	$LAR_x = LAR_x + LAR_y$
ditto, w/ immediate modulo	$addr = LAR_x$	$LAR_x = (LAR_x + LAR_y) \,\%\, idx$

Addressing modes written in italics are based on architectural side-effects and have not been designed in by purpose.

Table 5.6: Addressing modes supported by CRYPTONITE's AGU

addressing: The dashed location would be addressed with straight index addressing; using modulo-4 addressing, however, the resulting address is location 1 instead of 5.

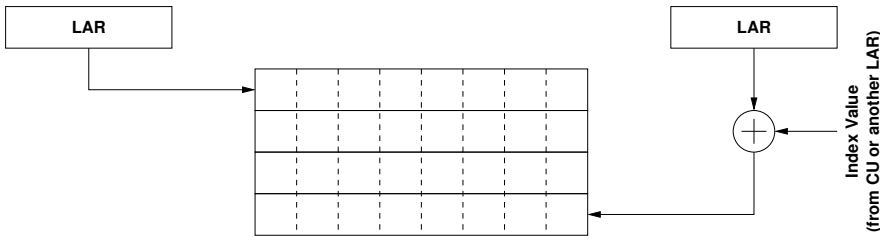

Figure 5.7: Direct and indexed address generation

Like the indexed modes also the modulo operation is deferred. For hardware reasons the modulo operation is limited to powers of two which makes it possible to implement this function using the Boolean `AND` operation as opposed to an extensive integer division unit.

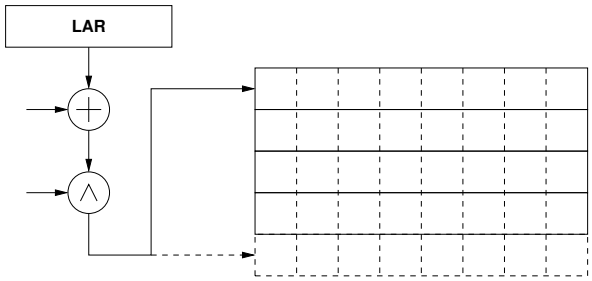

Figure 5.8: Modulo addressing

5.8.2 Speeding up table lookups: S-Box support

A very special feature of the AGU is the S-Box addressing mode. Unlike the previously described modes which address the memory in 64-bit mode, S-Box addressing makes use of the memory's feature to be addressed as eight 8-bit quantities instead of a single 64-bit chunk. The use of 64-bit memory

as eight user-definable 8-bit S-Boxes is a unique feature not found in crypto processor architecture so far and is also currently filed for patenting.

The address per 8-bit chunk is derived from an LAR supplying the S-Box address plus the S-Box index. Since each index is 8 bits in size the address space of an S-Box is 256 addresses.

With S-Box addressing mode, the base LAR provides the S-Box number where the index value contains the eight 8-bit indices into the selected S-Box as depicted in Figure 5.9. S-Box access can happen only aligned to 2048-byte boundaries (256 addresses times 8 bytes); unaligned addressing is not possible since the lower address bits of the LAR are replaced by the corresponding S-Box index. This method was chosen over the use of an adder for timing reasons to not increase latency of this critical data path.

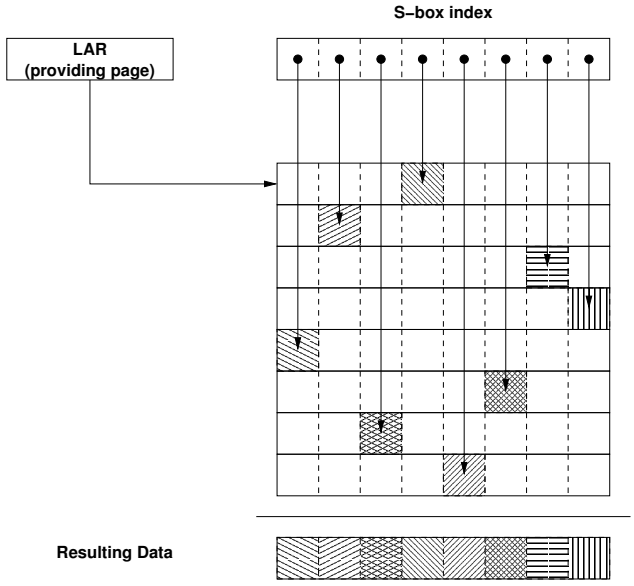

Figure 5.9: S-Box address generation

5.8.3 The Data Input/Output Unit

Due to the very tight timing of the architecture it is necessary to provide a method of synchronizing data for long distance transport as needed for transfer of data between embedded SRAM and the ALUs.

The DIO as depicted in Figure 5.10 consists of a Data Input Register (DIR) where data coming from the ALU is written into prior to be stored to memory. Respectively, a Data Output Register (DOR) exists which will buffer a value transferred from memory to ALU. Since the DIR will also provide the S-Box index coming either from ALU or a previous DES round, the DIR input can select either ALU data or a result coming from the DES unit discussed in Section 5.8.4. As already mentioned in Section 5.7, the DOR can serve as an additional source or destination for arithmetic operations. In Section 5.5 it has already been pointed out that DIR and DOR have accompanying shadow registers used to rescue and restore the original register values during external access interrupt handling.

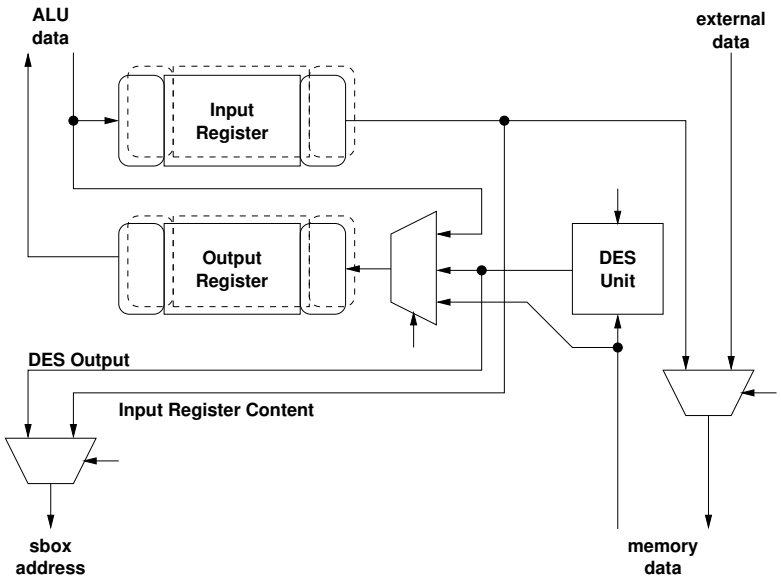

Figure 5.10: Data I/O from embedded SRAM

Besides these registers, the DIO also contains a DES Unit which will provide the necessary functionality needed for fast and efficient (3)DES computation. To allow further processing or storage of a DES operation's result, these can be either directly fed back to memory or into the ALU through the output register. Since DES provides independently computed S-Box indices, it can directly access the S-Box address path providing the

necessary 64-bit S-Box index value. The following Section 5.8.4 will discuss the DES Unit in detail.

5.8.4 The DES-specific unit

The heart of the DES algorithm is the non-linear function based on the so-called S-Box table lookup. All other functions within the DES algorithm are mainly bit-shuffling plus shifting of the two key halves.

These bit-shuffling functions are DES-specific and are not reusable for other functions. Similarly, the 28-bit rotating shift operation does not apply to other cryptographic functions which usually operate on multiples of 8-bit, mainly 32-bit. For this reason and since the main cryptographic operation of DES is an S-Box lookup the DES-specific functions were transferred from ALU into the memory unit. Doing so, the ALU can be less complex; in addition, the necessary physical data paths for DES computation are kept as short as possible which potentially supports higher clock frequencies.

5.8.4.1 ALU-based initial approach

The initial approach was to implement the bit-shuffling functions into the ALU and allow another shifter configuration for the 2x 28bit operation. The idea behind this approach was to avoid a huge, monolithic function for the sake of an increased cycle count.

- DES input permutation
- DES data expansion
- DES key compression
- DES P-Box permutation
- DES final permutation
- 2x28-bit rotation configuration for barrel shifter

Table 5.7: Special functions for non-monolithic DES implementation

As already explaind in Section 3.3.2 this leads to 5 cycles per DES round based on the 5 DES-specific ALU functions. In addition, a special shifter configuration for DES key computation is needed resulting in special operations as listed in Table 5.7.

5.8.4.2 Speed-up by pipelining DES

Looking at DES at a more coarse-grained scale, it turns out that DES is easy to pipeline: All specific functions apart from input and final permutation can be melted into two monolithic functions which as listed in Table 5.8. Besides these two functions only input and output permutation apply which can be realized as translation functions embedded into the data path of the L and R variables during computation as shown in Figure 5.11.

However, these monolithic functions are clearly DES-specific and can certainly not be reused for anything else. It would not be sensible to bloat the ALU with this heavy-weight functional unit as shown in Figure 5.11, especially since both functions are tighly coupled to memory lookups which makes the memory unit a by far more reasonable place to integrate this unit into.

Separating the DES unit also means that it needs to contain the necessary control logic for providing the round constants (shifting amount) since these values cannot be externally applied anymore.

For this reason, a clearable 4-bit register with an associated incrementer is employed for address generation. This address is fed into a small round constant memory which contains of 16 entries of 2 bits each to provide a single round's shift amount. The address can be inverted resulting in a backward address count as needed for decryption.

• expand R_{i-1}, shift & compress key, and XOR the results
• permutate the S-Box result using P-Box shuffling and XOR this result with L_{i-1}; forward R_{i-1} to L_i

Table 5.8: Special functions for pipelined DES implementation

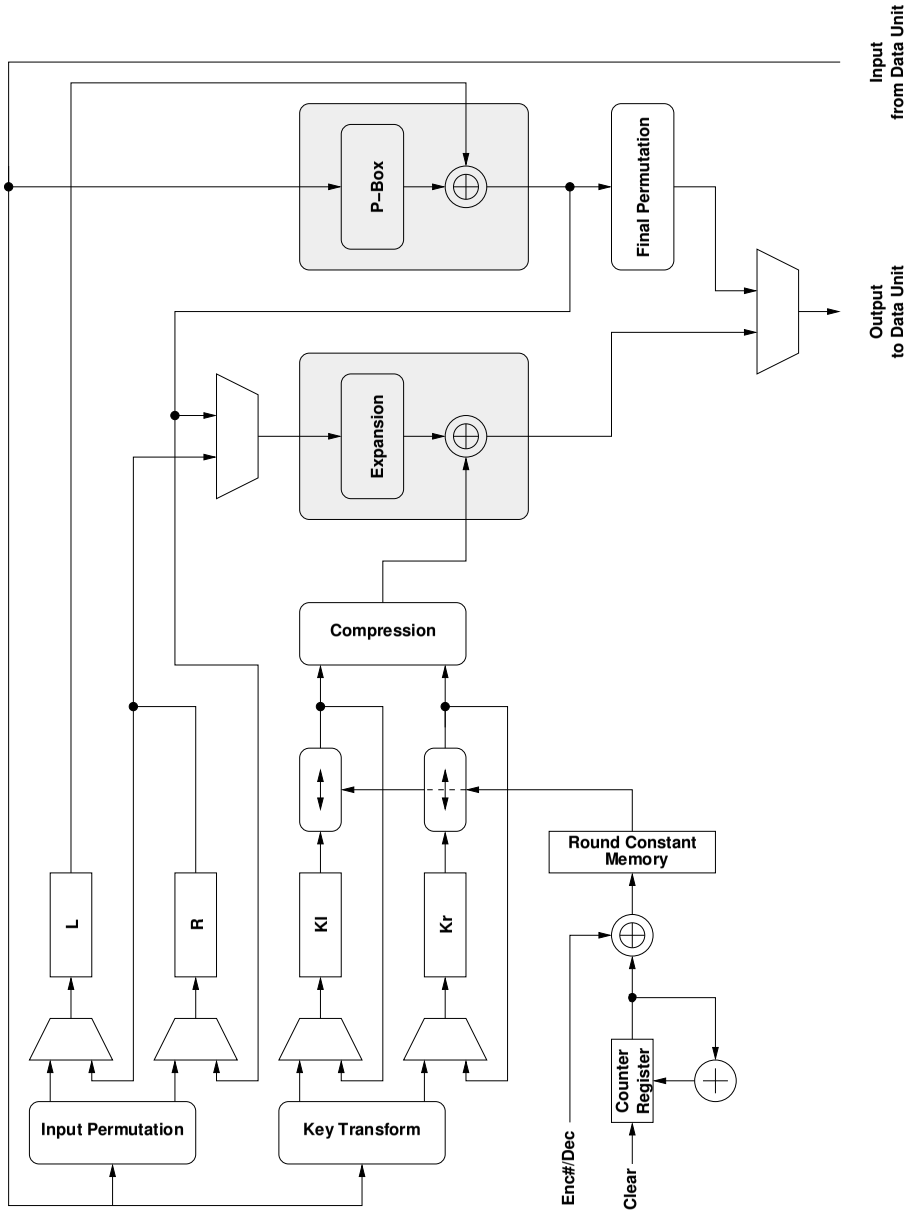

Figure 5.11: Detail of CRYPTONITE's DES Unit

5.9 Summary

In this chapter, the architecture of the proposed CRYPTONITE architecture was presented and reasons were given why such a specialized architecture makes sense even with powerful modern general purpose processors being able to deliver similar computation power.

This architecture consists of a Control Unit (CU) parsing the instruction stream and creating control signals for the other units. These other units form two strands consisting of Arithmetic Logical Unit (ALU) and its corresponding memory unit (MU) and associated SRAM-type data memory. Besides, the CU is responsible for data transfer from program to data memory and vice versa. It also controls program flow: For this reason it employs counter registers, status flags and a small decrement/compare logic to allow conditional branching.

The ALU further divides into an Arithmetic Unit (ALU) including the accumulator, the XOR Unit (XU) and a register file (RF) holding four 64-bit general purpose registers. Memory access uses two buffering registers named Data Input (DIR, for memory writes) and Data Output Register (DOR, for memory reads). Computation results can be stored in either DIR, DOR or both, where the DOR can be read back by AU and XU through an individual input. Storage of RF register values to memory are only possibly through the DIR. The ALU employs a crosslink mechanism which allows forwarding either register `r0` or an ALU result to the sibling ALU.

Similarly, also the MU dissects into further units which are Address Generation Unit (AGU) and Data I/O Unit (DIO). The AGU holds two sets of eight local address registers; these sets are individual for read and write operations. It contains simple logic to support various addressing modes as explained in Section 5.8.1, S-Box addressing being one of these allowing 8 parallel S-Box lookups within one cycle. A DES-specific unit is part of the DIO.

The presented architecture is the result of the algorithm analysis presented in Chapter 3. It is as general as possible and contains only special functional units and operations where needed for speeding up certain algorithm computation.

Based on the algorithm analysis in Chapter 3 this architecture provides the necessary computation power to allow fast and efficient implementation of several crypto algorithms, but was kept as simple as possible to enable

high clock rates when being implemented in hardware. The performance numbers of both, algorithm implementations and hardware realization, will be presented in Chapter 6.

6

Implementational Aspects

The previous chapters focused on algorithm analysis (Chapter 3), how these algorithms influenced the CRYPTONITE architecture (Chapter 4), and finally presented the resulting architecture (Chapter 5). This chapter now will present actual results based on the proposed architecture to answer how the algorithms will perform on CRYPTONITE and what the technology implications are.

Since no complete CRYPTONITE processor has currently been built, the numbers presented for CRYPTONITE are based on software simulation and synthesis results of dedicated hardware parts. Although the author is confident that these simulation results accurately reflect reality, he wants the reader to be clear about the numbers not being based on measurements performed on existing hardware.

6.1 Software Implementation

This section concentrates on the algorithm implementations in software based on the architecture described in Chapter 5 to provide comparative numbers for processing speed and data throughput. To collect these numbers, test implementations in of each algorithm were created. These implementations, with the exception of AES, were run on a self-written simulator to verify the correctness of these implementations. This simulator is basically a C library emulating the CRYPTONITE assembly language as described in Appendix C by providing appropriate function calls together with sanity checks to ensure that an algorithm implementation does not vi-

olate CRYPTONITE hardware contraints. AES was implemented using the Standard Modelling Language (SML) language and runtime environment. Both simulation environments do not provide any timing analysis besides cycle count; hardware parameters will be investigated in Section 6.2 using Synopsys [118] hardware development software.

Cycle count of all algorithms implemented using the aforementioned softwre simulators is summed up in Table 6.1. Comments to each implementation are given within the referring subsection. Where appropriate, cycle count is provided as a formula $s+r*c+p$. Here, s represents number of cycles used for setup, $r*c$ gives number of rounds times number of cycles per round, and p represents cycles used for clean-up and post-processing.

