Reconfigurable Computing for Symmetric-Key Algorithms

by
Adam J. Elbirt

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Electrical Engineering

April 22, 2002

Approved:

Prof. Christof Paar
ECE Department
Dissertation Advisor

Prof. Fred Looft
ECE Department
Dissertation Committee

Prof. Berk Sunar
ECE Department
Dissertation Committee

Prof. William Michalson
ECE Department
Dissertation Committee

Prof. John Orr
ECE Department Head

Prof. Wayne Burleson
ECE Department
University of Massachusetts Amherst
Dissertation Committee

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Importance of Security

Cryptography currently plays a major role in many information technology applications. With more than 100 million Americans connected to the Internet [5], information security has become a top priority. Many applications — electronic mail, electronic banking, medical databases, 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: [101, 125, 135]

- *Confidentiality*: Information cannot be observed by an unauthorized party. This is accomplished via the use of private-key and public-key encryption.

- *Data Integrity*: Transmitted data within a given communication cannot be altered in transit due to error or an unauthorized party. This is accomplished via the use of 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 the use of digital signatures.

- *Non-repudiation*: Neither the sender nor the receiver of a message may deny transmission. This may be accomplished via digital signatures and third party notary services.

An important tool for providing the aforementioned security services are cryptographic algorithms. Cryptographic algorithms fall primarily within one of two categories: private-key (also known as symmetric-key) and public-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. 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 is then be used in conjunction with a symmetric-key algorithm. Symmetric-key algorithms tend to be significantly faster than public-key algorithms and as a result are typically used in bulk data encryption [125].

1.2 Implementation of Symmetric-Key Algorithms

The two types of symmetric-key algorithms are block ciphers and stream ciphers. Block ciphers operate on a block of data while stream ciphers encrypt individual bits. Block ciphers are typically used when performing bulk data encryption and the data transfer rate of the connection typically follows the encryption/decryption throughput of the implemented algorithm. High throughput encryption and decryption are becoming increasingly important in the area of high speed networking. 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 at speeds exceeding Asynchronous Transfer Mode (ATM) rates exceeding one billion bits per second (Gbps).

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. The advantages of a software implementation include ease of use, ease of upgrade, ease of design, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [43, 125]. 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 [43]. 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 [125].

Processors in personal computers are general purpose and are not optimized for block cipher implementation, resulting in a performance degradation when compared to hardware implementations. Even the fastest software implementations of block ciphers cannot obtain the required data rates for bulk data encryption [14, 22, 44, 58, 17, 136, 155]. As a result, hardware implementations are necessary in order for block ciphers to achieve this required performance level. The down side 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 block ciphers. The potential advantages of block ciphers implemented in reconfigurable hardware include:

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 [112, 113].

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 [1]. 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, a 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).

Efficient implementation of block ciphers is critical towards achieving both high security and high speed applications. Numerous block ciphers have been proposed and implemented, covering a wide and varied range of functional operations. As a result, it has become increasingly more difficult to develop a hardware architecture that results in both efficient and high speed block cipher implementations. To reach this goal, a study of a wide range of block ciphers was undertaken to develop an understanding of the functional requirements of these algorithms. To facilitate this study, an in-depth implementation study of the AES candidate algorithm finalists was undertaken. Once this study was completed, other well known block ciphers were also studied. These studies led to the development of a reconfigurable hardware architecture suited for block cipher implementation that yields acceptable results in the areas of efficiency and performance.

1.3 Dissertation Goals

The following goals were set for this dissertation:

1. *Discuss the advantages and disadvantages of a reconfigurable architecture for block cipher implementation as compared to other potential architectures.*

2. *Perform an in-depth study of the AES candidate algorithm finalists to determine common functionality and methods required to achieve efficient implementations.*

3. *Examine a wide range of block ciphers to determine the atomic operations that a reconfigurable architecture must support.*

4. *Develop a reconfigurable architecture that is able to efficiently implement a wide range of block ciphers.*

5. *Analyze performance data for numerous block ciphers when implemented on the proposed architecture.*

6. *Present a discussion of the future direction for the developed architecture.*

1.4 Dissertation Outline

Chapter 2 begins by presenting a history of block ciphers and the AES standard. It continues with a discussion of previous work done in implementing block ciphers in hardware. Additional material is presented on commercially available cryptographic processors.

