CSE 551 TERM REPORT
FALL 2004

DESIGN OF A RECONFIGURABLE HARDWARE

FOR EFFICIENT IMPLEMENTATION OF SECRET KEY AND PUBLIC KEY CRYPTOGRAPHY

SUBMITTED BY
SYED ALI MUSTAFA ZAIDI (230201)
MUSTAFA IMRAN ALI (203203)

TABLE OF CONTENTS

DESIGN OF A RECONFIGURABLE HARDWARE

FOR EFFICIENT IMPLEMENTATION OF SECRET KET AND PUBLIC KEY CRYPTOGRAPHY

ABSTRACT

For high throughput, area efficient and algorithm independent hardware implementation of cryptographic algorithms, cryptography specific reconfigurable hardware seems a promising solution. This report describes the design of such hardware for supporting both public key and private key cryptography. Though this work is preliminary in nature and needs detailed VLSI level implementation to verify the competitiveness with existing approaches, it nevertheless indicates the feasibility of potential gains mentioned before for both public key and secret key cryptography.

1. INTRODUCTION

1.1 THE IMPORTANCE OF HIGH SPEED CRYPTOGRAPHY

Cryptography currently plays a major role in many information technology applications. The Internet is the biggest information technology application by itself. According to a 2002 survey [1], more than 605.60 million people are connected to the Internet globally. In the case of confidential data, the cost-effectiveness and globalization provided by the Internet are diminished by the main disadvantage of public networks: *security risks*. Communicating over the Internet involves significant security risks since the Internet is an insecure network. Therefore, the need for securing the Internet traffic has become a fundamental issue. Many Internet applications, the most common being electronic mail, electronic banking, and electronic commerce, require the exchange of private information. As an example, when engaging in electronic commerce, customers provide credit card numbers when purchasing products. If a given connection is not secure, an attacker can easily obtain this sensitive data. In order to prevent this from occurring, the connection must be made secure. To implement a comprehensive security plan for a given network, and thus guarantee the security of a connection, the following services must be provided:

- Confidentiality: Information cannot be observed by an unauthorized party. This is accomplished via the use of secret-key and public-key encryption.
- Data Integrity: Transmitted data within a given communication cannot be altered in transit due to error or tampering by an unauthorized party. This is accomplished via Message Authentication Codes (MACs) and digital signatures.
- Authentication: Parties within a given communication session must provide certifiable proof of their identity. This may be accomplished via digital signatures, and their associated certification and validation mechanisms.

- Non-repudiation: The sender of a message may deny having made any sensitive transmission. This may be accomplished via digital signatures and third party notary services.

Cryptographic algorithms are important tools for providing the aforementioned security services. Cryptographic algorithms fall primarily within one of two categories: private key (also known as symmetric-key) and public-key (also known as asymmetric-key). Symmetric-key algorithms use the same key for both encryption and decryption. Conversely, public-key algorithms use a public key for encryption and a private key for decryption.

Symmetric-key algorithms tend to be significantly faster than public-key algorithms and as a result are typically used in bulk data encryption. In a typical session, a public-key algorithm will be used for the exchange of a session key and to provide authenticity through digital signatures. The session key may then be used in conjunction with a symmetric-key algorithm for encrypting and exchanging the bulk of the communication.

Fast implementations of cryptographic algorithms are essential to speed up critical operations as part of an overall communication protocol. For example, an SSL server needs very high speed public key encryption schemes such as RSA to sustain the high throughput rates required to manage the multitude of transactions taking place at once, while applications using the IPSec protocol requires a high speed implementation of a block cipher such as Rijndael. High throughput encryption and decryption are becoming increasingly important in the area of high speed networking as well. Many applications demand the creation of networks that are both private and secure while using public data-transmission links. These systems, known as Virtual Private Networks (VPNs), can demand encryption throughputs exceeding Asynchronous Transfer Mode (ATM) rates of one billion bits per second (Gbps). Thus, the implementation of a cryptographic algorithm must achieve high processing rate to fully utilize the available network bandwidth.

1.2 NEED FOR ALGORITHM INDEPENDENCE

The enormous success of Internet has made the Internet Protocol (IP) one of the primary communication protocols. For securing the Internet traffic, Internet Protocol Security (IPSec) was developed by the Internet Engineering Task Force. The IPSec standard is a framework of open standards for ensuring secure private communication over the Internet [2]. It extends the IP protocol by securing the IP traffic at the IP level using cryptographic methods. The major advantage of IPSec is its flexibility: It is not based on a particular cryptographic method. The cryptographic methods to be used are negotiated between the communicating entities. IPSec provides an open framework to implement any cryptographic algorithm. As a result, the IPSec standard makes it possible for security systems developed by different vendors to interoperate.

Two of the services that IPSec provides to protect IP traffic from security threats are:

- *Confidentiality*: data encryption, and
- *Key Management*: negotiating security associations and key exchanging.

Initially the two communicating parties negotiate and establish a security association (SA) for protecting the transferred data. An SA determines the cryptographic methods and the related keys to be utilized. The cryptographic methods include private- and public-key cryptography, keyed hash algorithms, and digital certificates.

Cryptography is the fundamental component of IPSec. Therefore, architectures that implement IPSec have to meet the enormous computing demands of cryptographic algorithms. Moreover, since the security parameters are dynamically negotiated by the communicating entities, IPSec architectures have to flexible enough to adapt to diverse security parameters (e.g., cryptographic algorithms, cryptographic operation mode, key length). Such flexibility is also crucial to adapt to the evolving requirements of state-of-the-art algorithms and standards.

A similar case applies to implementing support for high speed SSL implementations [3]. The new version 3.0 allows any cipher algorithm of choice to be used for encryption of session data after the establishment of initial key which can use any public key algorithm such as RSA or Diffie-Hellman for the key exchange. The changing nature of such protocols requires that the hardware should be upgradeable to support newer standards and at the same time support a high speed implementation that is faster than a software only implementation.

1.3 SOLUTIONS IN RECONFIGURABLE HARDWARE

1.3.1 WHY HARDWARE IMPLEMENTATIONS?

General purpose processors are not optimized for block cipher implementations. They also cannot provide the amount of parallelism that is required to support high speed long integer arithmetic operations required in public key algorithms. This results in a degradation of performance when compared to hardware implementations. Even the fastest software implementations of block ciphers cannot obtain the required data rates for bulk data encryption [4] [5]. As a result, hardware implementations are necessary in order for block ciphers to achieve this required performance level.

1.3.2 SECURITY ADVANTAGE OF HARDWARE IMPLEMENTATIONS

Besides performance advantages, security issues also make hardware implementations more advantageous than software-based solutions. A software implementation offers only limited physical security, especially with respect to key storage [6]. In a typical personal computer, memory external to the processor is used to store instructions and data in unencrypted form. The potential exists for an attacker to determine a cipher's keys if they are able to gain access to the system's memory. Conversely, cryptographic algorithms (and their associated keys) that are implemented in hardware are, by nature, physically more secure as they cannot easily be read or modified by an outside attacker. This is done by storing the key in special memory internal to the device. As a result, the attacker does not have easy access to the key storage area and therefore cannot discover or alter its value in a straightforward manner. In general, hardware-based solutions are the embodiment of choice for military and serious commercial applications (e.g., NSA authorizes encryption only in hardware) [7].

1.3.3 SPECIFIC ADVANTAGES OF RECONFIGURABLE HARDWARE

As mentioned in the previous section, increasingly, security standards and applications are defined to be algorithm independent. Although context switching between algorithms can be easily realized via software implementations, the task is significantly more difficult when using classical hardware implementations (i.e. ASICs based). The advantages of a software implementation include ease of use, ease of upgrade, ease of design, portability, and flexibility. The downside of traditional hardware implementations are the lack of flexibility with respect to algorithm and parameter switching. Reconfigurable hardware devices are a promising alternative for the implementation of configurable

cryptography. The potential advantages of cryptographic algorithms implemented in reconfigurable hardware include [8]:

- **Algorithm Agility**: This term refers to the switching of cryptographic algorithms during operation. The majority of modern security protocols, such as Secure Sockets Layer (SSL) or IPSec, allow for multiple encryption algorithms. The encryption algorithm is negotiated on a per-session basis; e.g., IPSec allows (among others) the Data Encryption Standard (DES), Triple-DES, Blowfish, CAST, the International Data Encryption Algorithm (IDEA), RC4, and RC6 as encryption algorithms, and future extensions are possible. Whereas algorithm agility can be very costly with traditional hardware, reconfigurable hardware can be reconfigured on-the-fly. While reconfiguration time is still an open issue to be solved, algorithm agility through reconfigurable hardware appears to be an attractive possibility.

- **Algorithm Upload**: It is perceivable that fielded devices are upgraded with a new encryption algorithm which did not exist (or was not standardized) at design time. In particular, it is very attractive for numerous security products to be upgraded for use of the Advanced Encryption Standard (AES) now that the standardization process has been completed. Assuming there is some kind of (temporary) connection to a network such as the Internet, new configuration code can be uploaded to reconfigurable devices.

- **Algorithm Modification**: There are applications which require modification of a standardized algorithm, e.g., by using proprietary S-Boxes or permutations. Such modifications are easily made with reconfigurable hardware. Similarly, a standardized algorithm can be swapped with a proprietary one. Also, modes of operation can be easily changed.

- **Architecture Efficiency**: In certain cases, hardware architecture can be much more efficient if it is designed for a specific set of parameters; e.g., constant multiplication (of integers or in Galois fields) is far more efficient than general multiplication. With reconfigurable devices it is possible to design and optimize an architecture for a specific parameter set.

- **Throughput**: Although typically slower than Application Specific Integrated Circuit (ASIC) implementations, reconfigurable implementations have the potential of running substantially faster then software implementations.

- **Cost Efficiency**: The time and costs for developing a reconfigurable implementation of a given algorithm are much lower than for an ASIC implementation (however, for high volume applications, ASIC solutions usually become the more cost-efficient choice).

1.3.4 SUITABILITY OF FPGAS

FPGA technology is a growing area of research that has the potential to provide the performance benefits of ASICs and the flexibility of general-purpose processors. Application specific hardware circuits can be created on demand to meet the computing and interconnect requirements of an application. As a result, superior performance can be expected compared with the performance of the equivalent software implementation executed on a processor. Moreover, these hardware circuits

can be dynamically modified partially or completely based on the computation requirements. Such post-fabrication customizations distinguish FPGAs from ASICs and can lead to fast turn-around time. Turnaround time is critical in environments where the application requirements and standards are changing rapidly (e.g., information technology environments). Therefore, FPGA-based solutions can result in devices with increased lifetime compared with ASIC-based solutions.

The basic feature underlying FPGAs is a fine-grained programmable logic cell. FPGAs consist of a matrix of logic cells overlaid with a network of wires [9] [10]. Both the computation performed by the cells and the connections between the wires can be configured. Moreover, on-chip memory modules can be incorporated for storing intermediate results.

The primary advantage of FPGAs over general-purpose computers is hardware parallelism. The spatial computing paradigm of FPGAs can efficiently exploit the inherent parallelism of cryptographic algorithms. On the contrary, the serial fashion of computing in general-purpose computers is a limiting factor of their performance. FPGA implementations can exploit at different levels [11]:

- Instruction-level: Multiple operations can be executed concurrently, thus there is potential for reduction in the time required to process data.

- Data-level: Multiple blocks of data can be processed in parallel in operation modes that allow parallel processing (e.g., non-feedback, interleaved). In addition, blocks of data corresponding to different keys can be processed in parallel. As a result, superior throughput can be achieved compared with a serial implementation.

- Task-level: This is particularly useful in the case of symmetric-key ciphers. The key-setup can run concurrently with the cryptographic core. The cryptographic core can start operation as soon as the key-dependent data for the first round is derived. In software implementations, the cryptographic core can not commence before the key-dependent data for all the rounds is derived.

1.3.5 THE POTENTIAL OF CUSTOM RECONFIGURABLE HARDWARE

While flexibility is an important feature of reconfigurable devices, conventional FPGAs are simply too generic to provide high performance in many situations. General-purpose reconfigurable devices, while well suited to small or irregular functions, typically suffer a stiff penalty when implementing wide and complex arithmetic operations. These types of functions need to be built from too many small logical resources and end up being spread across too general a routing structure to be efficient.