Algorithm	**Cycle Count**	**Throughput (@400MHz)** Bits/Cycle	MBit/s
AES-128/128 ☒	2+9*8+6=80	1.6	640
AES-128/128 □	2+9*7+5=70	1.83	732
DES	4+15*2+1=35	1.83	732
3DES	3*(30+5)=105	0.61	244
IDEA	8*11+2=90	0.71	284
RC6	3+20*(7+3)+3=206	0.62	249
MD5	2+16*(8+8+7+8)+6=504	1.02	406
SHA-1	2+16*(8+7+8+7)+6=488	1.05	420

Table 6.1: CRYPTONITE Performance Data for Selected Algorithms

6.1.1 AES/Rijndael

For comparison, the 128/128-configuration of AES/Rijndael [45] was implemented in two ways. The simpler implementation (□) relies on precomputed round keys, the second implementation (☒) creates needed round keys on the fly from the supplied user key.

Cycle count sums up to two initial cycles, 8 cycles (7 cycles for the simple version) for the first 9 rounds and 6 (5) cycles for the last round and writeback. Performance numbers are given for both versions.

6.1.2 DES and 3DES

These implementations are fairly trivial since DES is supported through special instructions. Since these instructions are strictly sequential and only one key register exists, it does not make any difference if 3DES is used in two- or three-key configuration since the key register needs to be reloaded for every key change.

The given formula for cycle count represents a 4 cycle init phase followed by 16 rounds per 2 cycles each and a final cycle for writeback.

6.1.3 IDEA

The IDEA implementation does only include the plain encryption/decryption algorithm; key generation is omitted since it needs division and modulo arithmetics which are not included within the CRYPTONITE architecture. Furthermore, it makes the assumption, that a mod_{16} 16-bit multiplier is present.

For these reasons, the cycle count cuts down to 8 rounds per 11 cycles plus two cycles for final processing and result writeback.

6.1.4 RC6

RC6 in AES configuration showed an interesting result; although this algorithm configuration seems to be perfectly suited for the CRYPTONITE architecture it needs 11 cycles (8 for computation, 3 for swap) per round. This implementation partly omits the round key generation, which means that the implementation assumes that the key array has already been written to memory. Also, it makes the assumption that a mod_{32} 32-bit multiplier is present.

Then, the cycle count distributes on 3 cycles each for init and postprocessing plus 220 cycles for the 20 rounds.

6.1.5 MD5 and SHA-1

Due to the similarities of the two algorithms their implementations are fairly identical. SHA-1 shows slightly better performance since it incorporates two “fast” rounds of 7 cycles as opposed to only one for MD5.

Initialization takes 2 cycles, postprocessing 6 cycles. The main impact comes from the 16 iterations per each of the 4 rounds and equals 480 (SHA-1) or 486 (MD5) cycles.

6.2 Hardware Implementation

Within this section, the hardware parameters are evaluated to generate realistic numbers for core speed and die or logic size. These numbers can be compared against parameters provided by core vendors; however, these are highly dependent on the technology used. Usually, the numbers are based on FPGA families like Altera APEX [8, 7] or Xilinx Virtex [132, 133] and ASIC fabrication processes from TSMC [119] and UMC [125]. Although they lose relevance when considering chip mask design, the numbers still give an estimation about the proposed 400 MHz core speed being possible or not.

For testing and implementing the VHDL models as well as creating timing and chip usage reports the Synopsys software package running under Sparc/Solaris together with the FPGA compiler provided by Xilinx [65] was used.

6.2.1 Technology Requirements

The CRYPTONITE architecture consists of the processing unit itself and associated memory which means that the used fabrication process must be able to provide fast and efficient static memory (SRAM). This SRAM has to be fast enough to deliver data content within the running cycle.

As for memory size, the proposed address space requires 12 address lines. With these, 4kBit*64 equivalent to 32kB of plain data or 16 sets of 8 S-Boxes can be held in memory which the author believes to be adequate for future algorithms.

When realized on FPGAs and ASICs, a dedicated hardware multiplier would help greatly. Otherwise, the multiplier has to be realized using standard techniques which are quite space inefficient and create long delays.

6.2.1.1 Register File

The register file consists of four identical registers with an input and output multiplexer. These multiplexers make possible to read in data from either A or X bus, for register `r0` also the interlink serves as an additional input.

6.2.1.2 Arithmetic Unit: Shift & Rotate Unit

This is the most complex unit within the arithmetic unit consisting of the swap multiplexer, a barrel shifter and the final bit splicer containing the longest single data path.

6.2.1.3 Address Generation Unit

The AGU is the by far most timing critical unit. It needs to provide the memory address fast enough to comply with the memory's setup time allowing single-cycle memory accesses regardless of selected addressing mode.

6.2.2 Implementation Results

For test purposes Xilinx' [132] latest FPGA family Virtex-II Pro [133] was chosen since this family is almost the biggest FPGA family on the market. The reason doing so was to avoid running into resource limitations like too few logic elements, interconnects or I/O pins when synthesizing and analyzing parts of the CRYPTONITE architecture. However, it appeared that even with the largest model the sheer number of employed signals made it impossible to test specific behaviour like the ALU interlink timing for instance. For this, two complete ALUs had to be instantiated with a rather large amount of input and output signals. Interestingly, this problem would not arise for testing the complete design, since these numerous signals would not occur as physical I/O signals on the final design but only as internal signal nodes.

For this reason, only small parts of the CRYPTONITE architecture could be synthesized and analyzed to get an idea about the signal delays within the chip.

These results were very promising: Without further optimization the designs reach speeds up to 141 MHz; the limiting factor here is neither logic nor routing, but delay caused by routing the signals to the I/O pads:

Unit	Speed		Size			
	MHz	Logic	IOBs		Slices	
Register File	122.7	22%	749	(87%)	929	(4%)
Swap&Rotate	99.8	34%	201	(23%)	545	(2%)
Adress Generation	141.3	50%	188	(22%)	274	(1%)

Table 6.2: Synthesis Results on Xilinx Virtex-IIp FPGA family

For the Register File the delay ratio caused by logic is only up to 22%, for the Swap&Rotate Unit this value reaches up to 35%; interestingly, the logic needed for address generation makes 50% of the delay ratio.

Since these results are based on a straight-forward implementation and do not make use of any special feature offered by the Virtex-IIp FPGA family, it can be expected that an optimized design and proper floorplanning will push the speed even for the FPGA-based design into the speed range of slower ASIC designs (155 MHz).

When targetting usual 0.18μm ASIC technologies a speed-up of 100 to 200 percent compared to the FPGA-based solution can be expected; this is not only an educated guess among chip designers but also demonstrated by the numbers presented in Table 7.2 which show a speed-up range from 1.73 to 2.74 equalling 73 to 174 percent; this will give an estimated clock frequency of about 300 to 450 MHz for the ASIC solution.

With fully custom chip design situation is even better considering that the main timing impact comes from signal routing following given signal as dictated by the used FPGA technology. Since also the logic circuitry can be tailor-made and does not need to be mapped to present logic cells, a speed-up by a factor of 4 as compared to the FPGA technology is possible. Based on the used fabrication technology the estimated core logic speed for the CRYPTONITE architecture can reach 400 MHz to 600 MHz.

The effective operating frequency will, however, be dictated by memory access time. Using aforementioned processes, 400 MHz access rate is possible, but achieving higher frequencies is not likely.

6.3 Summary

Within Section 6.1 of this chapter it was shown that the proposed CRYPTONITE architecture is able to perform the selected crypto algorithms at sufficient throughput.

The single algorithms were listed with complete cycle counts and additional explanations about the numbers where applicable; also, if an implementation makes assumptions about special units and configurations, this was remarked.

Within this paper, a clock rate of 400 MHz was proposed. The reason for doing so was given in Chapter 5. In Section 6.2 it was proven that this rate indeed is possible when using custom chip fabrication; the presented numbers are promising enough that even with current ASIC technologies the postulated 400 MHz can be reached.

This speed estimation was done by realizing selected parts of the architecture as VHDL modules, synthesizing and fitting them into the Xilinx Virtex-IIpro FPGA family and creating a timing analysis based on routing information and logic consumption as provided through the Synopsys development suite and the technology-specific programs provided by Xilinx.

These tests have proven the suitability of the CRYPTONITE architecture which will now be compared against existing hardware cores and programmable solutions in Chapter 7.

7

Comparison of CRYPTONITE against Existing Solutions

Comparing this architecture against existing solutions is not an easy task. First of all, almost no programmable solutions for solving cryptographic tasks exist. Most of the existing solutions are hard-coded, usually not even parametrizable which makes CRYPTONITE quite unique compared to its potential competitors.

Another obstacle are the numbers given by chip vendors. Hardly any company provides detailed performance data nor an insight into the architecture itself. Instead, an overall peak pipeline performance is given but no further information about the chip's architecture, especially its pipeline depth.

In some cases, these numbers only reflect the plain encryption performance and do not involve round key generation. Similarly, sometimes only numbers for the encryption process are given which makes comparison hard if not impossible: Chapters 2 and 3 have shown that round key generation is usually the by far more complex task compared to encryption itself as the IDEA example demonstrates; also, the fast AES implementation [45] indicates that decryption can be more complicated leading to slower performance. A comparison of a monolithic routine incorporating both directions and the round key generation against lightweighted solutions is quite difficult and easily draws a distorted picture.

Despite these problems, this chapter will try to compare the numbers for the CRYPTONITE architecture as presented in Chapter 6 against existing solutions and concepts. Based on this comparison a proper positioning of

the proposed architecture within the field of crypto cores and programmable solutions will be worked out.

7.1 Hardware Solutions & IP Cores

In the very beginning crypto algorithms were designed to easily fit into the respective time's technology. A good example for this is the DES algorithm which was developed to match 1970's technology. Also, in later times special hardware was designed to perform crypto operations fast and effectively. Although general purpose processors have evolved and today easily outperform yesterday's crypto hardware, dedicated hardware solutions are still far from being obsolete as dedicated hardware solutions naturally are faster than any general purpose processor solution.

For this comparison, a number of crypto cores was selected based on the data sheets provided by the respective companies. Selection reasons were speed, availability and technology numbers. As a side effect, the collected data also allows a comparison between cutting-edge hardware solutions like those offered by Amphion and SecuCore and established products like those included in the Broadcom and Hifn portfolios.

7.1.1 Amphion Semiconductor Ltd.

Amphion [9] is employed in a wide range of communication applications like speech encoding, video/imaging, channel coding and signal processing. One of their working fields also covers cryptography where Amphion has developed a set of AES encryption and decryption cores which are applicable to a wide range of fabrication technologies.

Fortunately, Amphion is one of the rare companies who publish straight numbers allowing not even performance but also technology comparison with CRYPTONITE. These numbers are presented in Table 7.2 which shows the hardware requirements of the various cores referring to selected target architectures. Table 7.1 compares the resulting throughput based on synthesis technology and core architecture.

Device	Key Size	Cycles per Op.	Data Rate in Mbits/s			
			ASIC	APEX	Virtex-E	Virtex-II
Encryption Devices						
CS5210	128-bit	44	581	226	275	341
	192-bit	52	492	191	233	289
	256-bit	60	426	166	202	250
CS5220	128-bit	44	581	305	294	350
CS5230	128-bit	11	2327	999	1061	1323
CS5240	128-bit	1	25600	- n/a -	9882	10880
Decryption Devices						
CS5250	128-bit	44	581	215	246	426
	192-bit	52	492	182	208	369
	256-bit	60	426	158	181	290
CS5260	128-bit	44	581	233	264	290
CS5270	128-bit	11	2327	727	896	1064
CS5280	128-bit	1	25600	- n/a -	8704	9344

Table 7.1: CS52xx AES Core Performance

7.1.1.1 CS5210/40 High Performance AES Encryption Cores

The CS5210/40 family of AES encryption cores [10] are targeted towards a wide range of needs: Only the CS5210 is fully configurable and allows processing of any plaintext size together with any key size as defined within the AES specification, where the CS5220 is a shrunk down version of the CS5210 which supports only 128-bit key sizes but shares the same basic architecture. Plaintext data size is limited to 128 bits only.

CS5230 and CS5240 target high-performance areas and implement AES using pipelining techniques which boost the throughput by a factor of 4 (CS5230) and 44 (CS5240) achieving up to 25.6 GBit/s encryption rate compared to 581 MBit/s for the standard implementation as incorporated within CS5210 and CS5220.

These encryption cores have a basic interface which enables feeding in plaintext and key data on separate buses; similarly, the ciphertext output is available from an individual bus. The current chip status is presented through distinct status lines which signal if the chip is ready to accept new

Device	**TSMC 0.18μ**		**Altera APEX-20K**			**Xilinx Virtex-E**			**Xilinx Virtex-II**		
	Logic Gates	Clock Speed	Logic Elem.	ESBs	Clock Speed	Slices	Block RAM	Clock Speed	Slices	Block RAM	Clock Speed
CS5210	18.2K	200	1452	8	77.8	696	4	94.7	696	4	117.3
CS5220	14.8K	200	869	8	105	421	4	101	403	4	102.2
CS5230	27.0K	200	1167	20	85.9	573	10	91.2	573	10	113.7
CS5240	203.0K	200		– n/a –		2397	100	77.2	2181	100	85
CS5250	19.2K	200	1560	8	74.1	745	4	84.7	746	4	100
CS5260	16.4K	200	1176	11	80.4	549	4	91	549	4	100
CS5270	34K	200	1481	20	62.5	778	10	77	778	10	91.5
CS5280	283K	200		– n/a –		4626	100	68	3998	100	73

Table 7.2: Technology and Performance Comparison for Amphion Cores

plaintext or key data, if the encryption process is still running, or if the ciphertext data is valid.

7.1.1.2 CS5250/80 High Performance AES Decryption Cores

The CS5250/80 family of decryption cores [11] is the counterpart to the aforementioned encryption core family and incorporates the same basic design. However, compared to encryption the numbers show a decrease of throughput of up to 30% for decryption with FPGAs as target technologies together with an increase in size of up to 93%.

If realized using ASIC technology the throughput remains the same but also an increase in chip size of up to 40% can be monitored.

7.1.2 Broadcom Corporation

Broadcom [25] is specialized in broadband communications and networking of voice, video, and data services. They offer a wide product range of integrated silicon solution targeting network access using various technologies (optical, wireless, xDSL) and also data security. For the security market Broadcom offers a variety of cryptographic and security processors targeted towards high-bandwidth networks and e-commerce. Broadcom's portfolio holds integrated devices including PCI-bus interface which can be attached to any PCI-based computer system but also single-chip solutions with an arbitrary interface allowing the chips to be connected to almost any processor or I/O subsystem using minimal glue logic.

Although Broadcom provides quite informative data sheets [16, 15, 18, 17, 20, 19, 22, 21, 23, 24] these mostly focus on the PCI aspect and do not reveal any architecture information of their crypto cores. Also, the given numbers mostly refer to combined operation like encryption plus digesting like 3DES+MD5 according to data packet sizes as transmitted over network. This information is collected in Table 7.3.

Table 7.4 presents less individual data and compares raw throughput showing the core's theoretical maximum speed against system throughput giving a vague idea about how much speed is lost for network protocol processing. A most interesting number, the sustained performance, is unfortunately given for only one chip, the BCM5802. Since the data sheets claim that BCM5801, BCM5802, BCM5805, and BCM5820 share basically the

same architecture [15, 17, 19, 21], the sustained performance was calculated from the system performance for BCM5801, BCM5805, and BCM5820 based on the relation between system and sustained performance given for BCM5802.

Since only a summary throughput of combined operations like 3DES together with MD5 is given, it is not possible to calculate cycle numbers for individual algorithms which means that these processors cannot be directly compared to the CRYPTONITE architecture.

7.1.3 Corrent Corporation

The Corrent Corporation [36] is specialized in network security solutions for Gigabit-speed encryption in e-commerce (SSL) and virtual private networks (IPsec) applications [37] and gained big respect by earning the Microprocessor Report Analysts' Choice Award for Best Security Processor of 2001 [89, 90] for their CR7020 SSL Security Processor.

However, Corrent was unable to provide the necessary data sheets allowing a deeper insight into the architecture and its capabilities. Based on [89] the CR7020 compares to Broadcom's BCM5840 and Hifn's 8154 but achieves a peak IPsec bandwidth of 3.8 GB/s and up to 5000 RSA operations per second. It supports AES, DES, 3DES and ARC4 encryption together with SHA-1 and MD5 hashing functions and random number generation and can be interfaced to 33 and 66 MHz PCI bus systems. For the lack of proper datasheets this chip – although very interesting – was omitted from further comparison.

7.1.4 Hi/fn, Inc.

Hifn [63] is a supplier of compression, encryption, authentication, and application recognition technologies especially targetting virtual private network and e-commerce markets. They offer a broad range of products of which the security-related processors were selected for comparison.