Chapter 3 presents a discussion of architecture goals for use in achieving a hardware platform that results in efficient block cipher implementations. Various implementation options will be compared and contrasted to determine an optimal solution.

Chapter 4 presents a Field Programmable Gate Array (FPGA) implementation study of the AES candidate algorithm finalists. Functional descriptions and an analysis of hardware suitability is provided for each algorithm.

Chapter 5 presents an analysis of common elements among block ciphers.

Chapter 6 presents a discussion on the proposed reconfigurable architecture tailored to block cipher implementation. Functional blocks that support common block cipher elements are described. Other data manipulation elements are discussed and the high level layout and interconnection of all of the architecture elements is presented.

Chapter 7 presents an in depth discussion of the tools developed to support the proposed reconfigurable architecture.

Chapter 8 presents an analysis of the theoretical implementations of multiple block ciphers in the proposed reconfigurable architecture.

Appendix A presents the FPGA implementation data for the AES candidate algorithm finalists implementation study.

Appendix B presents the COBRA assembly language user manual.

Chapter 2

Previous Work

2.1 Block Cipher Background

Many block ciphers may be characterized as Feistel networks. Feistel networks were invented by Horst Feistel [53] and are a general method of transforming a function into a permutation. The basic Feistel network divides the data block into two halves where one half operates upon the other half [126]. The function, termed the f-function, uses one of the halves of the data block and a key to create a pseudo-random bit stream that is used to encrypt or decrypt the other half of the data block. Therefore, to encrypt or decrypt both halves requires two iterations of the Feistel network.

A generalization of the basic Feistel network allows for the support of larger data blocks. Generalization occurs by considering the swap of the halves of the data block as a circular right shift. This allows for the use of the same f-function but requires multiple rounds to input all of the sub-blocks to the f-function [7]. Figure 2.1 details the block diagram for block ciphers employing both the basic Feistel network and generalized Feistel networks of both three and four blocks. The f-function is represented by the shaded box and the $\oplus$ symbol represents a bit-wise XOR operation.

The f-function employs *confusion* and *diffusion* to obscure redundancies in a plaintext message [131]. *Confusion* obscures the relationship between the plaintext, the ciphertext,

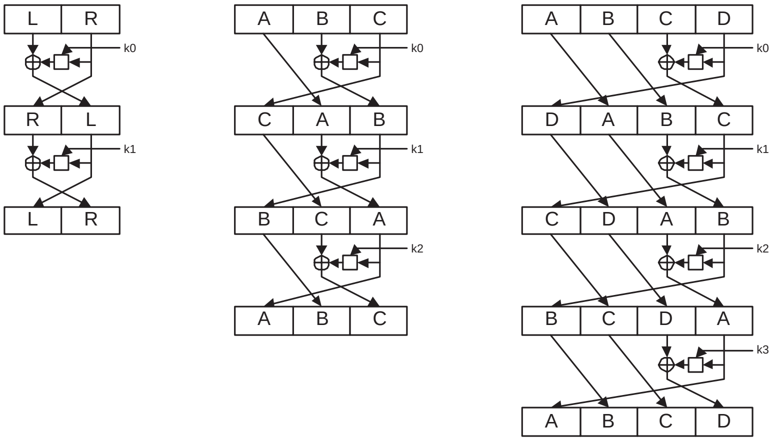


Figure 2.1: Block diagram for standard block ciphers [7]

and the key. S-Box look-up tables are an example of a *confusion* operation. *Diffusion* spreads the influence of individual plaintext or key bits over as much of the ciphertext as possible. Expansion and permutation functions are examples of *diffusion* operations [125]. The basic operations that may be found within an f-function include:

- Bitwise XOR, AND, or OR.
- Modular addition or subtraction.
- Shift or rotation by a constant number of bits.
- Data-dependent rotation by a variable number of bits.
- Modular multiplication.
- Multiplication in a Galois field.
- Modular inversion.
- Look-up-table substitution.