However, if the range of applications that a device is intended for is known beforehand, a designer can specialize the logic, memory and routing resources to enhance the performance of the device while still providing adequate flexibility to accommodate all anticipated uses. Common and complex operations can be implemented much more efficiently on specialized coarse-grain functional units while routing and memory resources can be tuned to better reflect the requirements.

FPGAs employ fine-grained architectures to maximize system performance at the bit level. Because block ciphers are dataflow oriented and employ wide operands, a coarse-grained architecture could offer a better solution for achieving maximum system performance, especially when implementing operations such as integer multiplication, constant multiplication in a Galois field, table look-up, or

bit-wise shift and rotation, as these operations do not require a fine-grained architecture. Finally, the interconnection matrix of FPGAs provides significantly more flexibility than is required by most block ciphers. Block ciphers require a single feedback path for iteration over their given round function, resulting in an interconnection model that is both simpler and more regular than the model of FPGAs and ASICs.

A specialized reconfigurable architecture that is optimized for the implementation of cryptographic schemes results in a system with greater flexibility as compared to custom ASIC or specialized processor solutions, as well as faster reconfiguration time when compared to a commercial FPGA solution. Overhead and off-chip communications are minimized by maintaining most or all of the system's resources on chip.

1.3.6 ISSUES IN DESIGNING CUSTOM RECONFIGURABLE HARDWARE

Internal to the chip, it is imperative that the reconfigurable datapath be tightly coupled with the control mechanisms so that the overhead associated with their interface does not become the system bottleneck. Ensuring full support of algorithm-specific operations requires maximizing the functional density of the datapath. This results in the need for a generalized and run-time reconfigurable datapath with functional blocks optimized to efficiently implement the core operations of the target cipher space. Because cryptographic algorithms are dataflow oriented, a reconfigurable element with coarse granularity offers the best solution for achieving maximum system performance when implementing operations that do not map well to more traditional, fine-grained reconfigurable architectures. Finally, an interconnect matrix must also be designed to support both Feistel networks (common to most block ciphers) as well as other forms of networks, such as Substitution-Permutation (SP) networks.

Although domain-specialized FPGAs can offer significant area, speed and power improvements over conventional reconfigurable devices, there are several unique and unexplored design problems that complicate their development. One source of these problems is that designers often opt to replace more universal, fine-grain logic elements with a specialized set of coarse-grain functional units to improve computation speed and reduce routing complexity. While this can greatly improve the performance of the system, developers need to closely consider the specific ways in which each algorithm uses the provided resources since the logical elements are no longer universally flexible.

Merely given a domain of applications it is not obvious what the most appropriate set of functional units would be, much less what routing architecture would be appropriate, what implications this might have on necessary CAD tools, or how any of these factors might affect each other. Simultaneously considering all applications in a domain and determining the most appropriate overall number and ratio of the different functional units is a very difficult task at best.

The selection of functional units can be subdivided into three steps. First, the applications in a domain must be analyzed to determine what operations they require. Crucial parts such as wide multipliers or fast adders should be identified. Next, this preliminary set of functional units can be distilled to a smaller set by capitalizing on potential overlap or partial reuse of other types of units. Different sizes of memories, for example, may be combined through the use of multi-mode addressing schemes. Lastly, based upon design constraints, the exact number of each type of unit in the array should be determined. For example, if the applications are memory intensive rather than computation-intensive, the relative number of memory units versus ALUs should reflect this. [12].

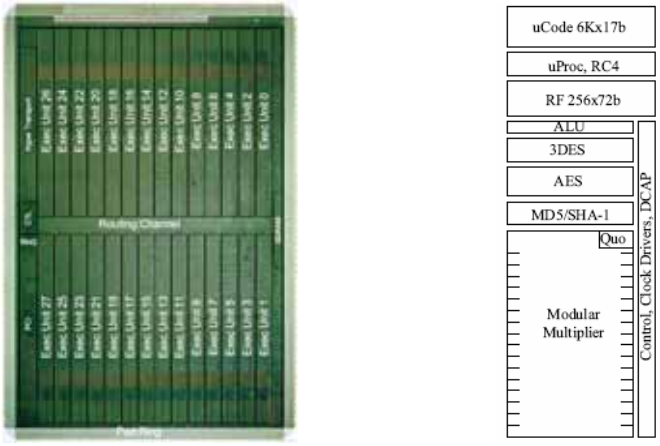
Our goal in this project was to explore the feasibility combining both secret key and public key encryption hardware into a common, domain specific reconfigurable design, given the potential benefits highlighted above for such a device.

This report is organized as follows. In section 2, previous work in similar direction is discussed and the significance of this work is explained in the context of previous efforts. Section 3 gives a highlight of our design approach and contribution while section 4 discusses the methodology used to develop the hardware architecture presented in this work. It also discusses the analysis of several symmetric key algorithms and RSA implementation used for sake of identifying a common architecture for supporting both secret and public key cryptography. In Section 5, the proposed design is discussed in detail. Section 6 discusses the implementation of RSA, Rijndael and Serpent on the proposed hardware architecture. In section 7, we present some remarks on the comparison of this work with similar works in terms of expected performance. This is followed by a description of the design implementation (programming) on our proposed hardware and finally we present the future extensions to this work.

2. RELATED WORK

2.1 CAVIUM NETWORKS' SSL & IPSEC PROTOCOL AWARE SECURITY PROCESSOR

The idea of using a single chip hardware solution for high speed, multi-protocol and multi-algorithm cryptographic processing has been addressed only recently in a few attempts. In [13], the authors have proposed an security processor shown in Figure 1 that contains 28 execution units, each of which has dedicated blocks for modular multiplication, AES, 3-DES, SHA, MD5 as well as a 16-bit RISC processor, code memory and a general purpose 64-bit ALU controlled by the RISC processor. The chip runs at 500 MHz and can achieve tens of Gbps rates for block cipher algorithms and 3-120 Kbps for various RSA key sizes from 512 bits to 2048 bits. The processor is reconfigurable only at the level of software by use of programmable RISC processor to implement protocol specific sequencing.



(a) (b)

Figure 1: (a) The overall chip layout showing multiple execution units (b) the structure of each functional unit showing the code memory, the microprocessor, the ALU, 3DES, AES, hashing units and the modulo multiplier.

2.2 USC MARK II 'S ADVANCED CRYPTOGRAPHIC ENGINE FOR IPSEC

In [11], the authors have exploited the benefits of reconfiguration of FPGA for designing a platform termed as adaptive cryptographic engine (ACE) that targets flexible but high speed implementation of IPSec cryptographic framework. It uses FPGA optimized implementations of various block cipher stored as FPGA bit configurations in an external to FPGA memory and loaded by a controller into the FPGA according as per demand. The author focuses on low latency key setup time so that as soon as the FPGA has been configured for a particular algorithm, processing can begin as soon as possible when the key is available.

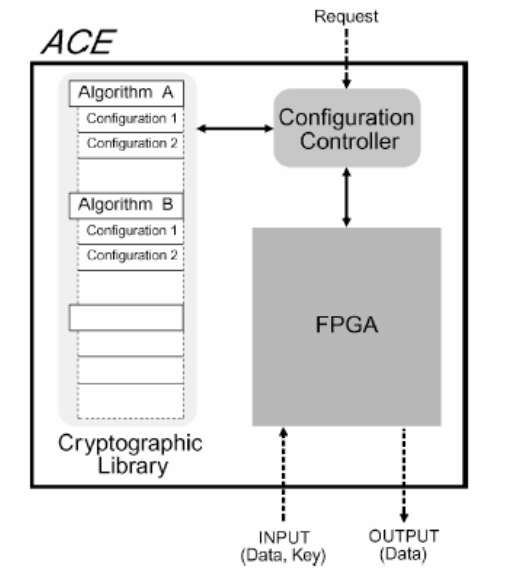


Figure 2: The architecture of ACE showing the memory storing various algorithm configurations, the FPGA and a controller that loads the appropriate configuration on demand

2.3 WORCESTER POLYTECHNIC INSTITUTE'S COBRA ARCHITECTURE

In [8], the author has designed a reconfigurable hardware architecture shown in Figure 3 for efficiently supporting block ciphers. It incorporates almost all the ideas mentioned in the previous section and the work is very similar to the one being presented in this report. The author has done a comprehensive analysis of the operations used in representative block cipher, with special focus on AES finalist algorithms. Also, the various common architectural techniques that can be used to implement the various block ciphers (e.g. loop unrolling, pipelining and sub-pipelining) achieving different area/throughput tradeoffs are also presented in detail. The author has come up with a hardware architecture that functions like a VLIW processor with computational elements that are macro blocks with configurable functionality. Inside each block, which the author calls Reconfigurable Logic Elements (RCE and RCE MUL for blocks with multiplication capability), all the required operations are implemented in a cascade fashion, with the output of each operation feeding the next and the order of operations optimally decided by observing the order of operations in large number of block ciphers.

2.4 OUR WORK

Although [8] has undertaken considerable work in providing the benefits of Reconfigurable Computing to Symmetric key ciphers, no one has to date attempted to develop a reconfigurable platform that combines both public-key and secret key algorithms. And while the ASIC developed by

[13] supports both Symmetric and Asymmetric key algorithms, it does not provide the flexibility associated with reconfigurability. Furthermore, their implementation of public-key cryptography gives extremely poor performance.

The primary aim of our project is not only to bring the advantages of reconfigurability to both cryptographic domains, but to do so without compromising efficiency or performance. Furthermore, by implementing both symmetric and asymmetric ciphers on a single chip, we introduce the potential security advantages of not storing any cryptographic keys off chip, where feasible, thereby dramatically reducing the risk of exposing secrets to unauthorized entities [6].

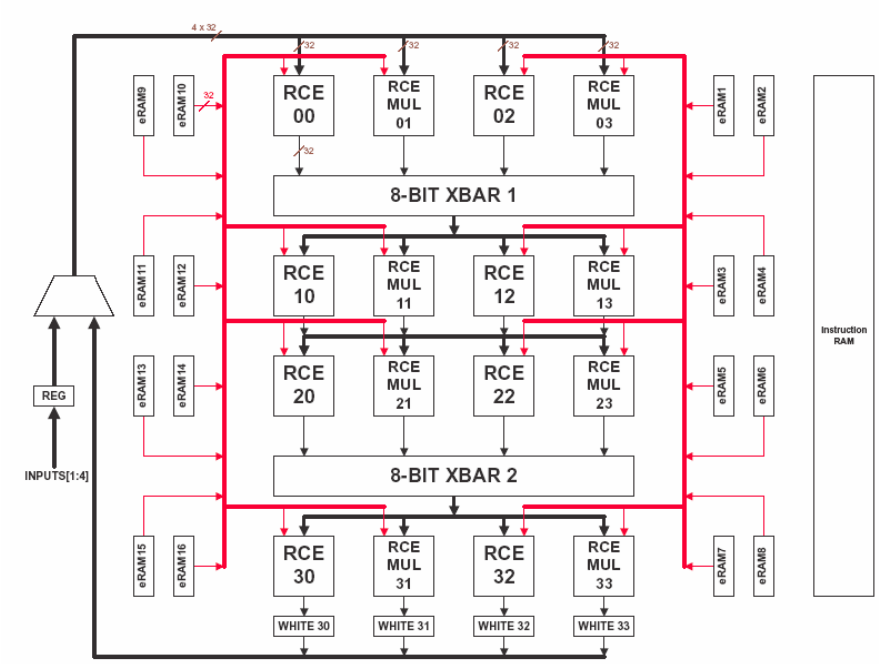


Figure 3: The COBRA architecture showing the linear pipeline structure and multifunctional macros RCE and RCE MUL connected with crossbars and RAM

3. WHAT WE HAVE DONE / OUR APPROACH

3.1 DEFINING A PROJECT STRATEGY

Our primary aim in this project was to explore the feasibility of developing a reconfigurable cryptographic core that is capable of supporting both secret and public-key encryption with an area efficient design, and without compromising performance. Justifications for this combination are given in Section 1. If such a combination was found to be feasible, we were to proceed with defining and detailing such an architecture.