Much like Broadcom Hifn provides detailed data sheets [53, 54, 55, 56, 57, 58, 59, 60, 61, 62] which concentrate on the PCI aspect but hardly give any insight into the architecture besides the information that the crypto engines use pipelining technologies. Also, the core performance is not given for all architectures but only the network throughput: As for 7814 and 7854

Device	PCI Speed	Config	Core Speed	Throughput/Mbps @ Packet Size/Bytes							
				64	128	256	512	1024	2048	5120	10240
BCM5801	33MHz	–	66	41	n/a	97	127	149	162	n/a	n/a
	66MHz	–	66	43	n/a	112	152	186	205	n/a	n/a
BCM5802	33MHz	–	33	28	n/a	67	89	104	113	n/a	n/a
BCM5805		encrypt	50	44	75	110	145	168	198	198	204
		decrypt	50	75	101	133	162	179	194	201	205
BCM5820	33MHz	inbound	33	26	45	62	84	101	111	n/a	n/a
		outbound	33	45	64	81	98	109	116	n/a	n/a
		inbound	90	55	95	132	178	211	233	n/a	n/a
		outbound	90	77	117	154	191	219	236	n/a	n/a
BCM5820	66MHz	inbound	66	50	88	124	168	201	233	n/a	n/a
		outbound	66	87	119	158	193	218	233	n/a	n/a
		inbound	90	55	97	138	193	240	269	n/a	n/a
		outbound	90	87	119	163	212	248	274	n/a	n/a

Table 7.3: Broadcom BCM58xx performance regarding packet size

Device	Supported Algorithms		Speed	Performance (MBit/s)			
	Crypto	**Hash**	**(MHz)**	ctrraw	system		sustained
BCM5801	DES, 3DES	SHA-1, MD5	33	320	200		*100*
BCM5802	DES, 3DES	SHA-1, MD5	33	150	100		50
BCM5805	DES, 3DES	SHA-1, MD5	33	240	200		*100*
BCM5820	DES, 3DES, ARC4	SHA-1, MD5	33	330	290		*145*
BCM5821	DES, 3DES, ARC4	SHA-1, MD5	125	n/a	470	(3DES/SHA1)	n/a
					600	(ARC4)	n/a
BCM5840	DES, 3DES	SHA-1, MD5	n/a	n/a	2400	(3DES/SHA1)	n/a

Table 7.4: Broadcom BCM58xx performance data

only the vague information that *"AES [has] somewhat greater performance than 3DES"* [56] is given, for 7851, 8x65 and 8154 no core performance numbers at all are provided.

For this reason, the core performance data as provided through Hifn's data sheets is listed in Table 7.5. From these given values rough cycle count numbers as shown in Table 7.6 were calculated to allow further comparison using the formula

$$cycles = \frac{(CoreSpeed/MHz)*(DataChunkSize/Bits)}{Performance/MBit/s}$$

Although these numbers might not reflect the exact cycle count they are a good base for architecture comparison. It must be noted, however, that the numbers for AES (where applicable) assume 128-bit data and key size (AES 128/128) and are based on the 3DES throughput numbers since the data sheets claim the AES performance to be *"somewhat greater than 3DES"* [56]. These numbers have to be taken with caution.

No numbers were calculated for RC4 since neither key nor data size were given; similarly the 80xx/81xx family of Secure Session Processors was omitted from this list as no detailed data was provided.

7.1.5 NetOctave, Inc.

NetOctave [84] is specialized in building security processors and security accelarator boards targeting Secure Sockets Layer (SSL), IP Security (IPSec) and IP Storage markets. With their NSP2000 [79, 80] and NSP3200 [82, 81] NetOctave provides dedicated PCI-enabled crypto processors tailored towards direct network attachment to support SSL (NSP2000 [80]) and IPsec (NSP3000 [81]) secured connections. The NSP series of processors was presented to a wider audience on the Communications Design Conference (CDC) 2001 through [99] and [100].

7.1.5.1 Performance Data

Also NetOctave does not provide sufficient data for an architectural comparison, only maximum numbers for network throughput are given. which reaches up to 4.8 GBit/s for the NSP3200 Security Processor [82] with 650 MBit/s for combined 3DES/SHA-1 or 3DES/MD5 running on a NSP300x IPsec Security Accelerator Board [81]. For the NSP200x SSL Security Accelerator Board [80] only the core performance of 650 MBit/s (NSP2000/2)

Processor	Core Speed	Performance (MBit/s) AES	DES	3DES	RC4	SHA-1	MD5
Encryption Processors							
7711	33	–	245	82	129	84	100
7751	33	–	164	83	122	80	96
Network Security Processors							
7811	90	–	n/a	252	200	301	376
7814	40	"AES somewhat greater performance than 3DES"					
7851	100	n/a (no AES support)					
7854	100	"AES somewhat greater performance than 3DES"					
7901	50	–	143	53	78	50	60
7902	50	–	143	53	78	50	60
	66	–	188	70	103	66	79
7951	66	–	143	70	103	66	79
Secure Session Processors							
8065/8165	66	AES performance similar to DES					
8154	100	AES performance similar to DES					

Table 7.5: Performance Data of Hifn Processors

Algorithm	Cycle Count for 7711	7751	78xx	79xx
AES	n/a	n/a	46	n/a
DES	9	13	n/a	22
3DES	26	25	23	61
SHA-1	201	211	153	512
MD5	169	176	123	427

Table 7.6: Algorithm cycle counts computed from given performance data

Product	Supported Algorithms Crypto	Hash	Performance Core	Network
Data provided by NetOctave				
NSP2000B	3DES, ARC4	MD5, SHA1	650MBit/s	n/a
NSP2002B	3DES, ARC4	MD5, SHA1	650MBit/s	n/a
NSP2004B	3DES, ARC4	MD5, SHA1	1300MBit/s	n/a
NSP3000B	3DES	MD5, SHA1	650MBit/s	n/a
NSP3200	AES, 3DES	MD5, SHA1	n/a	4.8GBit/s
Data provided by Chipcenter				
NSP2000	3DES		1000MBit/s	n/a
@100MHz		MD5	1190MBit/s	n/a
		SHA	2000MBit/s	n/a

Table 7.7: Performance Data for NetOctave SSL and IPsec Processors

and 1.3 GBit/s (NSP2004) are given. Chipcenter [79] gives more detailed but different numbers which have also been added into Table 7.7.

Although a white paper about NetOctave's *FlowThrough Security Architecture* [83] is provided, it does not give a detailed description of the architecture but contains only basic information about the various modules integrated into one chip.

Similar to Broadcom, NetOctave only provides numbers for combined operation which do not allow to calculate approximate cycle counts for individual operations as needed for comparison with the CRYPTONITE architecture.

7.1.6 OpenCores.org

OpenCores [86] follows the same ideology as the Open Software Foundation (OSF) [85]. Where OSF provides free software, OpenCores focuses on free hardware cores. Through their pages, OpenCores offers a wide range of Verilog and VHDL hardware models in different development stages; from the crypto cores section only the DES core [128] is readily developed and tested on various hardware achitectures. In addition, the IDEA [3] core is developed far enough to present at least minimal numbers. Currently under development are AES/Rijndael [136], RC4 [134], Twofish [5] and

Hardware Platform	Size	Speed (MHz) Core	Processing
Area Optimized			
0.18μm UMC ASIC	3k Gates	155	9.68
Altera APEX 20K	1106 Icells	27	1.68
Altera FLEX 10K50	1283 Icells	43	2.68
Performance Optimized			
0.18μm UMC ASIC	28k Gates	290	18.13
Altera APEX 20K	6688 Icells	53	3.31
Altera FLEX 10K50	6485 Icells	76	4.75

Table 7.8: OpenCores DES Performance Data

SHA-256 [135] cores as well as a monolithic crypto accelerator supporting AES/Rijndael, Twofish, and SHA-1 [120]. Since these cores are mostly still in planning stage they are listed for completeness but will not be further investigated.

7.1.6.1 OpenCores DES Core

The DES core has already been implemented and tested on multiple hardware platforms. It takes 16 cycles to perform a full encryption or decryption which implies that each round takes one cycle, input/output permutation are embedded into the data path. The performance data is shown in Table 7.8. For this core the bits-per-cycle ratio equals 4.

7.1.6.2 OpenCores IDEA Core

Although announced for quite some time, this core still is not finished. However, it has already been implemented in hardware and is said to achieve a throughput of 177 MBit/s, further information about the core speed and the needed cycles per IDEA encryption or decryption is omitted. The given diagrams are no help since they are somewhat contradictory: The core's block diagram seems to receive key and data 16-bit-wise, similarly the cipher text is written out in items of 16-bit. The IDEA Machine, however, as main

Core	Throughput (GBit/s)	Cycles	Data Chunk Size	Bits per Cycle
AES-128	1.94	11	128	11.6
DES	2.	> 5	64	12.04
3DES	0.67	< 16	64	4.01
MD5	1.25	68	512	7.53
SHA-1	1.01	84	512	6.10
SHA-256	1.25	68	512	7.53

Table 7.9: SecuCore Performance Data

functional unit seems to perform the complete IDEA operation within one cycle.

Since no additional documentation is given, this core is omitted from further comparison due to incomplete and contradictory data.

7.1.7 Secucore Consulting Services

SecuCore [116] is a provider of high performance hardware core models targeting data security. They offer a broad variety of cores for different crypto algorithms, namely AES/Rijndael [112], DES and 3DES [113], MD5 [114], and SHA-256 [115] which deliver high performance using a 0.18μm ASIC technology.

7.1.7.1 Performance Overview

According to the data sheets, SecuCore used TMSC's 0.18μm ASIC technology running at 166 MHz to benchmark their cores. Also, most of the data sheets provide the number of cycles needed to perform the crypto operation on the algorithm's specific data chunk size. For DES and 3DES, however, these values were not provided and had to be calculated from throughput, core speed and data size.

7.1.8 Zyfer, Inc.

Zyfer's [140] main area of operation is precision synchronization devices as used for communication, networks and military applications with focus on

secure and robust data transfer through ground and satellite links. One of their products is the SKP-100 Gigabit Network Security Processor [141]

7.1.8.1 SKP-100 Gigabit Network Security Processor

The SKP-100 is currently under development, so Zyfer publishes only a short product information presenting minimal facts. It supports AES/Rijndael-128/128, 3DES in 2- and 3-key configuration and plain DES, all in EBC and CBC mode. For authentication, MD5, SHA-1, and SHA-256 hash algorithms are supported.

For 3DES and Rijndael the product information claims that full duplex OC-48 speed (2.4 GBit/s) is achieved, but since this number is also given as "pipe speed" it can be safely assumed that that these algorithms are not implemented single-cycle.

Since further insight into the architecture like pipe length or number and structure of processing units are given, no comparative values can be calculated.

7.2 Programmable Solutions

Fully programmable crypto processors are still a novel concept: Until now almost nobody has entered this field and so far only one fully programmable and algorithm independent machine, the CryptoManiac [131], is known. Besides, a microprogrammable processor targeted towards IDEA exists, the PLD001 Cryptoprocessor [67]. Although it was developed with respect to IDEA it is not clearly IDEA specific and can be considered as another programmable solution.

7.2.1 CryptoManiac

CryptoManiac appeared in mid 2001 and is a 4-wide 32-bit VLIW architecture. It basically consists of 4 functional units operating on a commonly used data memory which are part of a 4-stage pipeline. To keep this pipeline filled a 16-entry branch target buffer is used for branch prediction. Figure 7.1 gives an overview over the CryptoManiac architecture.

CryptoManiac explicitly supports communication with a host processor through a queue mechanism providing input and output data queues. A request scheduler is responsible for distributing incoming requests in order of appearance. It also offers a special keystore interface to hold key data and substitution tables.

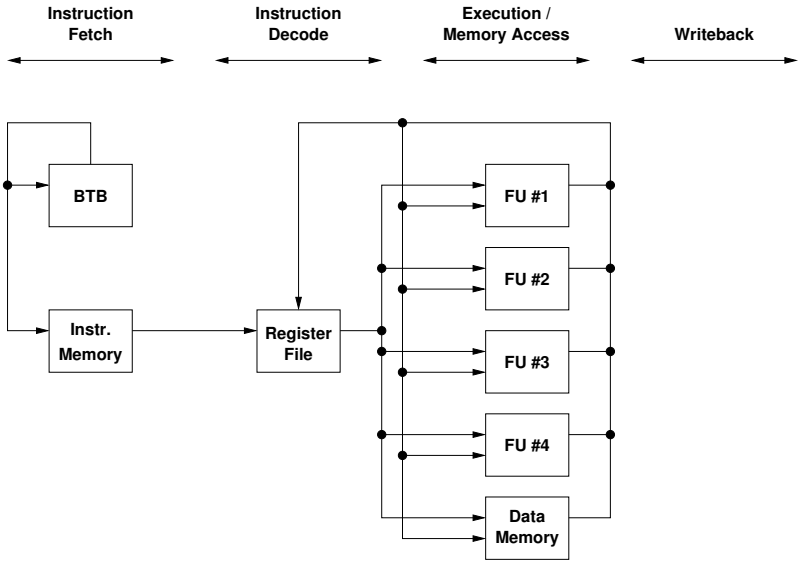

Figure 7.1: CryptoManiac Architecture Overview

The heart of the CryptoManiac architecture is the four functional units as depicted in Figure 7.2 which are less complex than CRYPTONITE's arithmetic unit, but are able to perform up to three sequential operations within one cycle.

7.2.1.1 Conceptual differences to CRYPTONITE

Although there are some similarities between CRYPTONITE and CryptoManiac concerning the concept of multiple functional units or VLIW, the architectures are quite different at a closer look.

Firstly, CRYPTONITE is a single-cycle execution architecture. It does not allow multi-cycle operations like CryptoManiac with its instruction-combining model. The reason to do so was mainly for speed reasons but

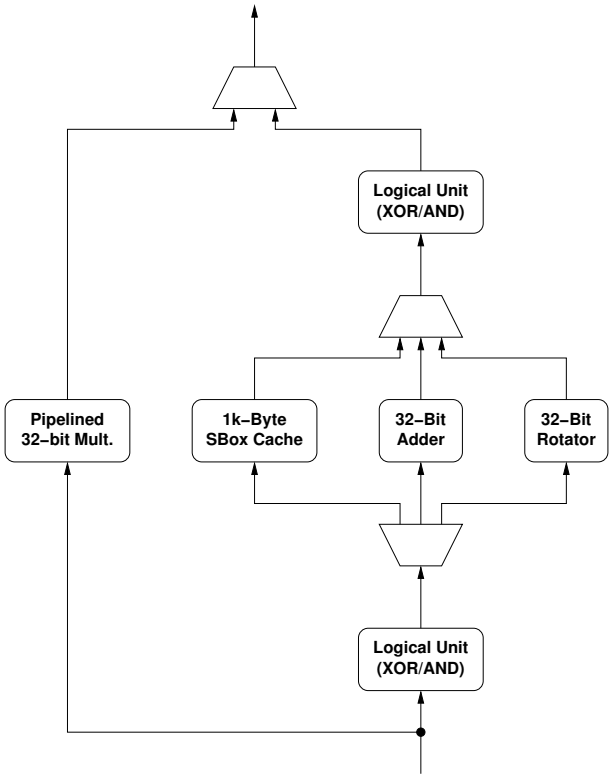

Figure 7.2: CryptoManiac's functional unit

to a smaller degree also to ease compiler design, namely resource allocation. The drawback of single-cycle execution is that it puts great pressure on the hardware design to ensure that all functions can be performed within one cycle.

On the level of functional units data access and data processing are strictly separated[1] in CRYPTONITE. All data like input data, keys, tables, or S-Boxes is held in memory, as opposite to CryptoManiac's functional units which employ S-Box caches. On the other hand, CRYPTONITE's arithmetic units are more powerful and allow quite complex monolithic operations to support the needs of advanced crypto algorithms.

[1] with the exception of the DES unit which is placed in the memory unit as explained in Section 5.8.4

	Alpha 21164	CRYPTO-NITE	**CryptoManiac**			
			4WC	**3WC**	**2WC**	**4WNC**
Blowfish	9.58	n/a	4	4	6	5
3DES	23.56	2/3	7	8	9	12
IDEA	91.95	11	14	14	17	15
Mars	28.86	n/a	9	9	9	10
RC4	11.49	n/a	8	8	8	9
RC6	23.24	7+3	7	7	7	9
AES/Rijndael	33.84	8/7	9	11	17	10
Twofish	27.36	n/a	7	8	11	8

Table 7.10: Kernel loop cycle counts per round

Another difference shows up together with memory access: Where CryptoManiac allows all four functional units to access a common data memory, CRYPTONITE has separate memory units for both ALUs. Communication with external units is handled through request pipelines within CryptoManiac; as of now, the CRYPTONITE architecture only provides a raw interrupt-based communication interface allowing external units to put CRYPTONITE on hold while accessing the internal memory to update or read out values. This means that an external control processor handling the network data stream has to frequently update CRYPTONITE's data memory to provide new and read out processed data.