As an example, consider the AES candidate algorithm finalist RC6. The RC6 encryption round function is detailed in Figure 2.2, where shifts are denoted by $\ll$, rotations are denoted by $\lll$, and XORs are denoted by $\oplus$. The RC6 encryption round function implements the equations:

$$\begin{aligned}
A_{i+1} &= B_i \\
B_{i+1} &= [(([(2D_i^2 + D_i) \lll 5] \oplus C_i) \lll ([2B_i^2 + B_i] \lll 5)] + S[2i+1] \\
C_{i+1} &= D_i \\
D_{i+1} &= [(([(2B_i^2 + B_i) \lll 5] \oplus A_i) \lll ([2D_i^2 + D_i] \lll 5)] + S[2i]
\end{aligned}$$

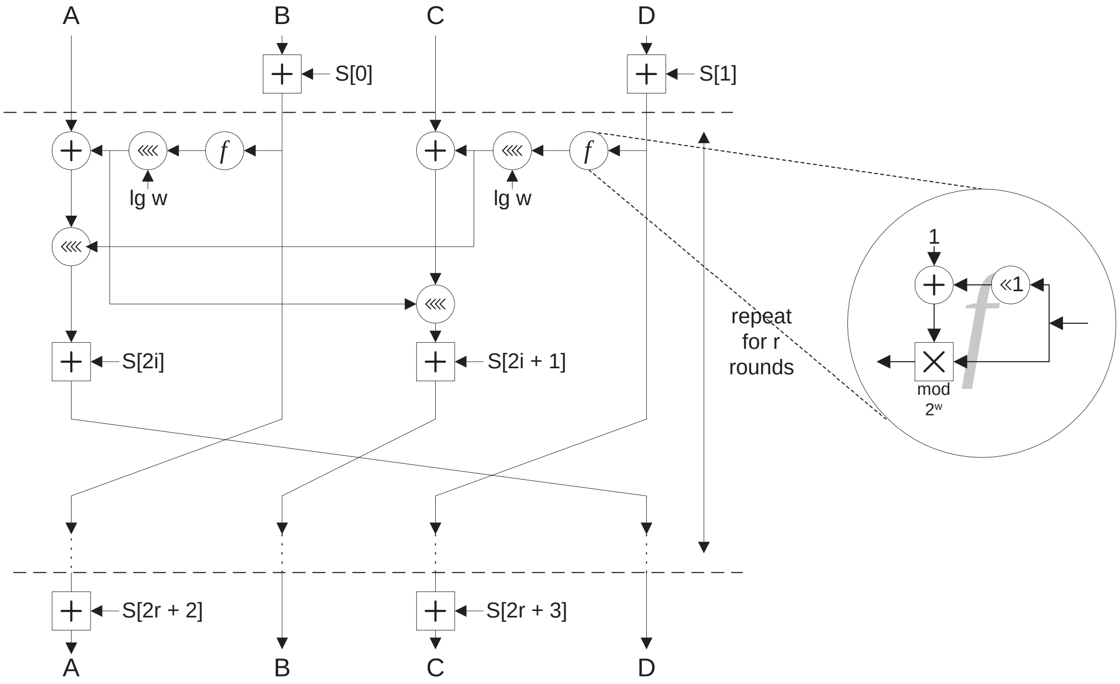


Figure 2.2: Block diagram of an RC6 encryption round

The operations required by an RC6 encryption round — bit-wise XOR, fixed shift, modular multiplication, variable rotation, and modular addition — are a representative sample of those found across all block ciphers, as will be shown in Chapter 5. Having established a basic background in the structure of block ciphers, the remainder of this chapter presents existing implementations of block ciphers in hardware, specifically custom ASIC, VLSI, reconfigurable logic, and reconfigurable processor implementations. In researching previous implementations of block ciphers in hardware, it was found that the most frequently implemented algorithms are DES, IDEA, and the AES candidate algorithms.