However, proceeding to accomplish these rather ambitious goals in such a limited amount of time was easier said than done. In the very early stages of the project, we had identified the vital need for a systematic methodology and approach. We listed and categorized our tasks as follows:

- Literature Survey
  - Gain familiarity with popular symmetric key algorithms; study implementation techniques and options; identify any useful patterns.
  - Study various public-key algorithms; identify common and essential operations; evaluate options for implementation of said operations.
  - Study and evaluate methods and approaches for developing custom reconfigurable architectures for a specified application domain; developing an in-depth understanding of issues and tradeoffs associated with the design of such devices.
- Architecture Development
  - Developing and specifying our reconfigurable architecture in a bottom-up manner.
  - Evaluation and iterative refinement of our design, to ensure effective and efficient support for common functions needed by the various supported cryptographic algorithms.
- Validating our design by mapping several representative algorithms
  - Evaluating the suitability of our design for implementing different algorithms; assessment of both logic and interconnect in this regard.
- Identification of Programming Methodology

In the next few sections, we detail the work undertaken during the course of this project.

4. DESIGN METHODOLOGY

In order to begin with the task of designing reconfigurable hardware architecture specialized for both symmetric key as well as asymmetric key encryption it was initially required to undertake a broad study of contemporary public- and secret- key algorithms to:

- Acquire a familiarity with this specific application domain,
- Assess the various approaches/structures proposed in the literature for hardware implementation of complicated operations in each algorithm,
- Identify the essential ingredients of efficient/high-performance implementations, and
- Identify potential commonalities and basic hardware trends that are consistent between various block ciphers, and most importantly for our work, between secret-key and public-key ciphers, so that they may be exploited to achieve our objectives.

At a most general level but nonetheless important for area and timing performance, the hardware implementation of every cryptographic algorithm can be studied in terms of it's logic, interconnect and operational memory requirements. It was identified early on during the study that in order to develop a highly efficient design that is capable of implementing both public and secret key cryptography, we will need to identify similarities among algorithm implementations in their utilization patterns of all the three hardware resource categories, namely logic, interconnect and memory (RAM/ROM). This insight provided us with a starting point to handle the complexity of the task to be achieved.

With these goals in mind, our methodology was to explore cryptographic schemes in decreasing order of their expected degree of influence on our final architecture. Thus, while accessing the hardware resource requirements for implementing a particular algorithm in one of several possible manners, the algorithms were ordered by importance and then by relative hardware complexity, so that the architecture developed supports the most important algorithms most efficiently, and support for esoteric requirements of the less important ones may be added later on if feasible and/or necessary. The ordering we observed was:

- AES (Rijndael)
- DES
- Modular Exponentiation (RSA)
- Serpent
- Twofish
- RC6, MARS, and others

Since [8] has undertaken considerable work towards identifying common logic components and the various common architectural choices and their tradeoffs for symmetric key ciphers implementations, this work served as our primary resource for study and analysis of block cipher algorithms. The author has undertaken an analysis of a wide range of block ciphers in order to identify their specific core operations. The analysis is restricted to block ciphers that operate on plaintext block sizes of 64 and 128 bits as they are representative of algorithms in use that meet current and expected future security requirements. The following basic operations were identified (Tables 1 and 2):

1) Bitwise XOR, AND, OR.
2) Addition or subtraction modulo 2^n
3) Shift or rotation by a constant number of bits.
4) Data-dependent rotation by a variable number of bits.
5) Multiplication modulo the table entry value (bullet indicates constant multiplication).
6) Multiplication in the Galois field specified by the table entry value.
7) Inversion modulo the table entry value.
8) Look-up-table substitution (asterisk indicates key dependency).

Of the various operations listed above, our architecture efficiently supports operations 1-3, 5, 6 and 8. Presently, our architecture doesn't support data-dependent rotations (4). Consequently, algorithms requiring variable rotates are not supported; please see Section 6.4 for a discussion. Modulo inversion was not deemed a function with enough utility to be supported by the author. We too have therefore not considered this operation.

Algorithm	XOR AND OR	ADD SUB Mod	Fixed Shift	Var. Rot.	MUL Mod	GF Constant MUL	Inv. Mod	LUT	Int. Block Size
SHARK		2^{64}				$GF(2^9)$		8-to-8	8
SKIPJACK	•		•					8-to-8	8
IDEA	•	2^{16}			$2^{16}+1$				16
MacGuffin	•		•					6-to-2	16
RC2	•	2^{16}		•					16
Blowfish	•	2^{32}						8-to-32*	32
CAST	•							8-to-32	32
CAST-128	•	2^{32}		•				8-to-32	32
DES	•		•					6-to-4	32
FEAL	•	2^8	•						32
GOST	•	2^{32}	•					4-to-4	32
Khafre	•			•				8-to-32	32
Khufu	•			•				8-to-32*	32
LOKI91	•		•					12-to-8	32
RC5	•	2^{32}		•					32
TEA	•	2^{32}	•						32
ICE	•		•	•				10-to-8	32
MISTY1	•					$GF(2^7)$, $GF(2^9)$			32
MISTY2	•					$GF(2^7)$, $GF(2^9)$			32
WAKE	•	2^{32}	•					8-to-32	32
WiderWake	•	2^{32}	•					8-to-32	32
CS-Cipher	•		•					4-to-4	64
SAFER K	•	2^8	•					8-to-8	64

Table 1: Core operations for 64 bit block ciphers

Algorithm	XOR AND OR	ADD SUB Mod	Fixed Shift	Var. Rot.	MUL Mod	GF Constant MUL	Inv. Mod	LUT	Int. Block Size
FROG	•							8-to-8	8
SAFER+	•	2^8	•					8-to-8	8
SQUARE	•		•			GF(2^8)		8-to-8	8
E2	•		•		2^{32}		2^{32}	8-to-8	16
CAST-256	•	2^{32}		•				8-to-32	32
CRYPTON	•		•					8-to-8	32
MARS	•	2^{32}	•	•	2^{32}			8-to-32	32
MMB	•				2^{32} - 1*				32
RC6	•	2^{32}	•	•	2^{32}				32
Rijndael	•		•			GF(2^8)		8-to-8	32
Serpent	•		•					4-to-4	32
Twofish	•	2^{32}	•			GF(2^8)		8-to-8*	32
DEAL	•		•					6-to-4	64
DFC	•	2^{64}			2^{64}				64
Hasty Pudding	•	2^{64}	•		•			8-to-64, 3-to-64	64
LOKI97	•	2^{64}	•	•				13-to-8	64
Lucifer	•		•					4-to-4	128
MAGENTA	•					GF(2^8)			128

Table 2: Core operations for 128 bit block ciphers

In [8], although logic and memory requirements of symmetric key ciphers are discussed in detail, the work does not cover interconnect requirements in depth. Our study revealed that at a macro level, interconnect is designed to be a linear pipeline with a global feedback path to implement reuse of logic for implementing multiple rounds. However, implementation of each of the macro operations themselves may require very complex interactions of logic resources via interconnect. Examples include the Linear Transformation step in Serpent cipher, or the Mix-Column operation in Rijndael.

Further, we undertook a broad study of hardware implementations of modular multiplication and modular exponentiation. Modular multiplication is an integral component of virtually all public-key ciphers (elliptic curve, RSA, Diffie-Hellman), while modular exponentiation is the basis for RSA, which is the most popular of these, and Diffie-Hellman.

Several approaches were studied for implementing efficient and fast modular multiplication architecture, keeping in mind the commonality with the basic logic operations required by symmetric key ciphers. We finally settled upon what we found to be the fastest and simplest approach that is described in [21].

5. OUR DESIGN

Our design was primarily motivated by the following observations:

- Based on the study of the logic requirements of symmetric key algorithms, we decided to evaluate various implementation approaches for modular exponentiation. It was noted that modular exponentiation primarily requires fast adders.

- Our study revealed that vast majority of asymmetric key approaches utilize word-oriented operations. The carry save adder (CSA) based implementations were found to be more common as opposed to systolic array based implementations which are not word oriented [16,20] and the former provided the highest throughput for reasonable resource requirements [15,17,18,19,21]. As such we decided to focus on redundant representation (CSA based) modular exponentiation implementation.

- At this point, the key insight we had was that since a CSA is made up of 2 half adders with 1 OR gate and each half adder itself is 1 XOR and 1 AND, if we could add some configurability to the basic CSA we would be able to provide support for most of the primitive operations found in symmetric key ciphers. Meanwhile, at the same time, our goal of achieving a fast and efficient design would be facilitated by use of simple and fast elements (basic gates) as opposed to more general purpose but slower elements (LUTs in FPGAs).

- The additional elements needed to make the design complete in it's support for both types of ciphers were shifts between rounds of addition (for our chosen modulo exponentiation architecture support), support for fixed length shifts, rotates and arbitrary permutes of 32-bit operands (for symmetric key algorithm implementations). To support all these operations, we decided to incorporate a general purpose permute structure.

- We then initiated an iterative fine tuning of the architecture for supporting each symmetric key algorithm in order of their perceived significance as mentioned in section 4.

Given below is a detailed description of the basic unit we have proposed:

- The structure of a full adder (also called a carry save adder unit) is shown in Figure 4. It can interpreted as being made up of two half adders (shown as two boxes) and a single OR gate.

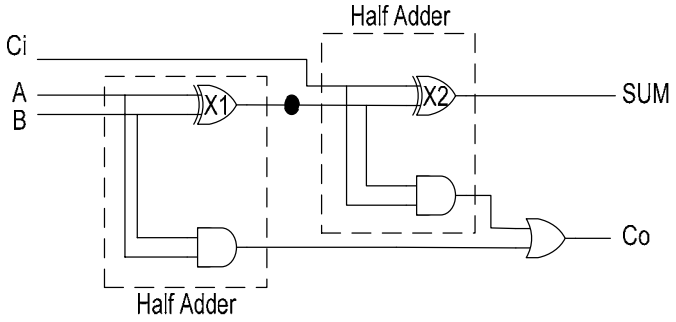


Figure 4: A carry save adder unit made from two half adders and an OR gate.

- A single Carry Save Adder (CSA) may be used to implement logical-XOR, AND as well as OR operations with the inclusion of multiplexing circuitry at the point marked with a solid circle in Figure 4. The MUX will select between either the A⊕B or another external input (not shown), say D, as an input to the XOR gate X2. Also, the outputs of the two half adders can be made available for use separately.

- A bit permutation unit will be absolutely necessary to support the random bit permutes used in DES, as well as the various algorithm dependent fixed-width rotate and shift operations,

- A certain number of configuration bits are required to configure the operation of (programmable multiplexers present in) each CSA. The overhead associated with adding this configurability to a CSA can be alleviated if the all CSAs in a group are programmed using a common configuration memory store instead of independently, thereby reducing the amount of configuration memory required, although flexibility is reduced to some extent. Conversely, one must also assess the amount of required flexibility to minimize overhead, without sacrificing efficiency.

- The basic unit of our design is referred to as a Cell, and consists of a modified CSA unit as shown in Figure 5.

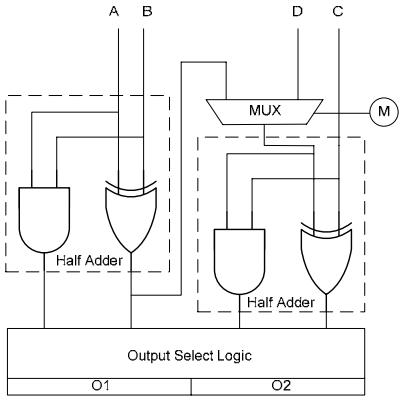


Figure 5: The structure of the basic element called a cell consisting of two half adders and outputs select logic

- A total of 64 such cells are clustered together in a single unit called a Block as shown in Figure 6. Aside from the basic CSA functionality, this block is able to

provide logical XOR/AND operations as follows: 1 x 128-bit operation i.e. <A|C>⊕<B|D>; 2 x 64-bit operations i.e. A⊕B, C⊕D, or 4 x 32-bit operations i.e. by sub-dividing each of the 64-bits input ports (i.e. A, B, C, or D) into two 32-bits groups (see Figure 2). OR functionality is also provided, although it is limited to 1 x 64 and 2 x 32 bit operations (we get A+D if we set B = C = 1). The entire block may also be configured as a ripple carry adder.