7.2.1.2 Performance Data

For CryptoManiac quite a number of evaluations were done covering computation speed, power dissipation, timing results, and on-chip area requirements. The speed evaluation was compared against a modern general purpose processor, the Alpha 21264; also, the numbers for the CRYPTONITE architecture were added where applicable. The results of these comparisons are shown in Table 7.10 for performance data and Table 7.11 for physical parameters.

Unfortunately, the cycle count is only given for an algorithm's round and not for the entire encryption process of a data block which makes throughput calculation a bit problematic. Based on Table 6.1 the ratio between round

	CryptoManiac			
	4W Comp.	**3W Comb.**	**2W Comb.**	**4W N-Comb.**
Chip Speed	2.78ns	2.66ns	2.54ns	2.75ns
	360MHz	376MHz	394MHz	364MHz
Chip Size	1.93mm^2	1.77mm^2	1.59mm^2	1.69mm^2
Power Consumption	606.37mW	593.51mW	568.50mW	586.86mW

Table 7.11: Timing and Area Results for CryptoManiac

cycle count and total cycle count was calculated; these ratios serve as a multiplication factor for the CryptoManiac kernel loop counts as provided through Table 7.10. Table 7.12 shows the comparative numbers based on this calculation.

These numbers have to be taken with extreme caution and are most likely too optimistic since [131] does not mention round key generation at all. As for CRYPTONITE code AES round key generation is responsible for about 30% of the cycle count; for a (3)DES implementation using single-operation instructions (as opposed to the monolithic instructions employed within CRYPTONITE) round key generation causes about 16% of the overall cycle count. To take this into account, corrected values were calculated for 3DES and AES by multiplying the ratio factor by 1.16 and 1.30 respectively.

It must be further noticed that only the CryptoManiac's 4WC configuration running at 360 MHz was considered.

Algorithm	**Round Cycles**		**Ratio**		**Total Cycles**	**Throughput Bits/Cycle**	**MBit/s**
3DES	7	*	48	=	336	0.19	68
3DES corr.	7	*	56	=	392	0.16	59
IDEA	14	*	8.18	=	115	1.11	400
RC6	7	*	20.6	=	144	0.89	320
AES-128/128	9	*	10	=	90	1.42	511
AES-128/128 corr.	9	*	13	=	130	0.98	353

Table 7.12: Estimated Throughput for CryptoManiac running at 360 MHz

7.2.2 PLD001 Cryptoprocessor

A crypto processor designed towards IDEA and RSA was developed by Jüri Pöldre during 1994 to 1997 [67]. Like this work it followed a very similar approach by analyzing distinct algorithms and tayloring an architecture towards these algorithms. The PLD001 is a microprogrammable architecture allowing different ALU configurations as needed for certain processing steps within IDEA computation.

The analysis of RSA and IDEA led to a quad-partite ALU as shown in Figure 7.3 consisting of four 24-bit wide units. For IDEA these units are configured as two 16-bit multipliers. Since each 24-bit block can perform an $8 * 16$-bit multiplication, two such multipliers do the needed $16 * 16$-bit multiplication in one cycle. If long modular calculation is needed, the ALU can be configured as an $8 * 96$-bit multiplier or 96-bit adder/negator with additional carry logic as shown in Figure 7.4. This configuration is called *calculation mode*. The complete data path within PLD001's ALU is depicted in Figure 7.5.

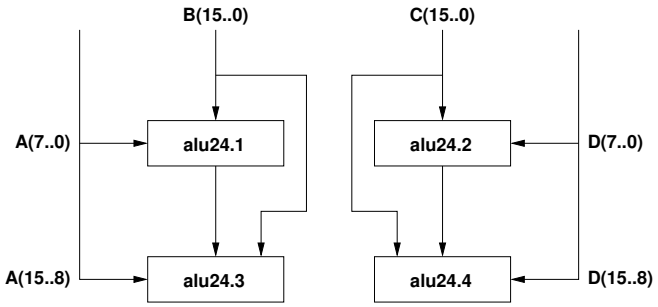

Figure 7.3: IDEA configuration

Besides ALU configuration, PLD001 microprograms allow access to the internal memory and selection of desired logical and arithmetic operations. Memory access is controlled by the so-called ALU control structure as shown in Figure 7.6. The `INDEX_CALC` unit is capable of either addressing 6-bit entities inside the 768-bit data registers or loading parts of the 768-bit memory into the 24-bit register `ER`. Logic operations between between immediate values, index register, and `ER` may apply. Arith-

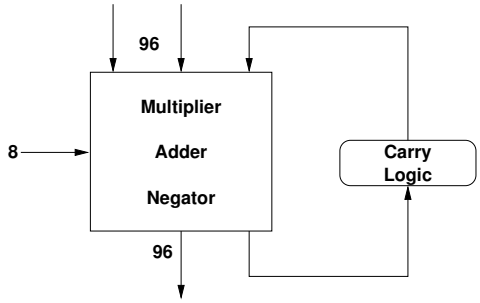

Figure 7.4: Calculation Mode

metic commands only operate on long-data registers and are executed in the ALU_SEQUENCER unit in parallel with index calculation.

PLD001 also supports so-called high-level commands which are basically calls to microcode programs allowing reusable complex operations. 32 of these are fall under the category *external commands* which means that these can be directly selected by applying the proper bit pattern to dedicated input pins.

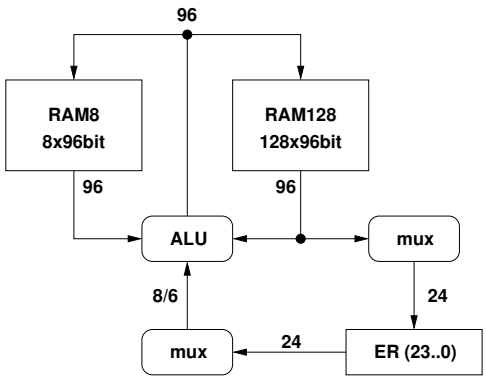

Figure 7.5: Data flow within the PLD001 ALU

7.2.2.1 Performance

[67] gives mainly numbers for IDEA calculation. According to this document it takes 50 clock cycles per IDEA transformation equalling 32 MBit/s

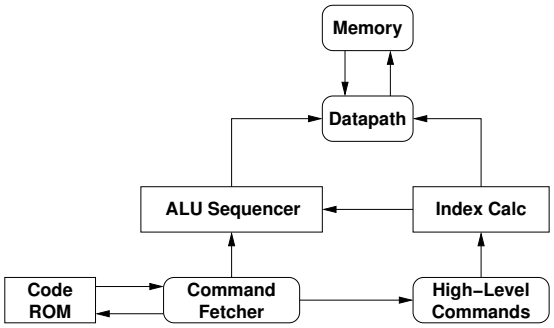

Figure 7.6: PLD001 ALU control structure

at the denoted 25 MHz operating speed. RSA key exchange is said to take about 0.5s which equals 2 key exchanges per second. The values are summed up in Table 7.13.

Supported Algorithms	IDEA and RSA Key Exchange
Core Speed	25 MHz (tested)
	20 MHz (calculated)
Technology	ES2 1.0μm and Xilinx FPGAs
Throughput	32MBit/s for IDEA
	2 key exchanges per second for RSA

Table 7.13: PLD001 Performance Data

7.3 Algorithm Performance Comparison

Within the last two sections many dedicated hardware solutions and two programmable solutions were presented. A comparation, however, is not an easy task since the various sources present quite different data. As certain parameters had to be estimated or calculated based on estimations, some values have to be taken with care but nevertheless allow a performance comparison between the CRYPTONITE architecture and its competitors.

7.3.1 AES-128/128 Performance

Table 7.14 shows performance data for those crypto solutions supporting AES-128/128. The Amphion CS5240 was omitted from this list since it is a very special single-cycle solution achieving a throughput of 25600 MBit/s for the sake of an over 10-times bigger gate count. Figure 7.7 compiles this data into a more pictoral form.

Architecture	**Cycle Count**	**Speed** (MHz)	**Throughput** (MBit/s)
Amphion CS5220	44	200	581
Amphion CS5230	11	200	2327
Hifn 7854	46	100	280
SecuCore AES-128	11	166	1931
CryptoManiac (raw)	90	360	511
CryptoManiac (corr.)	130	360	353
CRYPTONITE ($\leftrightarrow$)	80	400	640
CRYPTONITE ($\rightarrow$)	70	400	731

Table 7.14: AES-128/128 Performance Comparison

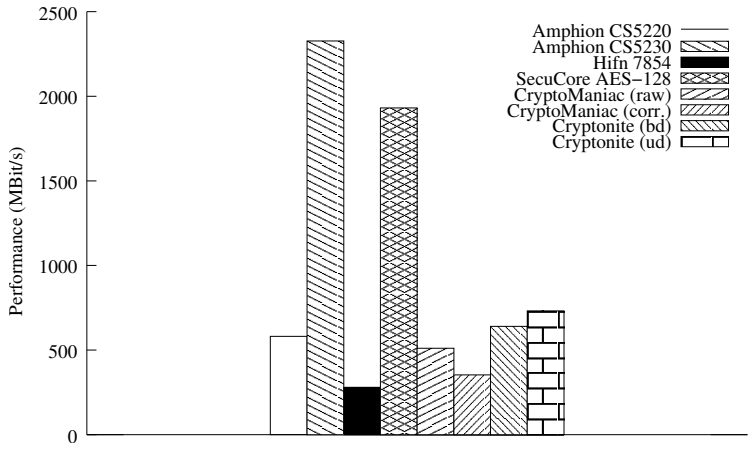

Figure 7.7: AES-128/128 Performance Comparison

Besides CryptoManiac, the 44- and 11-cycle versions from Amphion (CS5220, CS5230), the 46-cycle version from Hifn (7854) and another 11-cycle version from SecuCore (AES-128) which runs at slightly slower speed than the Amphion CS5230 are included which are positioned against the bidirectional ($\leftrightarrow$) and unidirectional ($\rightarrow$) versions of CRYPTONITE's fast AES implementation [45].

It can be observed, that CRYPTONITE running at 400 MHz outperforms the 44/46 cycle hardware implementations by a factor of 1.25 to 2.6; as expected, the 11-cycle versions outperform CRYPTONITE by a factor greater than 2.64, but it must been taken into account that these cores are extremely tailored towards the AES/Rijndael algorithm, which is especially true for the single-cycle solution CS5240. Compared against CryptoManiac CRYPTONITE shows an up to two times better performance.

7.3.2 DES Performance

As a result of the dedicated DES unit CRYPTONITE shows excellent crypto performance as listed in Table 7.15. It outperforms most of the cores by a factor of 1.18 to 5.12; only SecuCore DES shows again overwhelming performance and beats CRYPTONITE by a factor of 2.73. This data is graphically presented in Figure 7.8.

Architecture	**Cycle Count**	**Speed** (MHz)	**Throughput** (MBit/s)
Hifn 7711	9	33	245
Hifn 7751	13	33	164
Hifn 790x	22	50	143
OpenCores DES	16	155	620
SecuCore DES	5	166	1999
CRYPTONITE	35	400	732

Table 7.15: DES Performance Comparison

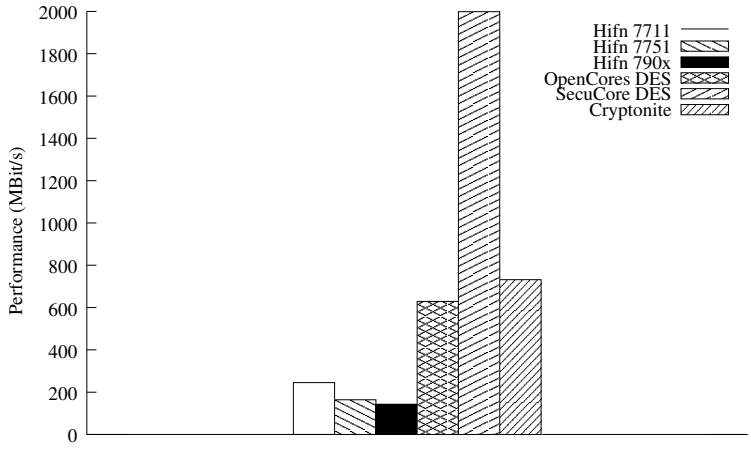

Figure 7.8: DES Performance Comparison

7.3.3 3DES Performance

CRYPTONITE's great DES performance also shows up for 3DES where it outruns its hardware competitors by a factor up to 3.13; only the Hifn 7811 shows slighty better performance and reaches 103% of CRYPTONITE's throughput as listed in Table 7.16 and depicted in Figure 7.9. The comparison against the programmable competitor CryptoManiac shows the great benefit of CRYPTONITE's DES unit. Thanks to this unit CRYPTONITE outperforms CryptoManiac by factors up to 4.14.

Architecture	**Cycle Count**	**Speed** (MHz)	**Throughput** (MBit/s)
Hifn 7711	26	33	82
Hifn 7751	25	33	83
Hifn 7811	25	90	252
Hifn 790x	61	50	78
CryptoManiac (raw)	336	360	68
CryptoManiac (corr.)	392	360	59
CRYPTONITE	105	400	244

Table 7.16: 3DES Performance Comparison

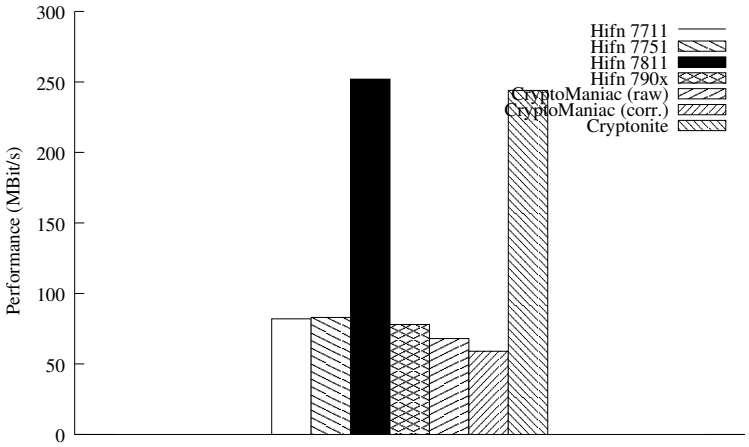

Figure 7.9: 3DES Performance Comparison

7.3.4 IDEA Performance

Architecture	Cycle Count	Speed (MHz)	Throughput (MBit/s)
CryptoManiac/4WC	115	360	400
PLD001	50	25	64
CRYPTONITE	90	400	569

Table 7.17: IDEA Performance Comparison

Also with IDEA CRYPTONITE shows good performance and outperforms its fully programmable competitor CryptoManiac by a factor of 1.42; even the IDEA-optimized PLD001 is outrun by a factor of 8.89 as shown in Table 7.17 and Figure 7.10.

7.3.5 RC6 Performance

With this algorithm, CryptoManiac is able to defeat CRYPTONITE using its complex functional units which make it possible to perform up to three sequential operations at once. It is not a matter of higher parallelism or

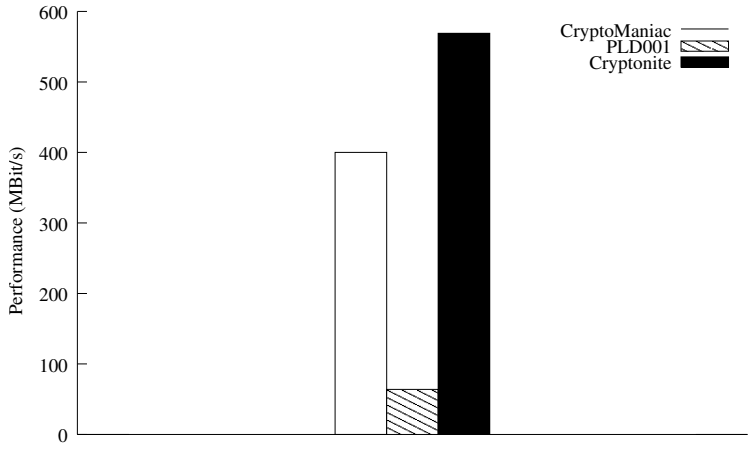

Figure 7.10: IDEA Performance Comparison

common data memory as Table 7.10 shows since the loop cycle count for RC6 is equal for all incarnations of the CryptoManiac architecture.

As shown in Table 7.18 and Figure 7.11 CRYPTONITE delivers about 29% less performance as Cryptomaniac for RC6 in AES configuration.

Architecture	**Cycle Count**	**Speed** (MHz)	**Throughput** (MBit/s)
CryptoManiac/4WC	144	360	320
CRYPTONITE	206	400	249

Table 7.18: RC6 Performance Comparison

7.3.6 MD5 Performance

For MD5, CRYPTONITE again shows very good performance against the competing hardware cores and is able to deliver 1.08 to 6.77 times better throughput as the Hifn cores; as with previous examples it is again defeated by the specialized SecuCore solution which outperforms CRYPTONITE by a factor of 3.08 as shown in Table 7.19 and Figure 7.12.