2.2 Commercial Hardware Implementations

2.2.1 DES

The earliest VLSI implementations of DES [40, 143] achieved throughputs ranging from 20 to 32 Mbps using 3 μm technology. The variances in throughput are compared and contrasted based upon speed versus area tradeoffs. The implementations support multiple modes of operation, including Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB) (see [2] for a detailed description of DES modes of operation). Other ASIC implementations of DES [45, 46] achieve a throughput of 1 Gbps using 0.8 μm Gallium Arsenide (GaAs) technology. More recently, a DES ASIC implementation has been demonstrated to operate at up to 10 Gbps using 0.6 μm technology [150]. Figure 2.3 provides an overview of DES and Figure 2.4 details the DES round function.

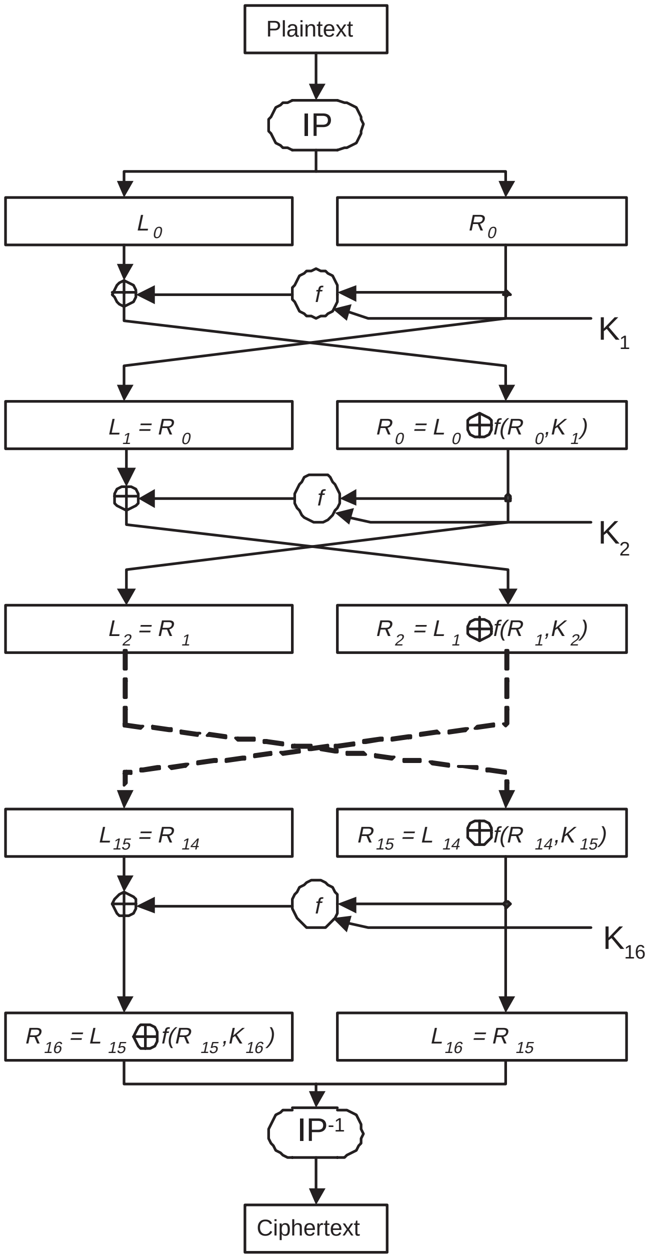


Figure 2.3: DES block diagram [125]