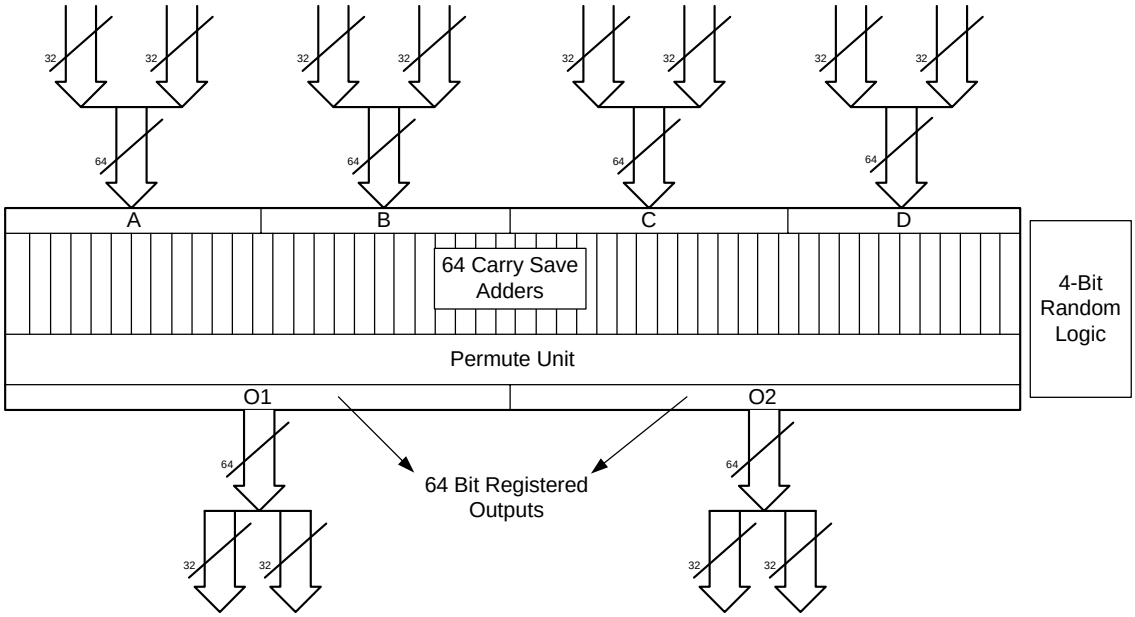


Figure 6: The structure of a block which is a grouping of 64cells, a permute unit, a block for random logic implementation and output select logic

- As shown in Figure 6, the 4 inputs ports A, B, C, D and the two output ports O_1 and O_2 are each 64 bits wide, but the interconnect buses feeding data between two blocks are 32-bits wide. This means that each of the 64-bit input and output ports use 2 x 32-bit busses for data transfers.

- The 4 input ports themselves are configurable in that each of them may be set to either 0, or 1, as well as the incoming 64-bit value may be ANDed with a single bit value set to either 1, or 0 (This functionality is specially required for modular multiplication architecture).

- The decision to have 128 bit wide clusters of logic operations, while having 32-bit wide *interconnect* flexibility is based on direct observations of the basic requirements of various symmetric key ciphers: AES requires 128 bit XOR operations, but also handles 32-bit chunks of data in the shift-row and Mix Column operations. Furthermore algorithms like DES and Serpent also operate on 32-bit blocks of data.

- The outputs of the Block/cluster are fed into a "permute unit" that is able to perform limited bit permutation operations on the outputs of the 64 CSA cells before passing them onto the output registers, or to the inputs of the next Block.

- At present, the permute unit is assumed to have a structure similar to the routing tracks of a Row based FPGA, although with much greater track density (minimum 32 tracks to fully support 32-bit rotate operations i.e. 16-bits of movement in either direction)

- Although not as flexible as a full crossbar, this kind of permute unit will not be as expensive either. The permute unit as perceived should be able to support fixed shifts of 128 bit values, fixed rotates of 32-bit values, and other limited random permutations. Those permutation operations that cannot be supported by a single such permute unit, may be performed across multiple blocks.

- The 128 bit outputs of each block may be optionally registered, again selectable in groups of 32. The registers are also connected in the form of a 128-bit Shift-Register chain.

- Considering that the XOR and AND operations have been grouped into 128 bit wide block, it would be inefficient to use these blocks to implement the limited number of single bit generation functions which are needed for modular multiplication. Thus, some structure (such as a 4-LUT) must be available along with the 128 bit wide blocks for implementing a limited amount of random logic to generate these bit signals.

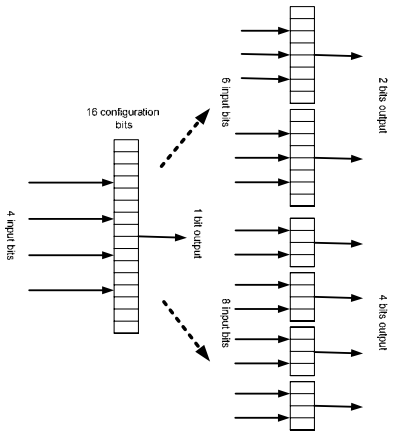


Figure 7: The inside structure of the random logic block shown in Figure 6

- Consequently, there is a LUT logic resource associated with each block as shown in Figure 4 for random logic implementation. Random logic block inputs are provided in the form of 4 bit lines from all the input ports side (one from each port), 4 bit inputs from the permute unit, thus a total of 8 distinct inputs, and finally there are 4 global output bit lines that reach all blocks.

- As shown in Figure 4, the random logic block is able to implement 1 x 4-input function, 2 x 3-input functions, and 4 x 2-input functions, and thus is able to produce and broadcast on as many as all 4 global bit lines.

Given below is the organization proposed for our overall configurable architecture:

- The basic blocks are organized into groups of 5, with an associated, configurable memory bank as shown in Figure 8.

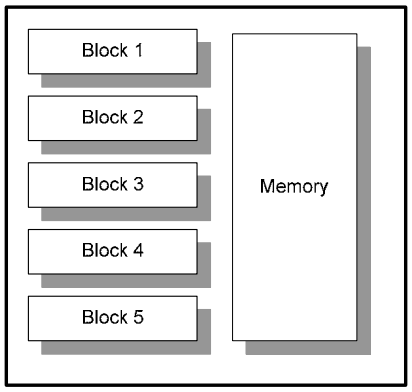


Figure 8: The clustering of 5 blocks and associated memory (RAM) called a Group.

- Interconnect within this group is at present completely flexible, and takes the form of a cross-bar as shown in Figure 9. A possible future extension is that, after a more thorough evaluation of all options, the flexibility available may be reduced to conserve area, so long as applicability of this organization to the encryption domain remains unconstrained.

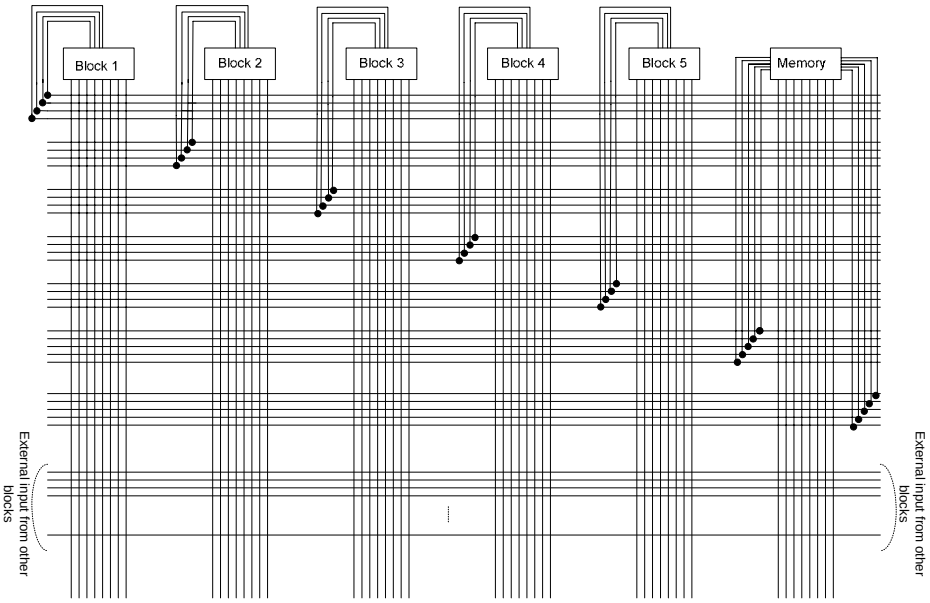


Figure 9: The cross bar structure for interconnect among blocks and memory in a group.

- The Memory block is configurable and its capacity is set to 2Kbits at present, as that is the largest size of any single S-box in any symmetric-key cipher.

- After the group level, we add a final level of hierarchy as shown in Figure 10. Each of our groups as defined above can be arranged into a grid as shown. At this level, our evaluation has shown that – with the exception of the global broadcast bit lines – only 128-bit nearest neighbor (NN) connections (8-NN) are required between blocks. Thus the performance scalability of our approach is only limited for applications that make use of these bit lines.

- We are currently exploring strategies to effectively pipeline the distribution of global single bit signals, as a means of sustaining scalability and high clock frequencies even for designs that use these lines. The expected cost would be an increase in pipeline latency.

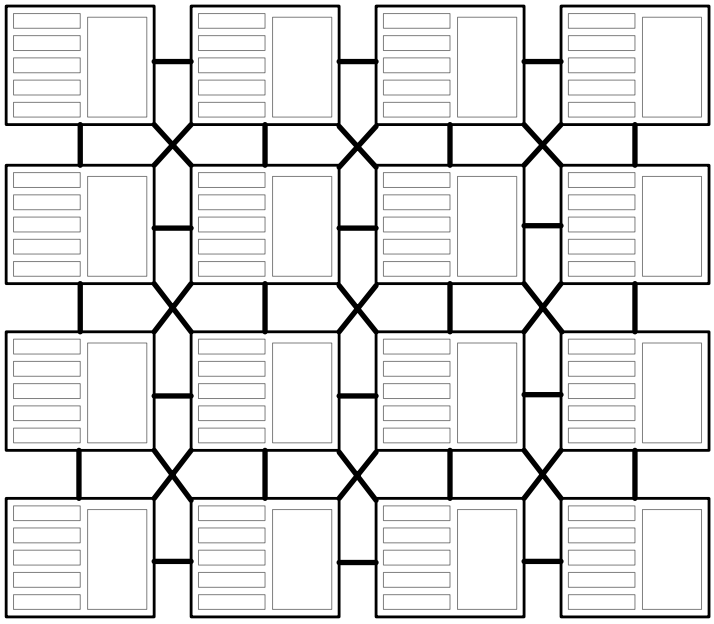


Figure 10: The block diagram of the overall architecture showing the interconnect structure among groups

6. CRYPTOGRAPHIC ALGORITHMS AND THEIR IMPLEMENTATIONS

6.1 PUBLIC-KEY ENCRYPTION AND MODULAR MULTIPLICATION

The basic algorithm for the computation of Modular Exponentiation, known as the Square and Multiply Algorithm, is shown in Figure 11. As can be seen it is based on repeatedly performing Modular Multiplication. The most efficient method for performing this operation is the Montgomery Multiplication algorithm, which is shown in Figure 12.

The algorithm in Figure 12 describes how Modular Multiplication of radix 2^k may be performed. If k=1, the algorithm performs radix 2 Montgomery Multiplication.