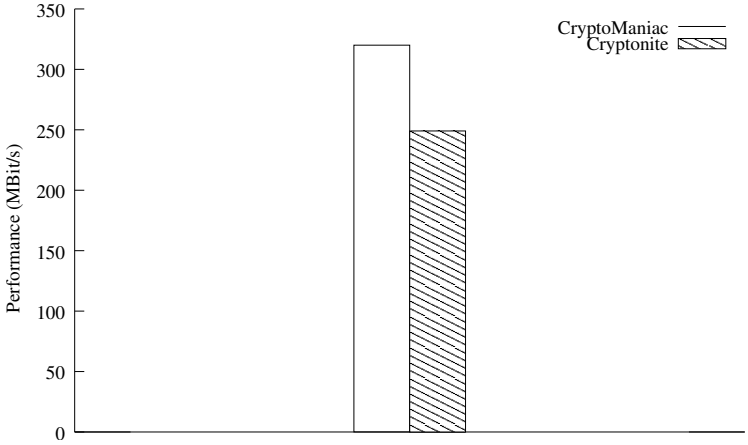

Figure 7.11: RC6 Performance Comparison

Architecture	**Cycle Count**	**Speed** (MHz)	**Throughput** (MBit/s)
Hifn 7711	169	33	100
Hifn 7751	176	33	96
Hifn 7811	123	90	376
Hifn 790x	427	50	60
SecuCore MD5	68	166	1250
CRYPTONITE	504	400	406

Table 7.19: MD5 Performance Comparison

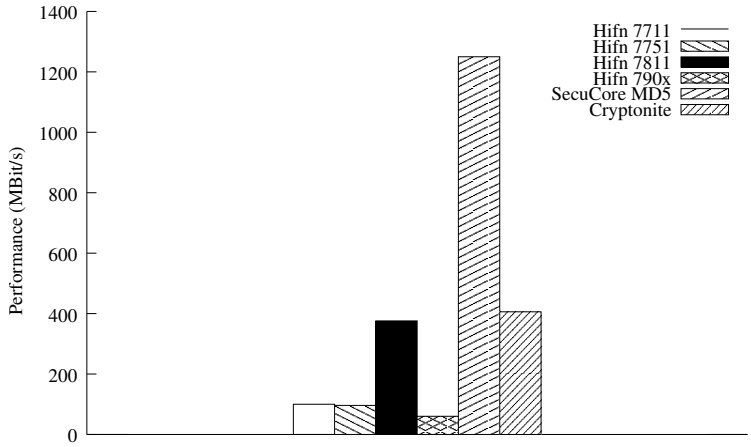

Figure 7.12: MD5 Performance Comparison

7.3.7 SHA-1 Performance

A similar picture is drawn by the SHA-1 performance comparison which shows 1.4 to 8.4 times better throughput for CRYPTONITE compared to the Hifn cores – but again the winning architecture is the dedicated hardware core from SecuCore defeating CRYPTONITE by a factor of 2.41 as shown in Table 7.20 and Figure 7.13.

Architecture	**Cycle Count**	**Speed** (MHz)	**Throughput** (MBit/s)
Hifn 7711	201	33	84
Hifn 7751	211	33	80
Hifn 7811	153	90	301
Hifn 790x	512	50	50
SecuCore SHA-1	84	166	1013
CRYPTONITE	488	400	420

Table 7.20: SHA-1 Performance Comparison

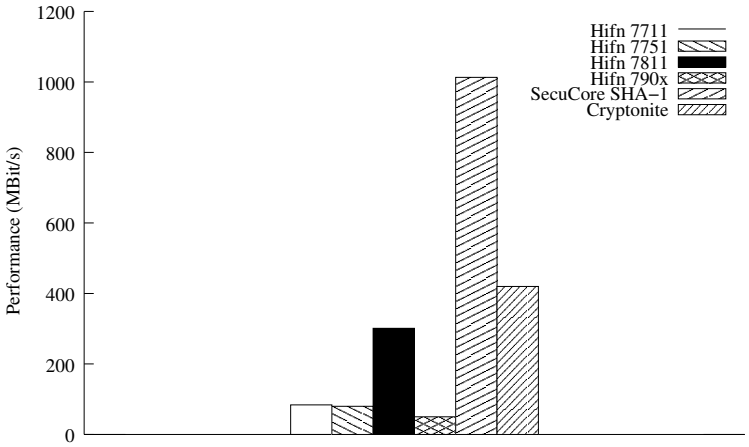

Figure 7.13: SHA-1 Performance Comparison

7.4 Summary

Within this chapter alternative crypto processors – both dedicated hardware and programmable solutions – were presented. In those cases where appropriate data was presented by the respective vendor, it was compared against the CRYPTONITE architecture; in some cases numbers had to be calculated from the given data and certain assumptions made. This was mentioned in the respective sections.

The comparison has clearly shown, that a modern programmable solution does not necessarily need to be slower than typical dedicated hardware solutions. In most cases CRYPTONITE outperformed the competing hardware cores by factors up to 5.42; even against high-performance cores like some of the Amphion solutions for AES-128/128 and all SecuCore cores CRYPTONITE was able to produce quite respectable results: The tradeoff for a fully programmable solution with as little special units or instructions as possible is a factor of less than 2.7 averaged over all cases where CRYPTONITE was defeated by hardware hardware solutions.

In addition the comparison especially with the CryptoManiac architecture has proven the algorithm analysis of Chapter 3 to be correct; also the architectural impact and proposals as elaborated in Chapter 4 resulting in the CRYPTONITE architecture presented in Chapter 5 were justified. The

CRYPTONITE architecture was proven to be efficient and competitive, both with programmable and dedicated hardware solutions.

8

Conclusion and Summary

Within this work a novel, fully programmable processor architecture focused but not limited to cryptographic algorithms was presented.

In Chapter 1 the need for such an architecture was demonstrated together with some real-life examples. High-performance cryptography is pervading all fields of digital data distribution where broadcasted content has to be secured against unauthorized access. Of special interest is the field of high-bandwidth data distribution as necessary for all kinds of real-time data streams like video-on-demand (or, more generally, multimedia-on-demand) services, secured video conferencing or spanning of virtual private networks (VPNs) over existing network infrastructures.

Chapter 2 gives a general introduction into the field of cryptography together with basic definitions. It shows the difference between symmetric and asymmetric algorithms and explains typical operation modes of cryptographic algorithms.

For Chapter 3 a set of algorithms was selected and analyzed. The selection criteria was current and future proliferation; focus has clearly been put on the old crypto standard DES and the recently defined new standard, AES. Additional crypto and hash algorithms were added to investigate additional requirements by algorithms different from AES and DES.

This analysis' results and their impact on the proposed architecture have been listed in Chapter 4. Based on the algorithm analysis, needed operations were collected and weighted. The operations were distributed on functional units, namely Arithmetic Unit (AU) and XOR Unit (XU) both being part of the Arithmetic-Logical Unit (ALU). An accompaning Memory Unit (MU)

enables the ALU to store its results to data memory or load previously stored data into registers of the register file. To exploit parallel structures found in certain algorithms this compound of ALU and MU called *strand* is doubled. An interlink mechanism provides a possibility to forward data and results between both strands.

Also, number and size of registers as well as supported data types were defined. For the analyzed algorithms a file of 4 registers fulfills the requirements if the already proposed interlink mechanism is extended by means of register value forwarding in addition to ALU results and memory data. As data types, 32-bit and 64-bit were defined with the possibility to individually access the two 32-bit halves of a 64-bit register allowing a 64-bit register to be used as two independent 32-bit registers. For data memory accesses this scheme is maintained and extended by the feature of so-called S-Box access as necessary for many crypto algorithms employing non-linear functions based on certain data tables; for an S-Box access the 64-bit of a memory cell are distributed evenly over eight 8-bit quantities each being addresses separately using a base address and an 8-bit offset provided through an similarly divided 64-bit index register. These eight 8-bit quantities are combined to a 64-bit result.

Furthermore, the need for special instructions was investigated. Due to its hardware-oriented nature DES clearly needs a specialized unit incorporating the necessary functions for the DES-specific operations, which are fixed bit permutations, compression and expansion operations together with round key generation. Also other algorithms could certainly benefit from specialized, monolithic instructions; the decision was made to only support AES with further instructions as needed for the fast AES/Rijndael implementation as provided by Agere Systems [45]. A reconfigurable modulo-multiplier as needed by IDEA and RC6, for instance, are recommended but not necessary.

Based on this analysis, an architecture was constructed which is presented in Chapter 5. It consists of a control unit (CU) being responsible for command decoding and program flow control. For this reason, it consists of 16 so-called counter registers (CR) together with a small associated arithmetic unit enabling loop constructs and conditional branching; to also include conditions based on ALU results, the topmost four CRs mirror the sign and zero flags of the last ALU operation for each of the two ALUs.

These ALUs divide into AU, XU and register file (RF) as described above with the ability to allow value and result forwarding between the two ALUs. Special operations as defined for fast AES operation are included into the AU. There is no direct connection from and to data memory; instead, registers of the RF can be loaded from data memory through the MU's Data Output Register (DOR); similarly, ALU results or register values can be stored to data memory through the MU's Data Input Register (DIR). Value transfer to registers is only possible through an ALU's A or X bus as shown in Figure 5.4 which means that during register load operations either AU or XU can not be used and have to be put in an idle state.

The memory unit (MU) is responsible for memory address generation and data transportation and thus divides into Address Generation Unit (AGU) and Data I/O Unit (DIO), the latter also containing the DES unit. The AGU does not only provide direct and S-Box addressing using but also indexed and modulo addressing; these more complex addressing modes are realized through a post-increment / post-modulo circuitry.

This architectural concept was tested using software emulators allowing test runs of algorithms implemented on the CRYPTONITE architecture. For selected algorithms the cycle count of certain algorithms is given in Chapter 4 which also covers the hardware implementation of the proposed architecture. Here, a number of functional units were described using VHDL, synthesized and fitted to the Xilinx [132] Virtex-IIpro FPGA family [133]. Based on these numbers together with other data provided by hardware core vendors speed estimations were done considering ASIC and custom chip layout implementations. This speed estimation forecasts a resulting speed range of 400 to 600 MHz when using custom chip fabrication techniques.

In Chapter 7 the CRYPTONITE architecture is compared against existing crypto processor solutions; these include hardware cores being able to process only one algorithm, or a fixed set of algorithms, and programmable processors. All these solutions are listed together with their performance data. To allow independent comparison, absolute (MBit/s) and relative (Bits/Cycle) numbers were calculated based on the given data where not provided. The comparison against existing solutions has proven CRYPTONITE's high performance; even compared to hardware solutions it performs very well; the performance impact to be paid for programmabilty does usually not exceed a factor of 2.4, which means that in worst case

the proposed programmable solution still offers about 42% of a dedicated hardware solution.

In summary, this work has presented a novel programmable crypto processor architecture. This architecture was carefully designed with respect to high performance at moderate hardware demands. Running at 400 MHz this architecture is able to outperform existing hardware solutions as well as comparable programmable solutions. Newer hardware solutions such as the SecuCore products of course show superior performance in their specific field; but even compared with these the programmable CRYPTONITE architecture is able to provide about 42% of the hardware core's performance.

9

Future Work

The work presented within this document describes the core CRYPTONITE architecture which was developed with respect to speed and flexibility. Associated with this architecture there are further outstanding projects necessary for industrial use of the CRYPTONITE architecture which are listed in the following two sections. The third section covers possible areas of use.

9.1 Software Development

Currently, two simulator frameworks exist. One is based on Standard ML (SML), the other one is a cycle-based emulator written in C making it possible to write programs in an assembly language-like form. Both emulators, however, do not allow true native program development. In addition, the SML solution does not know about the CRYPTONITE architecture thus it neither knows about architectural restrictions nor does it support variable-to-register mapping. Instead, the programmer needs to write programs architecturally mapped as if writing programs in assembly language.

The cycle-based emulator was designed to allow importing and running of CRYPTONITE binaries at a later development stage; until now it provides a framework for processing code in assembly language-like form, basically C files which call internal library routines mimicking assembly language commands.

9.1.1 CRYPTONITE Assembler

For proper program development an assembler is essential and would be the first step in further software development. The assembler will contain source file parser, syntactical and semantical checks being able to detect syntax errors as well as breaches against the architectural rules, and finally code generation.

9.1.2 C-Compiler for the CRYPTONITE Architecture

A C compiler for the CRYPTONITE architecture is an ambitious task since it not only has to translate C to assembly language but also incorporate code analysis to exploit the maximum possible parallelism by assigning code as densely as possibly to the functional units. These tasks resemble the requirements for compilers for EPIC-style processors like the Intel Itanium [66]. Hence, it might be possible to adopt work done for the Trimaran citetrimaran project which originally targeted the HPL-PD architecture citehpl-pd but is flexible enough to be used for similar architectures as already shown through the Triceps [31] implementation, which mapped Trimaran to the ARM architecture.

9.2 Hardware Development

As of now, CRYPTONITE is just an architecture study. Only selected parts of the chip were realized and simulated using VHDL. This work describes the core architecture and only covers a simple interrupt-driven I/O mechanism; more sophisticated interface methods as needed for PCI-bus attachment or inter-processor communication have not yet been targeted. Further work definitely needs to cover these items to allow easy use of CRYPTONITE within existing computer systems.

9.2.1 Single-Core CRYPTONITE

A single-core CRYPTONITE would contain the CRYPTONITE core architecture together with an appropriate interface technology. This can be either a PCI interface to allow attachment of the CRYPTONITE to any PCI-based

systems but also a simple synchronous (Intel x86-style) or asynchronous (Motorola 68k-style) bus interface to ease the use of CRYPTONITE with microcontrollers and non-PCI systems.

9.2.2 Multi-Core CRYPTONITE

Originally, CRYPTONITE was planned to be a SIMD multi-processor system consisting of multiple individual CRYPTONITE cores. Since quite a number of crypto algorithms is used in CBC mode, the SIMD approach is somewhat limited because it would allow only n parallel streams – where n is the number of cores – using the very same crypto algorithm.

A MIMD approach would be more flexible since it allows to process n parallel streams using different algorithms. Such an architecture could contain internal load balancing to allow the assignment of incoming data streams to eventually idle cores or the take-over of lower priorized streams by higher priorized ones by detaching a running encryption from a core and reassigning this freed core.

Such a MIMD architecture contains several interesting challenges which are:

- **Access of Program Data:** Program data should not be present n times individually for each processor core. Instead, it would be more sensible to employ some bootstrap mechanism which loads program routines as needed from a common program memory into individual program caches.

- **Scheduling of Workload:** To allow maximum throughput incoming data streams have to be assigned to idle cores; similarly, cores have to be freed after a stream has ended. The assignment procedure has also to cover loading of the correct program data into the individual core's program cache. Unnecessary loads have to be avoided by re-assigning already initialized cores where possible.

- **Priorizing of Streams:** Besides simple assignment of streams to idle cores, it must also be possible to priorize certain streams over others to allow some sort of quality-of-data service. This not only adds significant complexity to the scheduler but also brings up the problem of

buffering of incoming information as well as inter-core communication and signalling.

9.3 Use of CRYPTONITE

Many dedicated hardware solutions exist but only very few programmable ones which brings up the question, where a programmable solution could be used and if that makes sense – especially, since naturally more sophisticated hardware cores show superior performance to any programmable solutions.

9.3.1 Research & Development

R&D is the obvious candidate. Here a programmable solutions allows to create a very flexible experimental platform; algorithms and configurations can be checked without the need of re-designing hardware. A programmable high-performance solution will provide a realistic test environment.

9.3.2 Conditional Access Systems

The flexibility of a programmable high-performance solution makes it ideal for use in conditional access systems like TV set top boxes. In the past, several of these boxes were hacked in a way that the crypto algorithm was made public and pirate cards or completely independent decryption devices were developed.

In such cases, the access providers usually have to replace the existing conditional access module (CAM, also known as "Smart Card") which holds the decryption algorithm or even the complete set top box. Using a fully programmable high-performance crypto solution it would only require to update the crypto algorithm within the CAM.

9.3.3 VPN Devices

This is a field similar to the above. Usually, VPN devices make use of specific hardware solutions to enable high transfer speed. Once an algorithm becomes obsolete or unwanted because it is considered to be cryptographically weak, or if security demands require periodically change of the crypto

algorithm, any device employing dedicated hardware solutions has to be exchanged or becomes obsolete. A programmable device would allow on-the-fly reconfiguration and save investment.

Following the R&D approach, a programmable high-performance solution can be used for so-called market openers where already working, but less powerful (in terms of throughput) solutions are shipped to get later on updated by the final product which then uses dedicated hardware cores where sensible.

A

Instruction Set

The following chapter will present an overview over the CRYPTONITE instruction set. It does not cover the instruction word format which is explained in Appendix B.