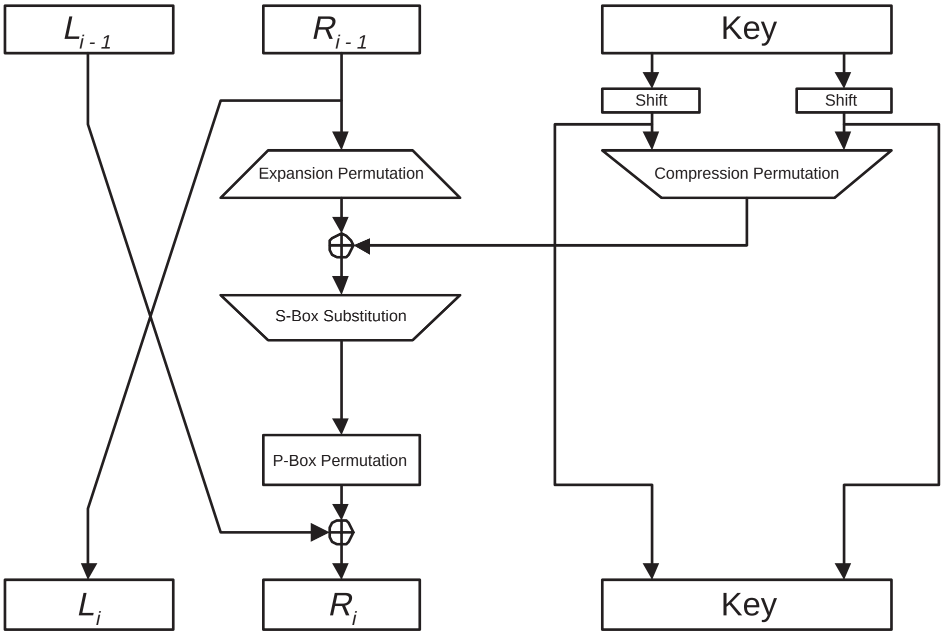


Figure 2.4: DES round function [125]

2.2.2 IDEA

IDEA began as the Proposed Encryption Standard (PES) [88] and evolved into its final form [89] due to modifications required to strengthen the cipher against differential cryptanalysis attacks [24]. IDEA is used in many commercial applications, such as Pretty Good Privacy (PGP). Like DES, IDEA operates across 64-bit blocks. However, while DES requires a 56-bit key, IDEA requires a 128-bit key, accounting for the increased security of the cipher as compared to DES. Figure 2.5 provides an overview of IDEA.

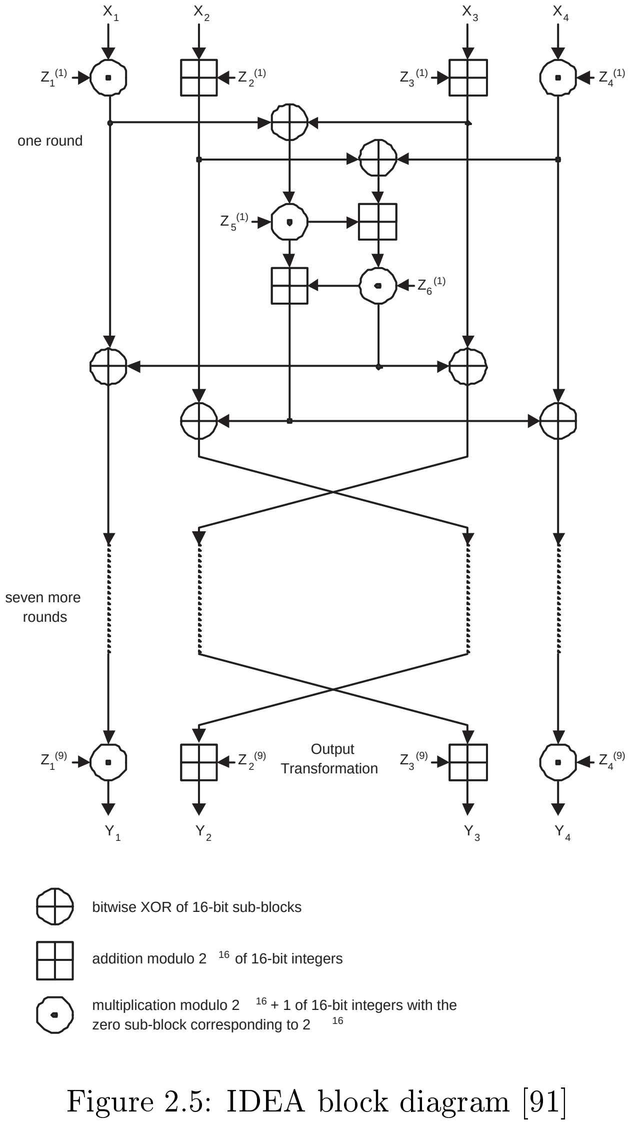


Figure 2.5: IDEA block diagram [91]