Considering the importance of Modular Multiplication to public key encryption, our primary approach for incorporating support for these schemes was to focus on efficiently implementing this

1. $P_0 = 1,\ Z_0 = X$
2. $FOR\ i = 0\ to\ n-1\ DO$
3. $Z_{i+1} = Z_i^2 \bmod M$
4. $IF\ e_i = 1\ THEN\ P_{i+1} = P_i \cdot Z_i \bmod M$
   $ELSE \qquad P_{i+1} = P_i$
5. $END\ FOR$

Algorithm *computes* $P = X^E \bmod M$, *where* $E = \sum_{i=0}^{n-1} e_i 2^i,\ e_i \in \{0, 1\}$

Figure 11: The basic algorithm for computation of modular exponentiation known as the square and multiply algorithm

Montgomery Modular Multiplication for computing $A \cdot B \bmod M$,
where $M = \sum_{i=0}^{m-3} (2^k)^i m_i,\ m_i \in \{0, 1 \ldots 2^k - 1\}$;
$\tilde{M} = (M' \bmod 2^k) M,\ \tilde{M} = \sum_{i=0}^{m-2} (2^k)^i \tilde{m}_i,\ \tilde{m}_i \in \{0, 1 \ldots 2^k - 1\}$;
$B = \sum_{i=0}^{m-1} (2^k)^i b_i,\ b_i \in \{0, 1 \ldots 2^k - 1\}$;
$A = \sum_{i=0}^{m-1} (2^k)^i a_i,\ a_i \in \{0, 1 \ldots 2^k - 1\}$;
$A, B < 2\tilde{M};\ 4\tilde{M} < 2^{km};\ M' = -M^{-1} \bmod 2^k$

1. $S_0 = 0$
2. $FOR\ i = 0\ to\ m-1\ DO$
3. $q_i = (S_i + a_i B) \bmod 2^k$
4. $S_{i+1} = (S_i + q_i \tilde{M} + a_i B)/2^k$
5. $END\ FOR$

Figure 12: The Montgomery modular multiplication algorithm

operation. To this end, we searched for efficient hardware implementations that were not only high performance, but also had a simple architecture. The emphasis on the latter is due to the fact that ours being a reconfigurable architecture, it would be difficult to support hardware constructs that are largely heterogeneous in their design and/or complicated.

Applying Montgomery's algorithm, the cost of a modular exponentiation is reduced to a series of additions of very long integers. To avoid the carry propagation in multiplication/addition architectures several solutions have been proposed in the literature. They either use Montgomery's algorithm in combination with a redundant representation (e.g. carry-save) [15, 17, 18, 19, 21], or by utilizing RNS representation [14], or base their designs on systolic arrays [16, 20].

After perusal of several promising candidates, we finally settled on the implementation style presented in [21] for Modular Exponentiation. This paper advocates an approach that implements the entire modular exponentiation scheme in Carry Save representation, thereby avoiding all the carry propagation delays associated with long integer addition. The algorithm is given in Figure 13.

As can be seen in Figure 13, the algorithm performs long integer-addition with carry rippling only at the very end of the Exponentiation operation, while all Modular multiplications are performed in

Carry Save Representation. The algorithm for modular multiplication using Montgomery's technique is given in Figure 14.

Five-to-two Multiplier Modular Exponentiation (P, E, M)

K = 2^{2k} mod M … computed externally

1. **$P1_0$, $P2_0$ = 5to2_MontMult(K , 0 , 1 , 0 , M) ,**
   $Z1_0$, $Z2_0$ = 5to2_MontMult(K , 0 , P , 0 , M);
2. **FOR i = 0 to n-1 DO**
3. **$Z1_{i+1}$, $Z2_{i+1}$ = 5to2_MontMult($Z1_i$, $Z2_i$, $Z1_i$, $Z2_i$, M)**
4. **IF e_i = 1 THEN**
   $P1_{i+1}$, $P2_{i+1}$ = 5to2_MontMult($P1_i$, $P2_i$, $Z1_i$, $Z2_i$, M)
   ELSE
   $P1_{i+1}$, $P2_{i+1}$ = $P1_i$, $P2_i$
5. **ENDFOR**
6. **$P1_n$, $P2_n$ = 5to2_MontMult(1 , 0 , $P1_{n-1}$, $P2_{n-1}$, M)**
7. **P = $P1_n$ + $P2_n$**
8. **RETURN P**

Algorithm *computes* $P = X^E \bmod M$, *where* $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0, 1\}$

Figure 13: The redundant representation based 5-to-2 multiplier modulo exponentiation

Five-to-two CSA Montgomery Multiplication (A1 , A2 , B1 , B2 , M)

1. **$S1_0$, $S2_0$ = 0 , 0**
2. **FOR i = 0 to m-1 DO**
3. **q_i = [($S1_i$ + $S2_i$) + A_i*(B1+B2)] mod 2**
4. **$S1_{i+1}$, $S2_{i+1}$ = CSR [($S1_i$ + $S2_i$) + A_i*(B1+B2) + q_i*M] div 2**
5. **ENDFOR**

Figure 14: The 5-to-2 Montgomery Multiplication algorithm using carry save representation

Figure 15(a) shows the logic diagram that implements the multiplication scheme in Figure 14. The A_i bit is produced by the combination of the two parts of A: A1 and A2, which is done in parallel by the structure shown in Figure 15(b). This way, no more than a single Full Adder delay is found on the critical path.

As can be seen from the diagram, implementation of Modular Exponentiation in this way only requires extremely simple Logic (CSA), interconnect, and Memory (for buffering input operands only). The generation of q_i utilizes only minimal random logic. This implementation style maps extremely well to our architecture described in the last chapter, as shown in Figure 16 and described as follows.

Once the 1024-bit carry save additions specified in the above algorithm are divided into 16 64-bit chunks, each 64-bit CSA maps directly onto our basic block. Each of the 16 identical 5-to-2 compressors may be implemented in a single group, utilizing only 3 of the 5 available blocks in a group, with the appropriate operands such as A, B, and M being provided by the memory block in that group. The last of the three blocks in each group will have its outputs registered.

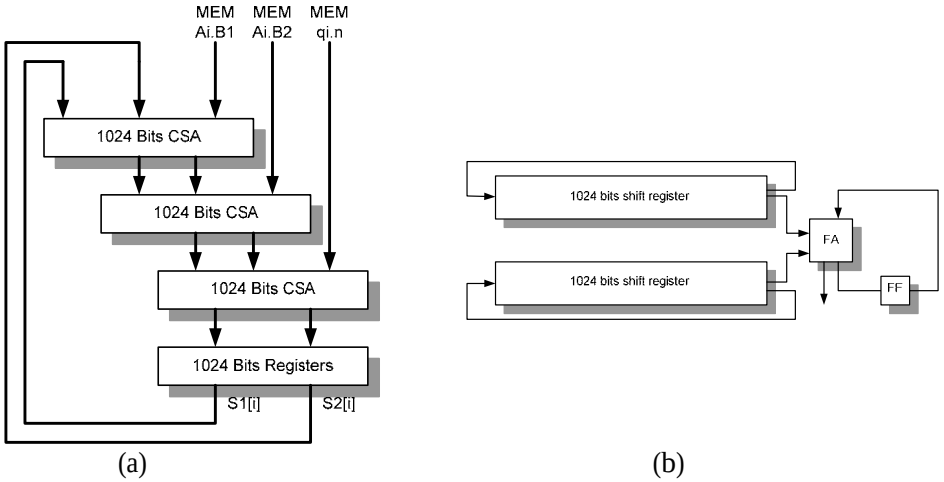


Figure 15: (a) The block diagram implementing the Montgomery multiplication in 5-to-2 carry save representation (b) The block diagram of structure that generates bit Ai each cycle

Step 4 of the algorithm also involves a left-shift operation of the output of the 5-2 compressor. In the original implementation [21], this is achieved through customized interconnect. In our design however, this shift is performed by the permute unit of the last block.

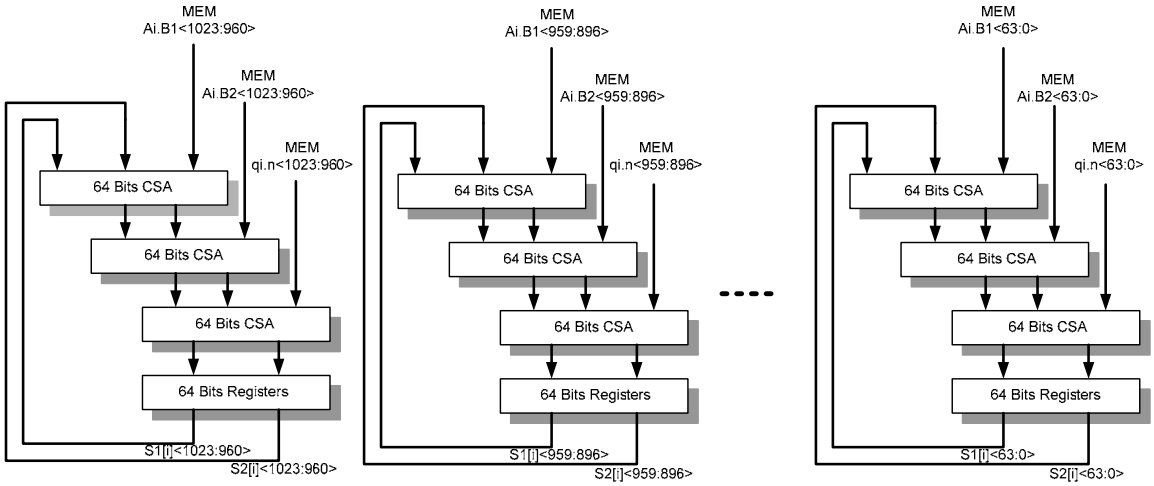


Figure 16: The implementation of Montgomery multiplication structure of Figure 15(a) when mapped to the proposed architecture

Generation of the q_i bit occurs at the random-logic unit associated with the second block of the least-significant 5-2 compressor. This bit then needs to be broadcast to all of the 16 groups. Generation of the A_i bit will utilize the 4th and 5th blocks of one of the groups (64 bits at a time instead of 1024 bits at once) as shown in Figure 16. Again, this bit is broadcast to all groups.

This implementation of Modular Exponentiation has claimed to provide the fastest encryption/decryption throughput to date. This is an obvious result of the inherent simplicity and parallelism in this approach. The only point of concern is perhaps the potential performance and/or scalability bottleneck that would be caused by the high fan-out bit broadcast lines.

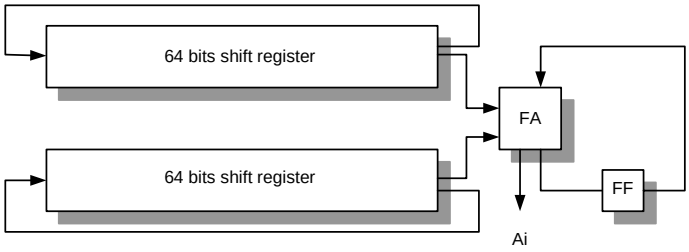


Figure 17: The generation of Ai by using 2 blocks only and considering 64 bits of A1 and A2 at a time

6.2 RIJNDAEL

The choice of Rijndael as an AES candidate algorithm finalist (and as the Advanced Encryption Algorithm) was due to the algorithm's fast key setup, low memory requirements, straightforward design, and ability to take advantage of parallel processing.

The Figure 18 details the Rijndael round structure. For the first nine rounds, the round input passes through the Byte-Sub, Shift-Row, Mix-Column, and Add-Round-Key functions. For round ten, the final round, the Mix-Column function is eliminated.

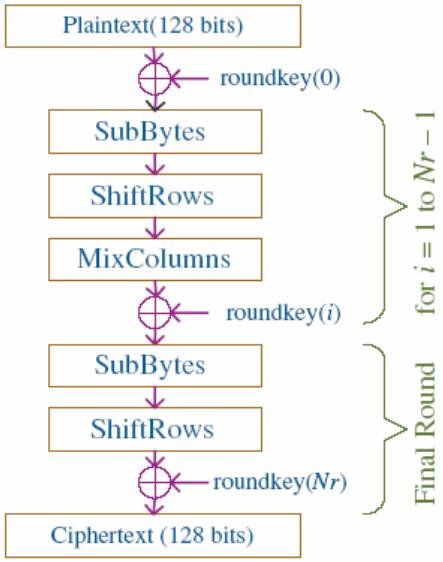


Figure 18: Encryption structure of the Rijndael Algorithm.