A.1 Immediate Values

As mentioned in Chapter 5, CRYPTONITE basically supports two kinds of immediate values which are the global immediate values (GIVs) of 64-bit being used for ALU and MU register initialization and a 12-bit immediate used for counter register initialization and providing program addresses. The latter can be either used as destination address for branch instructions or immediate values for the MAU's local address registers. In addition, the MU instruction vector contains a 4-bit immediate field providing immediate index values or local address register values.

GIVs are only available through the CU's `LDI` instruction as shown in Table A.1 which changes the interpretation of the remaining instruction word from ALU/MU control to GIV transport. The 12-bit values are part of the "normal" instruction word and are transported through the address/immediate (A/I) field. See Appendix B.1 for a more detailed information on this topic.

Instruction	Operands	Description
LDI	$reg_1, \#val_1, reg_2, \#val_2$	load 64-bit immediate value val into register reg strands 1 and 2; reg can be either one of the ALU registers r0 to r3 or the MU's data output register; Use of this instruction changes interpretation of the instruction word as explained in Appendix B.

Table A.1: Immediate Load Instruction (applies to Control Unit)

A.2 Control Instructions

These instructions apply to the Control Unit and are needed for program flow control as listed in Table A.2. The CU instructions divide into three groups (besides the special LDI instruction) which are counter register initialization (LD and CLR), branching (BRA, BNZ, BEZ, and DBNZ) plus the HOLD instruction which stops computation and puts CRYPTONITE on hold until an externally applied hold input signal is lowered. During this instruction, CRYPTONITE performs NOP operations on all slots while the program counter is not incremented. To enable and disable this kind of interrupt, EI and DI commands apply.

For the operands, reg refers to one of the multi-purpose registers employed within the control unit where $addr$ denotes a 12-bit address provided through the A/I within program memory. $imm8$ represents an 8-bit immediate also provided through the A/I field. This, of course, means that either an address or an immediate value can be provided at the same time.

A.3 Memory Instructions

Memory instructions fall into three classes which are register-to-memory, memory-to-register (A.3), and DES-specific operations (A.4). As men-

Instruction	Operands	Description
LD	$reg, \#imm8$	load immediate value $imm8$ into register reg
CLR	reg	short for `LD reg,#0`
DBNZ	$reg, addr$	decrement register and branch to address $addr$ if the result is not equal to zero
BNZ	$addr, reg$	conditional jump to address $addr$ if register `reg` is not zero
BEZ	$addr, reg$	conditional jump to address $addr$ if register `reg` equals zero
BRA	$addr$	unconditional jump to address $addr$
EI	n/a	enable external interrupt
DI	n/a	disable external interrupt
HOLD	n/a	put CRYPTONITE into idle loop until the signal applied to the `HOLD` input pin is lowered
NOP	n/a	no operation

Table A.2: Control Instructions

tioned in Section 5.8.4 the DES unit is part of the MAU, therefore its instructions are listed in this section.

Here, lar refers to a local address register, namely `lr0` to `lr7` for memory-to-register operations and `lw0` to `lw7` for the reverse direction. dst and src can be ALU registers (`r0` to `r3`, the data input/output registers `dir` and `dor`, or the link input (`lnk` (applies only to `r0`) together with size specification (`.w` or no specification for 64-bit, `.h` or `.l` for upper/lower 32-bit), e.g. `XFR r0.h,r1.l` moves the low 32-bit of `r1` to the high 32-bit of `r0`.

For index values or initialization data smaller than 16, these values will be encoded into the memory unit's 4-bit immediate field. Otherwise, the MU's `LAR` instruction has to be used for loading bigger constants from the common A/I field into the local address registers.

Instruction	Operands	Description
LAR	$lar, \#imm12$	load local address register lar with 12-bit immediate value $imm12$ encoded in the instruction word's A/I section; here lar is a 4-bit register address where values 0-7 map to `lr0` to `lr7` and 8-15 to `lw0` to `lw7`.
LDA	$lar[, idx[+idy/imm]]$	load data from address in lar into data input register; if index register idx is present the resulting address is $addr + [idx]$ as mentioned in 5.8.1
LBX	lar, idx	load S-box data from address indexed by register idx into destination register as described in 5.8.1
STA	$lar[, idx[+idy/imm]]$	store data from data output register to address in lar; if index register idx is present the resulting address is $addr + [idx]$ as mentioned in 5.8.1

Table A.3: Memory-to-Register and Register-to-Memory Instructions

Instruction	Operands	Description
DES_DL	lar	load and input-permutate data from address in lar into DES unit
DES_KL	lar	load and key-permutate key from address in lar into DES unit
DES_EX	n/a	perform first half of DES round including S-Box lookup
DES_PBEX	n/a	perform second half of current and first half of next DES round including S-Box lookup
DES_PBWR	lar	perform second half of DES round and store output-permutated data to address in lar

Table A.4: DES-specific Instructions

A.4 Arithmetic Instructions

Arithmetic instructions share the same generic form as defined by `instr(dst,src1,src2)` and follow the given definitions for src and dst operands with the extensions already mentioned in A.5. In addition, the ALU's accumulator `ac` can be used as an alternative source or destination operand. Also with the ALU it is only possible to assign a common size identifier to a sequence of destination registers.

The CRYPTONITE AU supports classes of operations which are Accumulator instructions, Boolean operations, simple arithmetic operations, complex arithmetic operations and special operations which are listed in Table A.5.

A.5 XOR Instructions

The XOR unit can be accessed in parallel to the arithmetic unit and provides only two instructions which are listed in Table A.6. The operand syntax follows the definitions given in A.3, although due to the nature of the XOR command up to six source operands, registers `r0` to `r3`, the respective in-

Operation Class	Operation	Description
Accumulator Instructions	LDA	load accumulator with register value
	STA	store accumulator value to register
Boolean Operations	AND	logical and
	OR	logical or
	XOR	logical exclusive-or
Shift Operations	SHL	logical left-shift
	SHR	logical right-shift
	ROL	rotate left
	ROR	rotate right
Arithmetics	ADD	addition w/o carry
	SUB	subtraction w/o carry
	MUL16	mod_{16} integer multiplication
	MUL32	mod_{32} integer multiplication
Special Operations	SWAP	swap 32-bit halves of registers
	SWRT0	swap and rotate halves individually by zero and 8 positions
	SWRT1	swap and rotate halves individually by 16 and 24 positions
	UPPER64	see Table 5.3
	LOWER64	see Table 5.3
	FOLDB32	see Table 5.5
	FOLDB64	see Table 5.5
	FOLDW64	see Table 5.5
miscellaneous	NOP	no operation (no operands)
	XFR	transfer source to destination operand (only uses two operands, src and dst)

Table A.5: ALU Instructions

terlink input `lnk` and the value of the data output register `dor` are possible. The destination dst can be a sequence of up to 6 registers consisting of `r0` to `r3`, `dor` and `dir`. This allows parallel storage of results into more than one register as needed for the fast AES implementation. To not overly increase the instruction word's size, parallel storage allows only one size identifier common to all destination registers.

Instruction	**Operands**	**Description**
`NOP`	n/a	no operation
`XFR`	dst, src	transfer source to destination register by routing the source value through the XOR unit
`XOR`	dst, $src1$, $src2$[, $src3$[, $src4$[, $src5$[, $src6$]]]]	logical XOR
`NEG`	dst, src	binary negation

Table A.6: XOR Unit instructions

B

Instruction Word Format

CRYPTONITE is a VLIW-style machine with a 170-bit instruction word which divides into the the command unit (CU) section, and two of each sections for MAU, ALU and XOR unit (called a $strand$) which will be explained in the following sections. Global immediate values (GIV, 64-bit) are provided through a special instruction `LDI` which turns the strand-relevant bit fields into a 64-bit immediate value followed by a 3-bit register number. As the instruction word size is big enough, the `LDI` instruction can assign individual 64-bit values to each strand within one cycle for the sake of disabling any other operation.

B.1 Instruction Word

The instruction word contains the complete instruction bit stream controlling the internal units of CRYPTONITE. As presented in Chapter 5, CRYPTONITE divides into the sections for Control Unit (CU), two Arithmetic Units (AUs) which further divide into Arithmetic Logical Unit (ALU) and XOR-Unit (XU), plus two Memory Units (MUs) corresponding with the AUs, and finally the 12-bit address/immediate field which either holds a 12-bit program address or an 8-bit immediate value for counter registers.

Naturally, the instruction word format as shown in Table B.1 resembles this architecture and divides into smaller chunks each being assigned to an internal unit. For controlling settings common within one strand the ALU Common (AC) fields were added; as depicted in Table B.1 and already mentioned in Section 5.6, a special interpretation of the instruction word applies

Normal Operation

CU	Strand #1				Strand #2				A/I
	AC	**ALU**	**XU**	**MU**	**AC**	**ALU**	**XU**	**MU**	
	– AU #1 –				– AU #2 –				

Immediate Load Instruction

CU	**Register**	**GIV**	**Register**	**GIV**	n/a

Table B.1: CRYPTONITE Instruction Word Format

for the load of 64-bit immediate values. The following sections will give a detailed description of these chunks.

In several cases the size of employed registers, addressed by a 3-bit field as listed in Table B.2 has to be denoted by a 2-bit control value. This value maps to register sizes as shown in Table B.3. If ALU registers `r0` to `r3` are addressed, state 00_2 on the input side means clearing the register; on the output side, the register is "muted" meaning that instead of the register's content the zero value is put on the referring output bus which is also true for the data output register. For the data input register (DIR) state 00_2 represents the `NOP` value meaning that the DIR is left unchanged and not updated with an ALU result.

Register Address		`rrr`	
rrr	=	0xx	register `r0` to `r3`
		100	data output register
		101	data input register
		110	interlink
		111	reserved

Table B.2: Register Adresses

B.2 Support of Immediate Values

Since CRYPTONITE is a 64-bit architecture, providing support of immediate values requires 64 bits for the value itself plus 4 bits for the destination

Register Size	ss	
ss	= 00	mute/clear register (see Section B.1)
	01	address lower 32-bit half
	10	address upper 32-bit half
	11	address all 64-bit

Table B.3: Register Size Selection Values

register (registers `r0` to `r3`, data input and output register of each strand). To not bloat the already wide instruction word for the CRYPTONITE architecture a different approach was taken to add support for immediate values. If the `LDI` (load immediate value) command is encountered, command vectors usually controlling ALU and associated MU will hold a 64-bit value and the register address. This is possible because the instruction vectors for ALU and MU are big enough to provide the necessary space to hold this information. Unused bits will be disregarded and should be padded by the assembler software with zero bits to ease future use.

B.3 Control Unit Command Encoding

As mentioned in Section 5.6 and further explained in Section A.2 the control unit supports 8 commands being responsible for program flow control. This also includes loading of counter registers with immediate values provided through the A/I field of the instruction word. Table B.4 shows the command encoding for CRYPTONITE's control unit. The CU contributes 8 bits to the instruction word.

B.4 ALU-common Control Encoding

The ALU contains some units which are shared either within the ALU (data input and output register of the corresponding MU) or between the two ALUs (interlink). Configuration is done through 6 bits per ALU as explained in Table B.5.

CU Instruction Pattern	`cccc rrrr`	
`cccc`	CU command code	
	0000 `NOP`	no operation
	0001 `HOLD`	suspend and wait for Hold input signal to lower
	0010 `EI`	enable external interrupt
	0011 `DI`	disable external interrupt
	0100 `LCR`	load counter register denoted by `rrrr`
	0101 `LDI`	load 64-bit immediate
	011x reserved	
	1000 `BRA`	branch immediate
	1001 `BEZ`	branch if zero
	1010 `BNZ`	branch if not zero
	1011 `DBNZ`	decrement and branch if not zero
	11xx reserved	
`rrrr`	register selection (`cr0` to `cr15`)	

Table B.4: Control Unit Instruction Format

ALU Common Configuration Pattern	`ll oo ii`
`ll`	Link Size (output size of opposite ALU's register `r0`)
`oo`	Data Output Register Size
`ii`	Data Input Register Size

Table B.5: ALU-common Configuration Pattern

B.5 XOR Unit Command Encoding

The XOR unit is special in a way that it supports up to 6 source operands which are the four ALU registers `r0` to `r3`, the interlink input and the current value of the MU's data output register. The command encoding is summed up in Table B.6. Each XU contributes 20 bits to the instruction word. The `XFR` instruction does not explicitly exist for this unit, instead the `NOP` operation applies together with appropriate source and destination register settings.

XOR Instruction Pattern	`c 0123oiDD 11 22 33 44 o l n`
`0123oi`	destination register switches
`DD`	destination register size
`11`	register `r0` output size selection
`22`	register `r1` output size selection
`33`	register `r2` output size selection
`44`	register `r3` output size selection
`o`	data output register (DOR) usage (1=use)
`l`	link input usage (1=use)
`n`	result inversion (1=invert)
`c`	XOR unit command code
	0 `NOP` no operation
	1 `XOR` exclusive-or operation

Table B.6: XOR Unit Instruction Format

B.6 Arithmetic Unit Command Encoding

This is the most complex unit of CRYPTONITE consisting of several subunits like the multiplication unit, the bitmux unit, and the configurable barrel shifter/rotator. Similarly to XU, also for the AU no dedicated `XFR` instruction exists; instead, register transfer is encoded into the appropriate register settings together with a `NOP` operation. The control word encoding is shown in Table B.7 and contributes 25 bits per ALU to the instruction word.

AU Instruction Pattern	`ccccc r 0123oiaDD sssSS tttTT`	
`0123oiaDD`	destination register switches and size	
`sssSS`	source #1 register and size	
`tttTT`	source #2 register and size	
`cccc`	AU command code	
	00000 `NOP`	no operation
	00001 reserved	
	00010 `LDA`	load accumulator
	00011 `STA`	store accumulator
	00100 `AND`	logical and
	00101 `OR`	logical or
	00110 `XOR`	logical xor
	00111 `SWAP`	swap half-words
	01000 `SWRT0`	swap and rotate (0/8)
	01001 `SWRT1`	swap and rotate (16/24)
	01010 `UPPER64`	upper64 operation
	01011 `LOWER64`	lower64 operation
	01100 `FOLDB32`	foldb32 operation
	01101 `FOLDB64`	foldb64 operation
	01110 `FOLDW64`	foldw64 operation
	01111 reserved	
`cccc`	AU command code	
	10r00 `SHL`	shift left
	10r01 `SHR`	shift right
	10r10 `ROL`	rotate left
	10r11 `ROR`	rotate left
	r: shifter configuration	
	0: 1x64-bit	
	1: 2x32-bit	
	11000 `MUL16`	16-bit mod_{16} multiplication
	11001 `MUL32`	32-bit mod_{32} multiplication
	11010 `ADD`	Addition
	11011 `SUB`	Subtraction
	111xx reserved	

Table B.7: Arithmetic Unit Instruction Format

B.7 Memory Unit Command Encoding

The Memory Unit provides 10 commands which cover register transfer, memory/register and register/memory transport, DES-specific operations and S-Box accesses as described in 5.8. Immediate values are supported in two ways, short individual 4-bit values encoded into the instruction format for providing index steps or table boundaries for modulo addressing, and 12-bit values for Local Address Register initialization and branch addresses as provided by the A/I field. The command encoding is shown in Table B.8. Each MU contributes 24 bits to the instruction word.

MU Instruction Pattern	`o cccc sSS dDD xxx yyy mmm iiii`		
`o`	select ALU out data		
	0	route XOR output to Memory Unit	
	1	route AU output to Memory Unit	
`cccc`	MU command code		
	0000	`NOP`	no operation
	0001	`LAR`	load locale address register with immediate value
	0010	`LDR`	load data input register from memory
	0011	`STR`	store data output register to memory
	01xx	reserved	
	1000	`DES_DL`	DES data load
	1001	`DES_KL`	DES key load
	1010	`DES_EX`	DES expand operation
	1011	`DES_PBEX`	DES expand & P-box operation
	1100	`DES_PBWR`	DES P-box & write-back operation
	1101	reserved	
	111x	reserved	

Table B.8: Memory Unit Instruction Format

(continued on next page)

MU Instruction Pattern	`o cccc sSS dDD xxx yyy mmm iiii`		
`sSS`	source operand `s` and size `SS`		
	0	Data Input Register	
	1	Data Output Register	
`dDD`	destination operand `d` and size `DD`		
`xxx`	index register		
`yyy`	increment register		
`mmm`	addressing mode		
	000		plain
	001	idx	indexed
	010	$idx + idy$	indexed w/ register postincrement
	011	$idx + imm$	indexed w/ immediate postincrement
	100		S-Box access
	101		DES-based S-Box access
	110	$(idx + idy)\ \%\ imm$	indexed w/ register postincrement and immediate modulo
	111	$(idx + imm)\ \%\ idx$	indexed w/ immediate postincrement and register modulo

Table B.8: Memory Unit Instruction Format $(continued)$

C

Proposed Assembly Language Format

Naturally, the assembly language format very much resembles the target architecture. Since CRYPTONITE is a parallel VLIW-style architecture dividing into Control Unit and two individual strands consisting of ALU and associated Memory Unit this chapter will work out definitions and conventions regarding the CRYPTONITE assembly language format.