6.2.1 MAPPING TO OUR ARCHITECTURE

The 128-bit data block that is taken as input can first be represented in row-major format, as shown in Figure 19. This row-wise grouping greatly facilitates in our implementation of the major Rijndael operations.

Implementation of the Add-Round-Key, Byte-Sub, and Shift-Row are relatively simple using the proposed hardware blocks and memory; however the implementation of the Mix-Column operations

is non-trivial [23]. Since each of our basic-blocks is capable of performing a 128 bit XOR operation, Round-Key-Addition is a trivial operation.

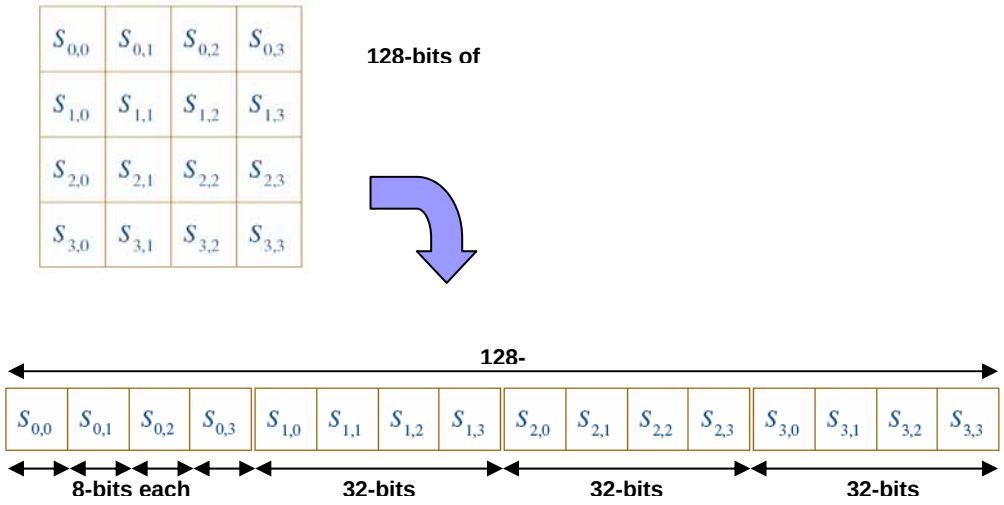


Figure 19: Row-major representation of the 128-bit input data block

The Byte-Sub operation requires none of the basic-blocks, but in order to enable parallel lookup for each of the 16 bytes in a 128-bit block of data, all of the available memory blocks from each group must be utilized. Shift-Row is again a trivial operation, as it requires a 32-bit rotate operation, which may also be seen as an arbitrary 32-bit permute. As shown in the figure above, each block of 4 bytes represents a row. It can easily be handled for the entire 128-bit data by a single basic-block.

The Mix-Column operation has the dataflow shown in Figure 20. The operation identified in Figure 21 forms a common operation in the Mix Column operation and has been termed as the xtime() function by the authors of the algorithm. The Mix-Column operation can be expressed in terms of xtime() function.

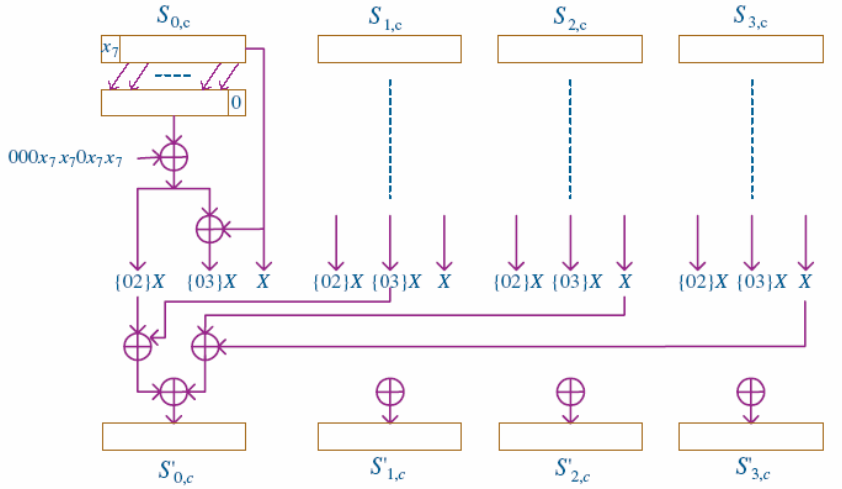


Figure 20: The straight forward implementation of the Mix-Column operation

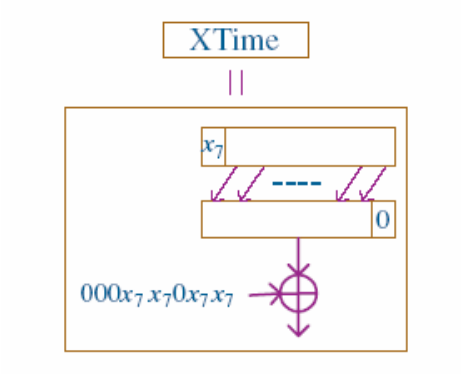


Figure 21: The structure termed as the xtime() function in Mix Column computation.

In order to implement X-time for each of the 16 bytes using only 2 blocks, we have implemented the operation as shown in Figure 22. The two blocks are needed as follows: the first block is used to generate the bit level masks for each of the bytes (i.e. $0000x_7x_70x_7$ for each byte, forming a 128-bit mask), while the second block performs the actual modified X-time operation: first the XORing at the logic level, followed by the left-shift of each byte and the LSB being replaced by x_7 of the appropriate byte, all of which is done by the permute-unit of the same basic block.

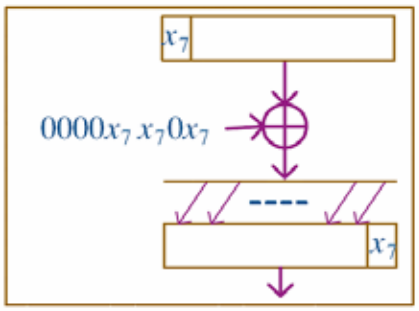


Figure 22: The modified xtime() function, for efficient implementation on our architecture

As we can see, implementation of the X-time function is followed by 4 XOR operations. Since our Blocks are 128 bits wide, once X-time is implemented for each of the bytes in the 128-bit input, the 4-XOR operations may be performed by 4 basic-blocks. Notice that due to the flexibility provided by the 32-bit inter-block interconnect, combined with the row-major ordering of data, we are able to perform the Mix-column operation on all of the columns at the same time (Figure 23).

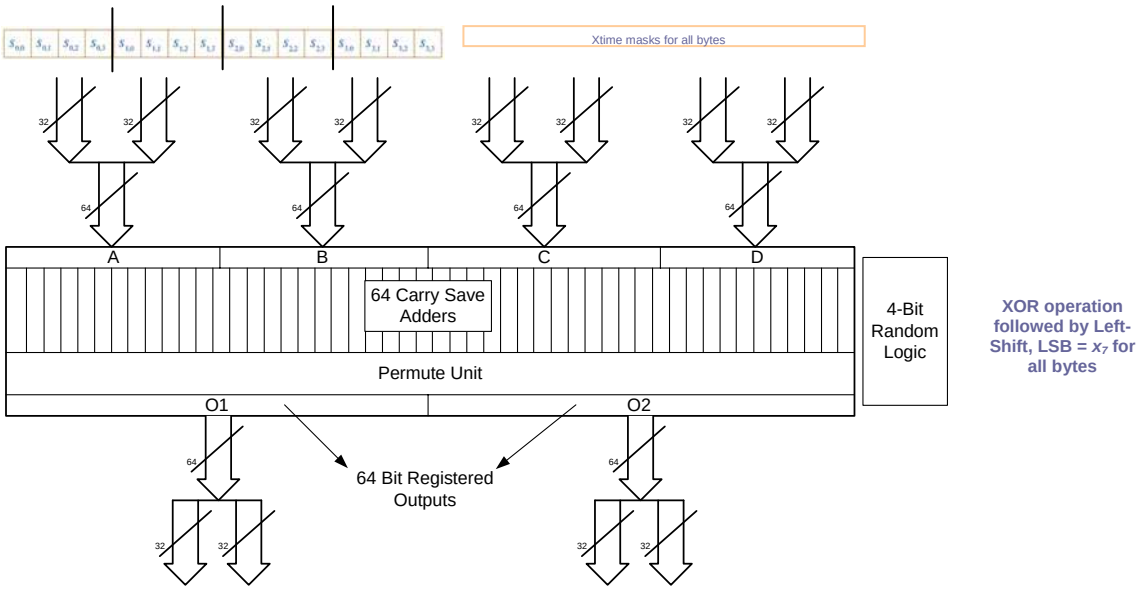


Figure 23: Mix-Column operation is performed in parallel on each column. This diagram shows implementation of xtime() using our basic block

Thus a total of 8 CSA basic-blocks and 16 Memory blocks are needed to implement a single round of Rijndael. Unfortunately, our basic design consists of 80 CSA blocks and only 16 Memory blocks. Thus even though we have more than 90% of our logic resources unused, we are unable to implement more rounds as we have run out of memory blocks, each of which is 2K bits!!

Due to this memory intensive nature, it becomes difficult to have a fully unrolled implementation of Rijndael, as we run out of memory blocks after implementing only one round. There are two options around this dilemma:

- Incorporate more memory (at least 10 times more for best utilization of resources!): this has the problem of deciding where to place the memory or whether to distribute it uniformly, without complicating routing and violating S-box boundaries (i.e. if memory is distributed uniformly, each memory-block should be able to hold multiple complete copies of the S-boxes).

- Implement memory-less version of AES. This entails the replacement of the memory S-boxes with non-linear computation: finding the Galois field GF(28) multiplicative inverse of each byte, followed by the application of an Affine transformation over GF(2).

Of these two approaches, the second seems more efficient if realized within the current hardware constraints, and is currently being explored.

6.3 SERPENT

Serpent is a substitution-permutation cipher comprised of key mixing, S-Box substitution, and a linear transformation. Serpent iterates over thirty two rounds with the final round performing an additional key mixing in place of the linear transformation.

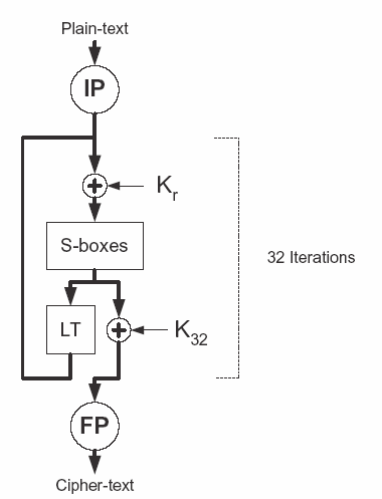


Figure 24: The block diagram of the Serpent block cipher

The Serpent S-Boxes are 4-bit x 4-bit fixed look-up-tables. For a given round, thirty two copies of the same S-Box are applied to the 128-bit input. There are a total of eight distinct S-Boxes, S0 through S7. Round 0 uses S0, Round 1 uses S1, etc. S0 is used again in Rounds 8, 16, and 24, resulting in each S-Box being used four times. A single S-box holds 64-bits, and 32 copies of a single S-box are required per round. This makes for a total of 2048 bits of storage.

Given in Figure 24 is the general flow of the Serpent cipher, as well as the details of the Linear Transformation step shown in Figure 25. As can be seen, a 128-bit word is divided into 32-bit sub-words, and each is subjected to several XORing and permutation operations. Due to the flexibility of our architecture, and the simplicity of the basic operations involved, the Serpent cipher is ideally suited to our design.

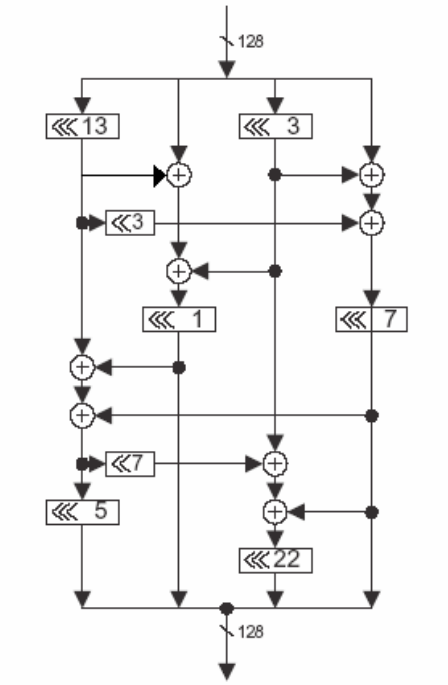


Figure 25: The Linear Transformation Step in Serpent

We see that a total of 8x32-bit XORing operations are required, as well as 8x32-bit permutations. These can easily be implemented using 4 of our CSA blocks: since each of our blocks is capable of performing 4 independent 32-bit XOR operations, as well as 4 independent 32-bit permute (fixed shift and rotate) operations. Given that the inter-block routing within a group is a full crossbar, implementing interconnect for such a compact design would be trivial.

The key mixing stage requires one more basic-block per group, while the S-Box lookup utilizes the available Memory block. Thus every round of Serpent can effectively fit in a single group. We are able to unroll the Serpent cipher for 16 rounds, with the present size of our design.

6.4 OTHER BLOCK CIPHERS: TWOFISH, RC6 AND MARS

6.4.1 TWOFISH

Twofish round function has a rather complicated structure in terms of interconnection of sub-components making up the round logic as shown in Figure 26. It also uses key dependent S-boxes shown in Figure 27 which also increases the complexity of its round function logic. However, the logic and interconnect resources required by the algorithm are supported by our proposed architecture. As can be seen in table 2, Twofish requires operations such as XOR, Mod 2^{32} addition, fixed shifts, S-box look-ups and Galois field 2^8 multiplication which can be implemented on the proposed architecture. We are not including implementation details of Twofish on the proposed architecture here as work is still ongoing for achieving a mapping. Our preliminary analysis hasn't revealed any hard constraints with regards to achieving a mapping of Twofish on the proposed architecture.

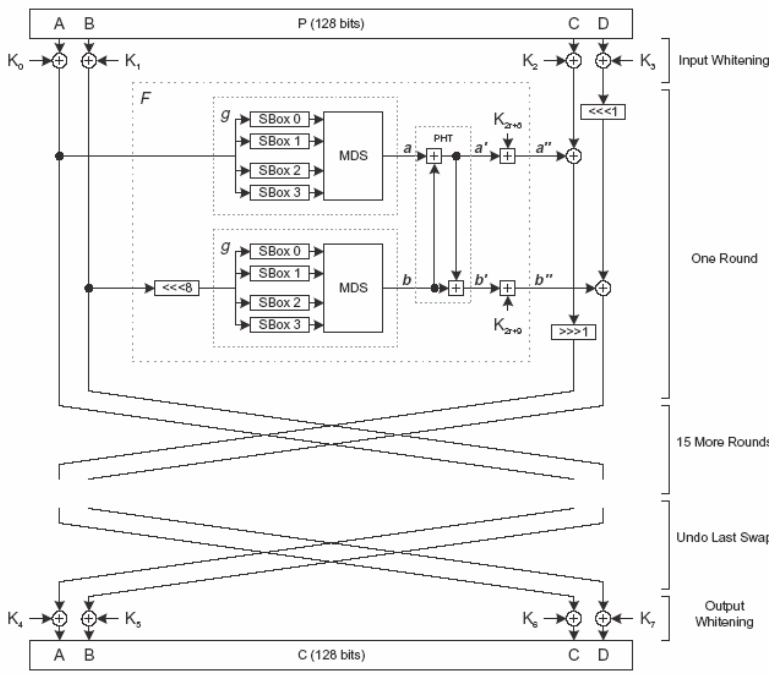


Figure 26: The overall structure of the Twofish block cipher showing the details of component operations in a single round. MDS implies Maximum Distance Separable and is actually Galois field multiplication similar to Mix-Column in Rijndael. The S-box structure is shown in Figure XX.

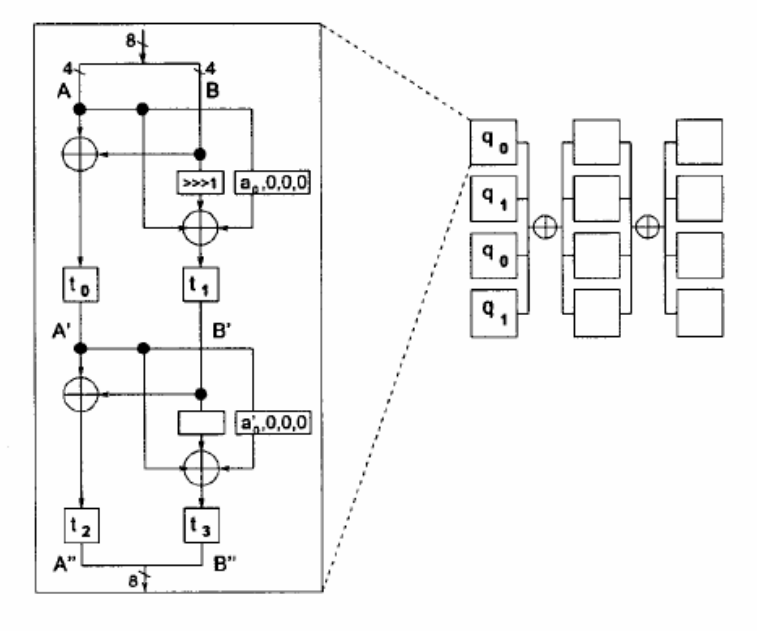


Figure 27: The structure of the Twofish key dependent S-box.

6.4.2 RC6 AND MARS

Both RC6 and MARS involve data dependent rotations and multiplication modulo 2^{32} in their round operations. The proposed architecture doesn't support data dependent rotations. However, the support of data dependent rotations may be added by use of barrel shifter in each Block. The reason for not adding this functionality is the large area overhead resulting from adding a barrel shifter in each of the blocks, though a single barrel shifter doesn't require a large area. But, due to required flexibility and ease of mapping, it wasn't clear as to which blocks to selectively complement with barrel shifting logic. Besides data dependent rotations, the multiplication operation also introduces significant area overhead when mapped to the architecture because of large resource requirements. Our study revealed that these similar problems were also faced by other authors when implementing RC6 and MARS in reconfigurable devices such as FPGAs. It has been shown that due to the need for data dependent rotations and multiplier implementations, the resulting implementation of these algorithms do not achieve efficiency compared to other AES finalist algorithms when implementing in reconfigurable hardware [8, 11, 24, 25]. This is one of the main reasons why these algorithms haven't been popular candidates for AES finalist selection criteria [26, 27]. As such, due to the lesser significance of these algorithms with regard to reconfigurable hardware implementations, we did not include the support of data dependent rotations and their specialized multiplication with the benefits of simplifying our proposed architecture.

7. DISCUSSION AND COMPARISON WITH RELATED WORK

7.1 COMPARISON WITH ACE [11] AND OTHER FPGA BASED IMPLEMENTATIONS

As discussed in Section 1.3.5, custom reconfigurable architectures naturally have the potential for providing a throughput and area-efficiency advantage over conventional FPGA based designs.

However, this obviously is not guaranteed, and depends on how well the various design issues associated with custom reconfigurable architectures (see Section 1.3.6) are handled.

Since at present, our design is still on paper and thus untested, we are unable to give empirical evidence of the expected benefits of our design over conventional FPGA based implementations. However, it is possible to present a logical framework for comparison of individual features of both architectures, and through deductive reasoning, identify what possible advantages one approach may have over the other, assuming all other factors normalized.

7.1.1 AREA EFFICIENCY

We expect our design to provide better area efficiency than FPGA based implementations because of the following reasons:

- Our basic functional unit consists of Full-adders implemented using actual logic gates, where as FPGA basic blocks consist of general-purpose Look-up tables that require considerably more area to implement (LUTs consist of Multiplexers and storage elements)
- Each functional unit of our design is special-purpose, and thus has limited flexibility (i.e. it has very few distinct modes of operation), where as FPGA Logic Blocks are inherently general-purpose, and thus require far more configuration information than our units.
- Our basic functional unit, the configurable Carry Save Adder is clustered into groups of 64, that may only be configured together to perform identical functions. Furthermore, our inter block interconnect is grouped into busses of 32-bits each. Each FPGA Logic Block and interconnect switch on the other hand is programmed independently of the others. This results in the need for considerably more on-chip configuration SRAM cells in FPGAs.

These disadvantages in FPGAs are a direct result of their general-purpose nature. However, this does not in any way imply that it is safe to assume our design to be area efficient: thus far, we have only considered the impact of Logic resources on area. For a thorough evaluation of area-efficiency of our design, the area that will be occupied by interconnect and routing also needs to be considered. In our design, we make extensive use of complex routing resources in the form of our permutation unit, and also the inter-block routing crossbar. The permute unit is similar to the routing tracks found in FPGAs, but the inter-block routing crossbar would require considerable silicon real-estate to implement. After an implementation and thorough analysis is undertaken, we may find the need to trim down our inter-block interconnect. Such a move though would not be unusual, considering that this is the very first design iteration for our architecture.

7.1.2 RELATIVE PERFORMANCE

Compared to FPGA architectures, we expect to provide better performance because of the following reasons:

- Use of basic gates instead of LUTs for implementation of logic functions – basic logic gates are expected to have much lower latencies, and therefore may be capable of sustaining higher operating frequencies.

- Inter-block interconnect is bus based, has fewer routing switches, and thus much lower RC delays are expected.

- Interconnection in general is hierarchically organized, thus aside from the bit-broadcast lines, there are no long wires, as in FPGAs. Also, design layout becomes uniform and predictable, providing further room for improving performance.

- Both configurable logic and interconnect are clustered, thereby reducing the amount of configuration data required. This in turn reduces configuration time..

As can be seen from the above points, we may note that although our design dedicates considerable amount of area to interconnect, this interconnect may be crucial to sustaining performance. Thus we face the classic area-performance tradeoff. The key measure of success for us however will be whether our cost-performance tradeoff is more suitable for implementing cryptosystems than other implementations.

7.2 COMPARISON WITH COBRA [8] RECONFIGURABLE ARCHITECTURE

The COBRA architecture described in [8] is a custom reconfigurable architecture designed for supporting symmetric-key cryptography. Figure 28 shows the structure of the basic configurable unit.

There are 16 such elements in the entire design, along with a simple top-down interconnect scheme and two byte-wise cross-bars. Additional Instruction RAM and key-storage memory is also provided, along with some control-logic and interconnect.

As we can see, there is a considerable amount of dedicated logic within each RCE: dedicated modulo multipliers/squarers, shift/rotate units, CF(2^m) Constant Multipliers etc.. At present, our design also possesses 16 configurable units, each with 5 basic-blocks, and roughly 2kbits of memory. However, instead of multiple dedicated logic blocks, our design has 5 identical configurable blocks.

The COBRA intra-block interconnect is linear and passes through every logic unit. When a unit need not be used, it simply forwards the data value to the next unit in line. This inherent inflexibility in interconnect is compensated for by increasing the number of logic blocks of each type in the RCE. For example there are 3 shift/rotate units, three Logic units, and 2 adders per block.