C.1 Register Naming Convention

CRYPTONITE consists of two identical strands, each containing four ALU registers and 16 local address registers distributing on 8 local read and 8 local write registers. In addition, 16 counter registers exist which are strand-independent. To linearize register naming, the convention listed in Table C.1 applies.

C.2 Assembly Language Format

Following this register naming convention, the assembly language format does not necessarily need to mimic the CRYPTONITE architecture as shown in Table C.2. However, doing so will add clear structure to the code and ease both, program development and debugging since the programmer can easily keep track on used and spare units and the data flow.

Since due to the comparatively small sizes of typical crypto algorithms it is very likely that CRYPTONITE will be programmed in plain assembly

Register Type	Register Name
Counter Register	`cr0-cr11`
ALU #1 Zero Flag	`cf0` or `cr12`
ALU #1 Carry Flag	`zf0` or `cr13`
ALU #2 Zero Flag	`cf1` or `cr14`
ALU #2 Carry Flag	`zf1` or `cr15`

Register Type	Strand #1	Strand #2
ALU Register	`r0-r3`	`r4-r7`
LAR/read	`lr0-lr7`	`lr8-lr15`
LAR/write	`lw0-lw7`	`lw8-lw15`

Table C.1: Register Naming Conventions

language. For this reason, human programmability becomes an issue. Programmers, however, usually dislike following strict schemes as shown in C.2 (although doing so would likely prevent quite a number of programming issues) but rather address units directly. For this reason, a CRYPTONITE mnemonic should follow the notation given in Table C.3.

```
control word;
mu#1 command; au#1 command; xu#1 command;
mu#2 command; au#2 command; xu#2 command
```

Table C.2: VLIW-style Assembly Instruction

As there is only one Control Unit within CRYPTONITE, there is no need to add any unit identifier to CU-related commands, these apply only for strand-relevant commands which are `A` for ALU, `M` for MU, and `X` for XU. These identifiers make it possible to maintain same names for equal operations since the assignment to units is done through the unit identifier which eases memorizing the CRYPTONITE commands: For example, the difference between an ALU-based or an XU-based XOR operation would simply be `XOR.A` as opposed to `XOR.X`. The assignment of commands to strands is usually done through the register numbering. In some cases, namely the `DES_EX` and `DES_PBEX` instructions, no parameters are assigned. Here, the

```
<mnemonic>::=<cu_cmd>|<strand_cmd>
      <cu_cmd>::=<letter><letter>{<letter>}
      <strand_cmd>::=<cmd>{"."{<unit>}<strand>}
              <cmd>::=<letter><letter>{<letter>}
              <unit>::="A"|"M"|"X"
      <strand>::="0"|"1"
```

Table C.3: BNF Notation of a CRYPTONITE Mnemonic

```
<instruction>::=
    "{"
    null|<command>{";" <command>}|<comment>
    "}"
<comment>::="#"<char>{<char>}"\n"
<command>::=
    <mnemonic>
    {
        <whitespace>
        <bparm>|<bparm>","<cparm>|<cparm>
    }
<bparm>::=<params>|"["<params>","<dst>"]"
<cparm>::=<params>|<params>","<dst>
<params>::=<letter>{<letter>|<digit>}
```

Table C.4: BNF Notation of a CRYPTONITE Instruction

```
<whitespace>::=" "|"\t"
<letter>::="A"-"Z"|"a"-"z"
<digit>::="0"-"9"
<symbol>::="!"-"/"|":"-"@"|"["-"`"|"{"-"~"
<op>::="+"|"-"

<alpha>::=<letter>|<digit>|"_"
<char>::=<whitespace>|<alpha>|<symbol>
<number>::=
        <hex_number>|
        <oct_number>|
        <dec_number>
<hex_number>::="0x"<hex_digit>{<hex_digit>}
<hex_digit>::="0"-"9"|"A"-"F"|"a"-"f"
<oct_number>::="0o"<oct_digit>{<oct_digit>}
<oct_digit>::="0"-"7"
<dec_number>::=<digit>{<digit>}
<name>::=
        "_"<alpha><alpha><alpha>{<alpha>}|
        <letter><alpha><alpha>{<alpha>}
```

Table C.5: BNF Notation of Primitives and Composita

strand is denoted by the respective number, `.0` or `.1`. If a command can be assigned to a unit by name – which is usually the case – the unit identifier can be omitted. Similarly, the strand number can be omitted if the assignment can be made based on the register number.

To not force the programmer to fill up unused slots with unnecessary `NOP` instructions, bundles of commands which should be assigned to the CRYPTONITE units within the same cycles are enclosed in curly brackets. This leads to the definition of a complete assembly instruction shown in Table C.4. Together with the unit/strand assignment rules, writing `NOP` instruction are completely avoided. Instead, the assembler initializes each instruction word with `NOP` instructions for all units and replaces these with the commands found within each assembly language instruction.

CRYPTONITE supports branch instructions. For this reason, the assembly language also needs to support labels as branch targets. The definition for labels is given in Table C.6. Similarly, an assembly language should support constant definitions which together with CRYPTONITE is especially useful for assigning local memory addresses to more descriptive names. The definition for such constant assignments is also given in Table C.6.

Finally, it must been noted that certain ALU commands allow multiple destinations. In such cases, the comma-separated list of destination operands has to be enclosed in square brackets as shown in Table C.4.

```
<label_decl>::=<name>":"
<const_decl>::=
        <name>
        <whitespace>{<whitespace>}
        <equ>
        <whitespace>{<whitespace>}
        <number>|
        <name>{<op><number>}
<equ>::="EQU"|"equ"
```

Table C.6: BNF Notation of Labels and Constant Declarations

C.3 Programming Examples

This section will now contain some basic programming examples followed by a complete algorithm implementation.

C.3.1 Example 1: Initialization

This example starts with a `HOLD` command which puts CRYPTONITE on hold and waits until external initialization (signalled through the level of the `HOLD` input line) is finished. Following this instruction, counter register `cr0` and ALU registers `r0` (strand #1) and `r4` (strand #2) are cleared. Local address registers `lr0` (strand #1) and `lr8` (strand #2) for reading are initialized to memory location `0x0000`.

```
# wait for initialization of
# internal memory
{ HOLD }

# initialize registers
{
        CLR cr0 ;
        LAR lr0,#0x0000 ; LAR lr8,#0x0000 ;
        CLR r0 ; CLR r4
}
```

C.3.2 Example 2: Update key & data, return processed data

For crypto algorithms it is vital to return processed data and fetch in new input data such as plain text and key updates. This example demonstrates how to achieve this with CRYPTONITE. The main crypto algorithm is embedded within `DI` and `EI` instructions to disallow interrupts during the encryption process which makes sure that updates will only happen after the current data block is processed. Since this is mainly a control flow example, it does not contain any computation relevant parts. In real life, it will be most likely possible to embed all these control instructions which will save cycles and program size.

```
main:
        { DI }
        # encryption takes place here
        { EI }

        # wait for external access
        { HOLD }

        # and jump back to main loop
        { BRA main }
```

C.3.3 Example 3: DES Implementation

This DES implementation will now give an example for a complete algorithm implementation. Although this algorithm does not make use of all the CRYPTONITE's features it will give an easy to understand example for using the CRYPTONITE assembly language. It also shows the need for partially unrolling loops to circumvent pipelining issues together with branch instructions.

```
# DES example w/ 37 cycles for DES processing
# lr0: pointer to input data
# lw0: pointer to output data
# lr1: pointer to key data
# lr2: pointer to S-Box

        # constant declarations
        in_dat  EQU 0x0000
        in_key  EQU in_dat+1
        out_dat EQU 0x0000
        sbox    EQU 0x0100

init:   # Cryptonite starts execution here

        # let external unit init S-Box memory
        # and transfer initial set of data/key
        { HOLD ; LAR lr0,in_dat }
```

```
        { LAR lr1,in_key }
        { DI ; LAR lw0,out_dat }
        { LAR lr2,sbox }

main:   # load data and key
        { DES_DL lr0 }
        { DES_KL lr1 }

body:
        # round 1.1, init counter
        { LD cr0,0x08 ; DES_EX.0 }
        { LBX lr2,des }

loop:   # rounds 1.2-16.1
        # 2-cycle delay for DBNE
        { DES_PBEX.0 }
        { DBNE cr0,loop ; LBX lr2,des }
        { DES_PBEX.0 }
        { LBX lr2,des }

        # round 16.2
        { DES_PBWR lw0 }
        { EI }

        # update/readback
        { HOLD }

        # process new data
        { BRA body ; DES_DL lr0 }
        { DI ; DES KL lr1 }
```

D

Glossary

3DES Also known as Triple-DES; consists of three sequentially performed iterations of 16 ▶DES rounds. Can be either used with two keys where iterations one and three use the first and iteration two the second key, or with individual keys per iteration.

AES Advanced Encryption Standard; new ▶NIST crypto standard from 2001. The AES standardization process was an open competition of many algorithms.

ALU Arithmetic Logical Unit; a CRYPTONITE ALU contains ▶AU and ▶XU

ASIC Application Specific Integrated Circuit; used as a cost-friendly alternative to fully custom chip design

Asymmetric Cryptography Cryptographic method using a pair of keys, one for encryption (K_1) and one for decryption (K_2) so that the formula $M = D_{K_2}(C_{K_1}(M))$ is valid with M being the message, C the encryption and D the decryption function.

AU Arithmetic Unit

Barrel Shifter Circuitry being able to perform bit shifts of arbitrary size within one cycle

CPLD Complex Programmable Logic Device; ▶EEPROM-based programmable logic device with complexities between 32 and 512 macrocells, typically used for glue logic or simple control devices

CU	Control Unit
DES	Data Encryption Standard; former ▶NIST crypto standard from 1977
DRM	Digital Rights Management; (mostly cryptographic) methods to maintain author rights for digital media
EAU	External Access Unit; interface between CRYPTONITE core and external devices
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field Programmable Gate Array; usually ▶SRAM-based programmable logic device allowing to be loaded with fairly complex logic, often used for prototyping ▶ASICs
IDEA	International Data Encryption Algorithm
Key Generation	Process of generating big prime numbers as ▶public and ▶private key for ▶asymmetric cryptography; not to be mixed up with ▶round key generation
Logical Element (LE)	Similar to macrocell but usually refers to ▶ASICs or ▶FPGAs
Loop	see ▶Loop
Macrocell	Building block within ▶CPLDs holding a configurable storage cell (flip flop) with input and output switch matrices
MD4	Message Digest #4, Hash Algorithm producing a 512-bit hash
MD5	Message Digest #5, Hash Algorithm producing a 512-bit hash; successor to ▶MD4
NIST	National Institute of Standards and Technology
Private Key	Secret key for asymmetric encryption and decryption
Public Key	Publicly shared key for asymmetric encryption and decryption
RC4	Rivest Cipher #4, Cipher Algorithm
RC5	Rivest Cipher #5, Cipher Algorithm; successor to ▶RC4
RC6	Rivest Cipher #6, Cipher Algorithm; successor to ▶RC5 and competitor in the ▶AES selection

Rijndael	Cipher Algorithm named after the inventors V. Rijmen and J. Daemen; winner of the AES competition
Round Key Generation	Method of generating individual keys for each encryption (or decryption) round of an algorithm; these keys are based on a provided cipher (or decipher) key.
SHA	Secure Hash Algorithm
SHA-1	Secure Hash Algorithm #1 producing 160-bit hash; first hash algorithm to be standardized by ▶NIST
SHA-256	Secure Hash Algorithm producing a 256-bit hash value
SHA-384	Secure Hash Algorithm producing a 384-bit hash value
SHA-512	Secure Hash Algorithm producing a 512-bit hash value
Slice	▶Xilinx-specific term for logical elements within their Virtex family of ▶FPGAs
Strand	Within this document, the term *strand* is used to describe (mainly) data independent branches of a data dependency graph. The more pictural term *thread* is omitted to avoid confusion since in computer science literature this term is commonly used as a synonym for lightweighted processes. The concept of strands is also mapped to the architectural description where one strand is a compound of functional units (▶ALU and associated ▶MU) being able to process such data independent branches.
SRAM	Static Random-Access Memory
Symmetric Cryptography	Cryptographic method using the same key K for encryption and decryption so that the formula $M = D_K(C_K(M))$ is valid with M being the message, C the encryption and D the decryption function. In addition, C and D can be identical and differ only in round key calculation.
Synopsys	Both, vendor (Synopsys, Inc.) of an integrated development software for hardware design, and the name of the software itself

Triple-DES	see 3DES
Tunnel	Virtual direct connection between two network nodes using ▶VPN mechanisms
VHDL	▶VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
VPN	Virtual Private Network; private network realized on a shared network infrastructure by means of cryptography
Xilinx	Vendor of ▶CPLDs and ▶FPGAs
XOR	exclusive-or (Boolean operation)
XU	▶XOR Unit

Bibliography

[1] Record-Breaking DES Key Search Completed. July 1998. http://www.cryptography.com/resources/whitepapers/DES.html.

[2] Project DES. January 1999. http://www.distributed.net/des/.

[3] Marta Dinata A., Sarwono, and Sigit. OpenCores IDEA Core. Sep 2001. http://www.opencores.org/projects/idea/.

[4] Advanced Micro Devices, Inc. AMD 29000 Family Databook. 1987.

[5] I Putu Gede Al, Rudi Harianta, and Hendro Suryadi. OpenCores TwoFish Core. Feb 2002. http://www.opencores.org/projects/-twofish_team/.

[6] Alliance for Telecommuncations Industry Solutions. ATIS Home-Page. 2002. http://www.atis.org.

[7] Altera Corporation. APEX 20K Devices. 2002. http://www.altera.com/products/devices/apex/apx-index.html.

[8] Altera Corporation. Corporate Web Site. 2002. http://www.altera.com.

[9] Amphion Semiconductor Ltd. Corporate Web Site. 2001. http://www.amphion.com.

[10] Amphion Semiconductor Ltd. CS5210-40 High Performance AES Encryption Cores Product Information. 2001. http://www.amphion.com/acrobat/DS5210-40.pdf.

[11] Amphion Semiconductor Ltd. CS5210-40 High Performance AES Decryption Cores Product Information. 2002. http://www.amphion.com/acrobat/DS5250-80.pdf.

[12] A. O. L. Atkin and F. Morain. Elliptic Curves and Primality Proving. *Math. Comp.*, 61:203:29–68, July 1993.

[13] Dario Benedetto, Emanuele Caglioti, and Vittorio Loreto. Language Trees and Zipping. *Physical Review Letters*, 88, January 2002. http://prl.aps.org/.

[14] T. Beth and D. Gollmann. Algorithm Engineering for Public Key Algorithms. *IEEE Journal on Selected Areas in Communications*, 7 no. 4:458–466, 1989.

[15] Broadcom Corporation. BCM5801 Cryptographic Processor Data Sheet. 2002. http://www.broadcom.com/cgi-bin/access/-new_request.cgi?file=5801-5805-5820-SRL101-R.pdf;rt=1.

[16] Broadcom Corporation. BCM5801 Cryptographic Processor Product Brief. 2002. http://www.broadcom.com/pbs/BCM5801.pdf.

[17] Broadcom Corporation. BCM5802 Cryptographic Processor Data Sheet. 2002. http://www.broadcom.com/cgi-bin/access/-new_request.cgi?file=5802-DS01-R.pdf;rt=1.

[18] Broadcom Corporation. BCM5802 Cryptographic Processor Product Brief. 2002. http://www.broadcom.com/pbs/BCM5801.pdf.

[19] Broadcom Corporation. BCM5805 Security Processor Data Sheet. 2002. http://www.broadcom.com/cgi-bin/access/new_request.cgi?-file=5805-DS03-R.pdf;rt=1.

[20] Broadcom Corporation. BCM5805 Security Processor Product Brief. 2002. http://www.broadcom.com/pbs/BCM5805.pdf.

[21] Broadcom Corporation. BCM5820 E-Commerce Processor Data Sheet. 2002. http://www.broadcom.com/cgi-bin/access/-new_request.cgi?file=5820-DS03-R.pdf;rt=1.

[22] Broadcom Corporation. BCM5820 E-Commerce Processor Product Brief. 2002. http://www.broadcom.com/pbs/BCM5820.pdf.

[23] Broadcom Corporation. BCM5821 E-Commerce Processor Product Brief. 2002. http://www.broadcom.com/pbs/BCM5821.pdf.

[24] Broadcom Corporation. BCM5840 Gigabit Security Processor Product Brief. 2002. http://www.broadcom.com/pbs/BCM5840.pdf.