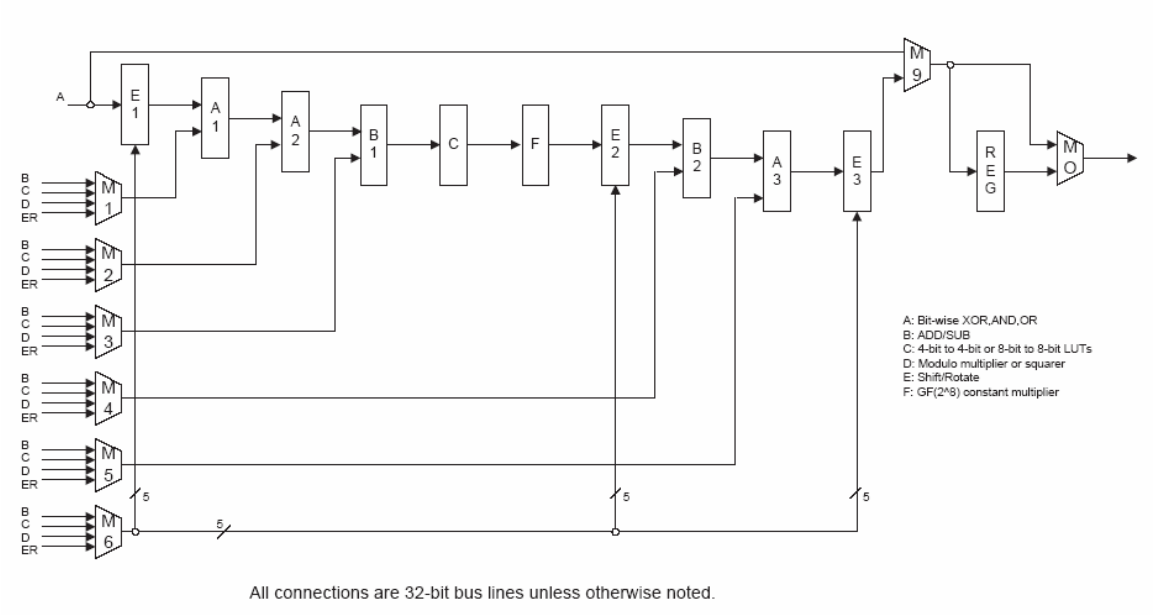


Figure 28: The RCE - basic configurable element of the COBRA architecture

Conversely, our design provides configurability at both the basic-block, as well as the interconnect. This allows us to use considerably simpler logic that can be combined together using interconnect to perform complex functions. Thus we are able to achieve higher logic utilization by offloading the complexity and area to the interconnect.

The COBRA approach has the following implications:

- Low utilization of logic in each block, since no ciphers require the use of all the provided logic, in exactly the same order. Conversely, our approach promotes high utilization of logic blocks, thanks to the flexible interconnect.

- Fixed delays through the RCE: since data must pass through all the logic elements irrespective of whether they are utilized or not, the latency of each operation mapped onto an RCE is constant. Furthermore, registering of outputs occurs only at RCE boundary, thereby limiting the achievable frequency of operation. In our approach, operation latency is controlled by the user to a greater degree, and registering is provided at the output of each basic-block, instead of each group, allowing our design to potentially attain higher frequencies.

Comparing our design with COBRA, it seems likely that for many symmetric-key ciphers, our approach may provide higher performance due to the above factors. However, it is unclear as to which design offers better area efficiency, since our design utilizes considerable amounts of interconnect, while COBRA utilizes replication of logic blocks.

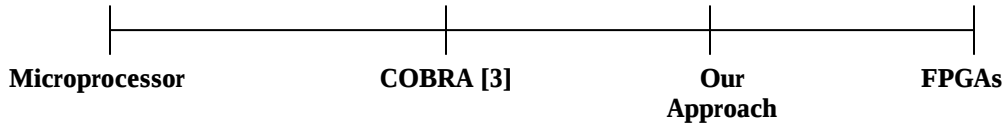
8. PROGRAMMING PARADIGM

8.1 ISSUES IN DEFINING A PROGRAMMING METHODOLOGY

Programming paradigms for computational resources vary greatly, from the traditional approach of describing computation as a control-flow oriented sequence of instructions, to the more esoteric configuration-oriented approach utilized by reconfigurable devices such as FPGAs. The suitability of a particular programming paradigm for a particular device is not so much a matter of choice, but is dependent on the degree and amount of configurability offered by the device.

The ALUs in conventional general purpose processors may be said to have minimal configurability, and thus require a large sequence of small instructions to define a complex operation. On the other hand, FPGAs are highly flexible in that their sub-components may be organized in any fashion. Thus ‘operation-selection’ from the vast variety of possible ‘operations’ requires a single large instruction that can completely define a complex operation.

The degree of configurability of the individual components of a device set the guidelines for how the device may be programmed. In the diagram below, the configurability of a device may be defined in the spectrum between FPGAs and Microprocessors. The COBRA reconfigurable crypto-processor described in [3] lies more towards the programmable end of the spectrum, as it is composed primarily of dedicated functional units with some additional configurability added on. It is in fact configured via VLIW style instructions.



Our approach on the other hand is more similar to FPGAs, but less flexible, simply because it is not intended for general-purpose computation. This reduction is due to removal of redundant flexibility that is beyond what is needed by our application domain, thereby allowing increased hardware efficiency and even performance.

When comparing with the COBRA architecture proposed in [3], our design does have one visible disadvantage. Being a predominantly reconfigurable architecture as opposed to a programmable one, our approach suffers from all the drawbacks associated with reconfigurable architectures, most notably:

- Lack of a comprehensive programming model.
- Lack of hardware virtualization.

The implication of the first issue is difficulty in programming, and hence cost-effectively utilizing reconfigurable architectures. Our approach may be just as susceptible to this as are FPGAs and other general and special-purpose reconfigurable devices. This limitation also affects COBRA, although to a much lesser extent, as their architecture is essentially abstracted as a programmable application-specific VLIW processor, with similar constraints as VLIW processors. The primary challenge there would be to automate instruction generation for the architecture.

The second issue is the primary limiting constraint associated with all reconfigurable architectures: the finiteness of hardware resources is exposed to the user. Thus it becomes extremely inefficient or difficult to implement designs that require more resources than are available. This problem is largely circumvented by the COBRA architecture, primarily because of its use of sequences of instructions, like conventional processors, which are immune to this issue. This perhaps is the primary limitation of our architecture that we have been able to identify at this preliminary stage.

8.2 PROGRAMMING OUR ARCHITECTURE

In order to partially address the first concern above, we propose the following programming model for our architecture. The following primitives will be made available to the end user:

- 32-bit Carry Save Adder
- 32-bit XOR
- 32-bit AND
- 32-bit OR
- 32, 64, and 128-bit Ripple Carry Adder
- 32, 64, and 128-bit Fixed Shifts
- 32 bit Rotates and random permutes.

- 64-bit, 128-bit limited permutes (TBD).
- 128-bit shift-register
- ANDing 32-bit value with a single bit
- Random bit-logic implementation, since each block is also capable of implementing:
- single 4-input function
- two 3-input functions
- four 2-input functions
- 4 global bit-broadcast lines
- 32-bit interconnect, point to point.

As our architecture is a special purpose design intended primarily for cryptography, the variety of tasks that can be performed by our basic block are limited, and consequently, so are the number of primitives available for use by prospective programmers. This has the advantage of dramatically simplifying the process of programming our design when compared to more general-purpose reconfigurable devices such as FPGAs.

Once a design is described as an interconnected mesh using the primitives defined above, mapping the design to the fabric should be relatively simple.

9. CONCLUSION: WORK IN PROGRESS

Although the potential exhibited naturally warrants a more in depth study in this direction, the following areas of our design are as yet still under consideration and have not been completely defined.

- A Complete definition of memory architecture required.
- VLSI Design to evaluate performance metrics and fine-tuning of logical design… (i.e. if found to be too slow, reduce number of switches, use longer wire, minimize the amount of interconnect to that which is necessary etc.)
- Evaluate the inclusion of a Barrel shifter in each Block or alternative blocks.
- Implementation of memory-less AES,

Furthermore, the iterative process of evaluating more symmetric-key algorithms and refining the architecture are still in progress.

BIBLIOGRAPHY

1) NUA INTERNET SURVEYS**, "How many online?",** http://www.nua.ie/surveys/how_many_online/

2) CISCO SYSTEMS, INC., "**IPSEC**", http://www.cisco.com/public/products tech.shtml

3) **"Introduction to SSL",** http://docs.sun.com/source/816-6156-10/contents.htm

4) AOKI, K. AND LIPMAA, H., "**Fast implementations of AES candidates**", *In Proceedings of the 3rd AES Candidate Conference*, 2000.

5) BASSHAM L. E. III., **"Efficiency testing of ANSI C implementations of round 2 candidate algorithms for the advanced encryption standard**", *In Proceedings of the 3rd AES Candidate Conference*, 2000.

6) S. W. SMITH, "Fairy Dust, Secrets, and the Real World", *IEEE Security & Privacy*, 2003.

7) SCHNEIER, B., **"Applied Cryptography**", *2nd ed. Willey, New York*, 1996. (PDF copy)

8) ADAM J. ELBIRT, **"Reconfigurable Computing for Symmetric-Key Algorithms"**, *PhD. Dissertation, Worcester Polytechnic Institute*, April 2002.

9) BROWN, S. AND ROSE, J., **"FPGA and CPLD architectures: A tutorial",** In *Proceedings of the IEEE Design & Test of Computers*, 1996.

10) ROSE, J., GAMAL, A., AND SANGIOVANNI-VINCENTELLI, A., **"Architecture of field programmable gate arrays"**, *Proc. IEEE*, 1993.

11) ANDREAS DANDALI, VIKTOR K. PRASANNA, **"An Adaptive Cryptographic Engine for Internet Protocol Security Architectures"**, *ACM Transactions on Design Automation of Electronic Systems, Vol. 9, No. 3, Pages 333–353.*, July 2004.

12) KEN EGURO, SCOTT HAUCK, **"Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development"**, *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

13) D. CARLSON, D. BRASILI, A. HUGHES, A JAIN, T. KISZELY, P. KODANDAPANI, A. VARDHARAJAN, T. XANTHOPOULOS, V. YALALA, **"A High Performance SSL IPSEC Protocol Aware Security Processor"**, *IEEE International Solid-State Circuits Conference*, 2003

14) K. C. POSCH, R. POSCH, **"Residue number systems: a key to parallelism in public key cryptography"**, IEEE, 1992

15) P. ADRIAN WANG, WEI CHANG TSAI, C. BERNARD SHUNG,**"New VLSI Architectures of Public-Key Cryptosystem"**, IEEE, 1997

16) C. N. Zhang, Y. Xu, C. C. Wu,**"A Bit-Serial Algorithm and Implementation for RSA"**, IEEE, 1997

17) CIARAN MCIVOR, MAIRE McLOONE, JOHN McCANNY,**"A High Speed, Low Latency RSA Decryption Silicon Core"**, IEEE, 2003

18) GONG PEIJUN, GUO LI, BAI XUEFEI,**"A 1024-bit Cryptosystem Hardware Design Based on Modified Montgomery's Algorithm"**, IEEE, 2003

19) CHEN-HSING WANG, CHIH-PIN SU, CHIH-TSUN HUANG, CHENG-WEN WU,**"A Word-Based RSA Crypto-Processor with Enhanced Pipeline Performance", IEEE AP-ASIC, 2004**

20) CHANG N. ZHANG, HEROLD L. MARTIN, DAVID Y. Y. YUN, **"Parallel Algorithms and Systolic Array Designs for RSA Cryptosystem"**, IEEE, 1988

21) CIARAN MCIVOR, MAIRE MCLOONE, JOHN V. MCCANNY, ALAN DALY, WILLIAM MARNANE, **"Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architecture"**, IEEE, 2003

22) THOMAS BLUM, **"Modular Exponentiation on Reconfigurable Hardware"**, *PhD. Dissertation, Worcester Polytechnic Institute*, April 1999

23) XINMIAO ZHANG, KESHAB K. PARHI, **"Implementation Approaches for the Advanced Encryption Standard Algorithm"**, IEEE, 2002

24) VIKTOR FISCHER, "**Realization of the Round 2 AES Candidates using Altera FPGA"**, MICRONIC s. r. o., www.micronic.sk, 2000

25) KRIS GAJ AND PAWEL CHODOWIEC, **"Hardware performance of the AES finalists - survey and analysis of results"**, George Mason University, 2001

26) **Serpent Home Page,** http://www.cl.cam.ac.uk/~rja14/serpent.html

27) IBM MARS TEAM, **"MARS and the AES Selection Criteria,** May 2000