[25] Broadcom Corporation. Corporate Web Site. 2002. http://www.broadcom.com.

[26] Ronald H. Brown, Mary L. Good, and Arati Prabhakar. Data Encryption Standard (DES) (FIPS 46-2). *Federal Information Processing Standards Publication (FIPS)*, Dec 1993. http://www.itl.nist.gov/fipspubs/fip46-2.html (initial version from Jan 15, 1977).

[27] Ronald H. Brown and Arati Prabhakar. FIPS180-1: Secure Hash Standard (SHA). *Federal Information Processing Standards Publication (FIPS)*, May 1993. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[28] Ronald H. Brown and Arati Prabhakar. Security Requirements for Cryptographic Modules (FIPS 140-1). *Federal Information Processing Standards Publication (FIPS)*, Jan 1994. http://www.itl.nist.gov/fipspubs/fip140-1.htm (initial version from Apr 14, 1982).

[29] Chris Caldwell. Finding primes & proving primality, 2002. http://www.utm.edu/research/primes/prove/index.html.

[30] C. M. Campbell. Design and Secification of Cryptographic Capabilities. *IEEE Cmputer Society Magazine*, 16(6):15–19, November 1978.

[31] Lakshmi N. Chakrapani, Krishna V. Palem, and Weng Fai Wong. Triceps: Enhancing the Trimaran Compiler Infrastructure for Strong/ARM Code Generation. 2001. http://www.crest.gatech.edu/publications/codegen.zip.

[32] Intel Corporation. Intel 82801 BA I/O Controller Hub 2 (ICH2) and Intel 82801BAM I/O Controller Hub 2 Mobile (ICH2-M). October 2000. Document Number 290687-002.

[33] Intel Corporation. Intel 850 Chipset Family: 82850/82850E Memory Controller Hub. May 2002. Document Number 290691-002.

[34] Intel Corporation. Intel Architecture for Applied Computing. 2002. http://developer.intel.com/design/intarch/.

[35] Intel Corporation. Intel Pentium 4 Processor with 512-KB L2 Cache on 0.13 Micro Process at 2 GHz, 2.20 GHz, 2.26 GHz, 2.40 GHz and 2.53 GHz. May 2002. Document Number 298643-003.

[36] Corrent Corporation. Corporate Web Site. 2002. http://www.corrent.com.

[37] Corrent Corporation. General Product Information. 2002. http://www.corrent.com/products.phtml.

[38] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer-Verlag, New York, 2001. ISBN 0-387-94777-9.

[39] J. Daemen and V. Rijmen. The block cipher Rijndael, 2000. LNCS1820, Eds: J.-J. Quisquater and B. Schneier.

[40] J. Daemen and V. Rijmen. Advanced Encryption Standard (AES) (FIPS 197). Technical report, Katholijke Universiteit Leuven / ESAT, Nov 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[41] William M. Daley and Raymond G. Kammer. Fips186-2: Digital signature standard. *Federal Information Processing Standards Publication (FIPS)*, Jan 2000. http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf.

[42] D. W. Davies and G. I. P. Parkin. The Average Size of the Key Stream in Output Feedback Encipherment. *Advances in Cryptology: Proceedings of Crypto 82*, pages 97–98, 1983.

[43] D. W. Davies and G. I. P. Parkin. The Average Size of the Key Stream in Output Feedback Encipherment. *Cryptogprahy, Proceedings of the Workshop on Cryptography, Burg Feuerstein, Germany, March 29-April 2, 1982*, pages 263–279, 1983.

[44] W. Diffie and M.E. Hellman. Privacy and Authentication: An Introduction to Cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[45] Dino Oliva. Efficient Implementation of the AES/Rijndael Algorithm. 2001. (internal paper).

[46] Dino Oliva and Nevin Heintze and Rainer Buchty. Straight-forward Implementation of the AES/Rijndael Algorithm. Sep 2001. (internal paper).

[47] Brian Gladman et al. AES Algorithm Efficiency. Feb 1999. http://fp.gladman.plus.com/cryptography_technology/aes/.

[48] J. Gait. A New Nonlinear Pseudorandom Number Generator. *IEEE Transactions on Software Engineering*, SE-3(5):359–363, September 1977.

[49] P. Gaundry, F. Hess, and N. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. January 2000. http://www.hpl.hp.com/techreports/2000/HPL-2000-10.html.

[50] Heise Online-Redaktion (wst). Zip-Algorithmus identifiziert Autoren. *Heise Newsticker*, January 2002. http://www.heise.de/-newsticker/data/wst-28.01.02-003/.

[51] M.E. Hellman. On DES-Based Synchronous Encryption. 1980.

[52] John L. Hennessy and David A. Patterson. Morgan Kaufmann Publishers Inc., 1996. 2nd Edition.

[53] Hifn Inc. 7711 Encryption Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0001-04-7711.pdf.

[54] Hifn Inc. 7751 Encryption Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0013-03-7751.pdf.

[55] Hifn Inc. 7811 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0018-02-7811.pdf.

[56] Hifn Inc. 7814/7851/7854 Network Security Processors Device Specifications. 2002. http://www.hifn.com/docs/a/DS-0030-04-7814-7851-7854-Device-Specification.pdf.

[57] Hifn Inc. 7901 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0023-01-7901.pdf.

[58] Hifn Inc. 7902 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0040-00-7902.pdf.

[59] Hifn Inc. 7951 Network Security Processor Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0028-02-7951.pdf.

[60] Hifn Inc. 8065/8165 Secure Session Processor Product Information. 2002. http://www.hifn.com/docs/HIPPII-8065-8165.pdf.

[61] Hifn Inc. 8154 Security Processor Product Information. 2002. http://www.hifn.com/docs/HIPPII-8154.pdf.

[62] Hifn Inc. 8165 Secure Session Processor Short Data Sheet. 2002. http://www.hifn.com/docs/a/DS-0049-B-8165-Short-Data-Sheet.pdf.

[63] Hifn Inc. Corporate Web Site. 2002. http://www.hifn.com.

[64] Renato Iannella. Digital Rights Management (DRM) Architectures. *D-Lib Magazine*, 7.6, June 2001. ISSB 1082-9873, http://www.dlib.org/dlib/june01/iannella/06iannella.html.

[65] Xilinx Inc. Synopsys FPGA Compiler II. 2002. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=-synopsys_fpga_express.

[66] Intel Corporation. Intel Itanium Processor Family Home. 2002. http://developer.intel.com/design/itanium.

[67] Jüri Pöldre. Cryptoprocessor PLD001 (Master Thesis). June 1998.

[68] R. R. Jueneman. Analysis of Certain Aspects of Output-Feedback Mode. *Advances in Cryptology: Proceedings of Crypto 82*, pages 99–127, 1983.

[69] S. T. Kent. Encryption-Based Protection Protocols for Interactive User-Computer Communications. *MIT/LCS/TR-162*, May 1976.

[70] Philip M. Klutznick and Ernest Ambler. DES Modes of Operation (FIPS 81). *Federal Information Processing Standards Publication (FIPS)*, Dec 1980. http://www.itl.nist.gov/fipspubs/fip81.htm.

[71] S. Shrivastava L. Gao and G. Sobelman. Elliptic curve scalar multiplier design using FPGAs. *Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, LNCS 1717, August 1999.

[72] X. Lai. Detailed description and a software implementation of the ipes cipher. Nov 1991. unpublished paper.

[73] X. Lai. On the design and security of block ciphers. *ETH Series in Information Processing*, 1, 1992.

[74] X. Lai, J. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. *Advances in Cryptology - EUROCRYPT 91 Proceedings*, pages 17–38, 1991.

[75] K. Leung, K. Ma, W. Wong, and P. Leong. FPGA implementation fo a microcoded elliptic curve cryptographic processor. *Eight Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 00)*, 2000.

[76] J. Lopez and R. Dahab. Fast multiplication on elliptic curves ofer $GF(2^m)$ without precomputation. *Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, LNCS 1717, August 1999.

[77] Microsoft Corporation. Technology & Tools: Digital Rights Management. December 2001. http://www.microsoft.com/windows/-windowsmedia/drm.asp.

[78] National Institute of Standards and Technology. Recommended Elliptic Curves for Federal Government Use, May 1999. http://csrc.nist.gov/encrytion.

[79] NetOctave, Inc. NSP2000 Security Processor. 2001. http://www.chipcenter.com/networking/images/prod/prod237b.pdf.

[80] NetOctave, Inc. NSP2000-SSL Security Accelerator. 2001. http://www.netoctave.com/netoctave.asp?template=products&-section=ssl.

[81] NetOctave, Inc. NSP3000-IPsec Security Accelerator. 2001. http://www.netoctave.com/netoctave.asp?template=products&-section=ipsec.

[82] NetOctave, Inc. NSP3200 Security Processor. 2001. http://www.netoctave.com/downloads/NSP3200Security-Processor.pdf.

[83] NetOctave, Inc. NSP3200 Security Processor. 2001. http://www.netoctave.com/downloads/FlowThrough_Security_-Architecture_WP.pdf.

[84] Netoctave, Inc. Corporate Web Site. 2002. http://www.netoctave.com/.

[85] Open Software Foundation (OSF). OSF Homepage. 2002. http://www.osf.org.

[86] OpenCores.org. Project Website. 2002. http://www.opencores.org/.

[87] G. Orlando and C. Paar. A high performance elliptic curve processor for $GF(2^m)$. *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, LNCS 1865, August 2000.

[88] C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast Arithmetic for Public Key Algorithms in Glaois Fields with Composite Exponents. *IEEE Transactions on Computers*, 48:1025–1034, October 1999.

[89] Peter N. Glaskowsky. At Least One Chip Niche Is Secure. *Microprocessor Report*, Feb 2001. http://www.corrent.com/pdf/-Corrent_awardarticle_reprint.pdf.

[90] Peter N. Glaskowsky. Cahners In-Stat/MDR Announcement. Feb 2001. http://www.corrent.com/pdf/MPR_press_release.pdf.

[91] R. Rivest. RFC1186: The MD4 Message-Digest Algorithm. October 1990. http://www.ietf.org/rfc/rfc1186.txt.

[92] R. Rivest. The MD4 message digest algorithm. *Advances in Cryptology - CRYPTO '90 Proceedings*, pages 303–311, 1991.

[93] R. Rivest. RFC1312: The MD5 Message-Digest Algorithm, April 1992. http://www.ietf.org/rfc/rfc1321.txt.

[94] R.L. Rivest. The RC5 Encryption Algorithm. *Dr. Dobb's Journal*, 20(1):146–148, Jan 1995.

[95] R.L. Rivest. The RC5 Encryption Algorithm. *K.U. Leuven Workshop on Cryptographic Algorithms*, 1995.

[96] Ronald R. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6^{TM} Block Cipher. August 1998. http://www.rsasecurity.com/-rsalabs/rc6/.

[97] Ronald R. Rivest, M.J.B. Robshaw, and Y.L. Yin. The Case for RC6 as the AES. May 2000. http://www.rsasecurity.com/rsalabs/rc6/.

[98] M. Rosner. *Elliptic curve cryptosystems on reconfigurable hardware*. ECE Dept, Worcester Polytechnic Institute, Worcester, USA, May 1998. Master's Thesis.

[99] Ray Savarda. Next Generation Network Security Processors: Optimal Design and Integration with Network Processors. 2001. http://www.netoctave.com/downloads/NextGen_Security-Processors.pdf.

[100] Ray Savarda. Next Generation Network Security Processors: Optimal Design and Integration with Network Processors. *Communications Design Conference 2001*, 2001.

[101] Bruce Schneier. 13.9: IDEA. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, pages 370–377, 1996. ISBN 3-89319-854-7.

[102] Bruce Schneier. 24.12: Pretty Good Privacy (PGP). *Angewandte Kryptographie*, pages 664–667, 1996. ISBN 3-89319-854-7.

[103] Bruce Schneier. 9.10 - Weitere Modi für Blockchiffrierung. *Angewandte Kryptographie*, pages 244–246, 1996. ISBN 3-89319-854-7.

[104] Bruce Schneier. 9.3 - Cipher Block Chaining. *Angewandte Kryptographie*, pages 227–232, 1996. ISBN 3-89319-854-7.

[105] Bruce Schneier. Algorithmenarten und Betriebsmodi. *Angewandte Kryptographie*, pages 223–250, 1996. ISBN 3-89319-854-7.

[106] Bruce Schneier. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*. Addison Wesley, 1996. ISBN 3-89319-854-7.

[107] Bruce Schneier. Einweg-Hashfunktionen: MD5. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, pages 498–503, 1996. ISBN 3-89319-854-7.

[108] Bruce Schneier. Einweg-Hashfunktionen: SHA. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, pages 504–507, 1996. ISBN 3-89319-854-7.

[109] Bruce Schneier. Noch mehr Blockchiffrierungen: RC5. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, pages 397–399, 1996. ISBN 3-89319-854-7.

[110] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. AES Performance Comparisons. Oct 2000. http://csrc.nist.gov/encryption/aes/round1/conf2/Schneier.pdf.

[111] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast Key Exchange with Elliptic Curve Systems. *Advance in Cryptography, Crypto 95*, LNCS 963, 1995.

[112] SecuCore Consulting Services. SecuCore AES/Rijndael Core. 2001. http://www.secucore.com/secucore_aes.pdf.

[113] SecuCore Consulting Services. SecuCore DES/3DES Core. 2001. http://www.secucore.com/secucore_des.pdf.

[114] SecuCore Consulting Services. SecuCore SHA-1/MD5/HMAC Core. 2001. http://www.secucore.com/secucore_hmac.pdf.

[115] SecuCore Consulting Services. SecuCore SHA-256 Core. 2001. http://www.secucore.com/secucore_sha256.pdf.

[116] SecuCore Consulting Services. Corporate Web Site. 2002. http://www.secucore.com/.

[117] L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing Systems*, 2 no. 22:1–17, 1997.

[118] Synopsys, Inc. Corporate Web Site. 2002. http://www.synopsys.com.

[119] Taiwan Semiconductor Manufacturing Company Ltd. Corporate Web Site. 2002. http://www.tsmc.com.

[120] Paulus Tamba. OpenCores Crypto Accelerator. Sep 2001. http://www.opencores.org/projects/crypto/.

[121] Telecom Glossary 2000. Cryptanalysis, 2000. http://www.its.bldrdoc.gov/fs-1037/dir-009/_1343.htm.

[122] Telecom Glossary 2000. Cryptography, 2000. http://www.its.bldrdoc.gov/projects/t1glossary2000/_cryptography.html.

[123] Telecom Glossary 2000. Cryptology, 2000. http://www.its.bldrdoc.gov/projects/t1glossary2000/_cryptology.html.

[124] The FreeS/WAN Project Team. FreeS/WAN Project. 2002. http://www.freeswan.org/.

[125] United Microelectronics Corp. Corporate Web Site. 2001. http://www.umc.com.

[126] United States Patent and Trademark Office. Official Gazette of the United States Patent and Trademark Office. May 1975. 934 O.G. 452.

[127] United States Patent and Trademark Office. Official Gazette of the United States Patent and Trademark Office. Aug 1976. 949 O.G. 1717.

[128] Rudolf Usselmann. OpenCores DES Core. Sep 2001. http://www.opencores.org/projects/des/.

[129] A. Wiles. Modular Elliptic Curves and Fermat's last theorem. *Ann. Math.*, 141:3:443–551, 1995.

[130] H. Wu. Low complexity bit-parallel finite field arithmetic using polynomial basis. *Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, LNCS 1717, August 1999.

[131] Lisa Wu, Chris Weaver, and Todd Austin. Cryptomaniac: A fast flexible architectore for secure communication. In *28th Annual International Symposium on Computer Architecture (ISCA 2001)*, June 2001.

[132] Xilinx, Inc. Corporate Web Site. 2002. http://www.xilinx.com.

[133] Xilinx, Inc. Virtex-II Pro Platform FPGAs. 2002. http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=-Virtex-II+Pro+FPGAs.

[134] Chi Lap Yuen. OpenCores RSA RC4 Core. Apr 2002. http://www.opencores.org/projects/rsa-rc4/.

[135] Chi Lap Yuen and Kurt Ting. OpenCores NIST Secure Hash Algorithm 256 Core. Apr 2002. http://www.opencores.org/projects/sha256/.

[136] Made Yusadana. OpenCores AES/Rijndael Core. Feb 2002. http://www.opencores.org/projects/rijndael/.

[137] Zilog, Inc. Z80 Product Details. 2002. http://www.zilog.com/-products/parts.asp?BusinessLineID=274.

[138] P.R. Zimmermann. PGP Source Code and Internals. 1995.

[139] P.R. Zimmermann. The Official PGP User's Guide. 1995.

[140] Zyfer, Inc. Corporate Web Site. 2002. http://www.zyfer.com.

[141] Zyfer, Inc. SKP-100 Gigabit Network Security Processor. 2002. http://www.zyfer.com/products/SKP-100_Data_Sheet.pdf.

I

Index