A VLSI implementation of PES [65] achieved a throughput of 44 Mbps using 1.5 μm technology. This implementation was limited in clock frequency to maintain compatibility with the Sun Microsystems SBus. The earliest VLSI implementations of IDEA [28, 164] achieved throughputs of 177 Mbps using 1.2 μm technology. More recent VLSI implementations [156] achieve a throughput of 355 Mbps using 0.8 μm technology. When using 0.7 μm technology, a throughput of 424 Mbps was achieved in a single chip solution [121]. However, the performance of these implementations were significantly reduced when operating in feedback modes.

2.2.3 AES

The National Institute of Standards and Technology (NIST) initiated a process to develop a Federal Information Processing Standard (FIPS) for AES, specifying an Advanced Encryption Algorithm to replace DES which expired in 1998 [125]. NIST solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five were selected as finalists. Overviews of four of the five AES candidate algorithm finalists may be found in Chapter 4. In October 2000, NIST chose Rijndael as the Advanced Encryption Algorithm.

A number of commercial hardware implementations have been presented for AES candidate algorithms that were not selected as finalists. A theoretical VLSI implementation of MAGENTA achieved an estimated throughput of 1 Gbps using 0.8 μm technology [74]. A VLSI implementation of SAFER K-128, the precursor to SAFER+, achieved a throughput of 251.8 Mbps using 0.7 μm technology [129, 130]. An ASIC implementation of CRYPTON achieved throughputs ranging from 898 Mbps to 2.69 Gbps using 0.35 μm technology [72].

Commercial hardware implementation studies have been performed for the candidate algorithm finalists [75, 146], the results of which are found in Table 2.1. Implementations of individual candidate algorithm finalists have also been performed. A VLSI implementation of MARS using 0.18 μm technology achieved throughputs of 459 Mbps when operating in CBC mode and 1.28 Gbps when operating in non-feedback modes [124]. A VLSI implementation of Rijndael using 0.18 μm technology achieved throughputs ranging from 435 Mbps to 1.82 Gbps depending on the key size [86].

Algorithm	**0.35 μm Technology** **Throughput (Mbps) [75]**	**ASIC** **Throughput (Mbps) [146]**
MARS	225.55	•
RC6	203.96	2171
Rijndael	1950.03	5163
Serpent	931.58	8030
Twofish	394.08	1445

Table 2.1: AES candidate algorithm finalists commercial hardware implementation studies — performance results

2.2.4 Other Algorithms

A VLSI implementation of RC5-32/12/6 achieved a throughput of 25.6 Mbps using 0.7 μm technology [129]. VLSI implementations of 3WAY achieved throughputs ranging from 440 Mbps to 1.6 Gbps using 0.7 μm technology [129].

2.3 Reconfigurable Logic Implementations

A great deal of work has been done on the implementation of block ciphers in reconfigurable logic, with most efforts being concentrated in FPGA implementations. The following summarizes block cipher implementations in FPGAs and other reconfigurable logic.

2.3.1 DES

The first published implementation of DES in an FPGA achieved a throughput of 26.4 Mbps [90]. However, this implementation required generation of key-specific circuitry for the target FPGA, a Xilinx XC4013-14, requiring recompilation of the implementation for each key. Until recently, the best performance an FPGA implementation of DES has achieved is a throughput of 402.7 Mbps when operating in ECB mode using a Xilinx XC4028EX-3 FPGA [78, 79, 80]. This implementation took advantage of pipeline design techniques to maximize the system clock frequency at the cost of pipeline latency cycles. However, with the advent of run-time reconfiguration and more technologically advanced FPGAs, implementations with throughputs in the range of ASIC performance values have been achieved. The most recent DES implementation employing run-time reconfiguration achieved a throughput of 10.752 Gbps when operating in ECB mode using a Xilinx Virtex XCV150-6 FPGA [113]. The use of the Xilinx run-time reconfiguration software application JBits^{TM} allowed for real-time key-specific compilation of the bit-stream used to program the FPGA, resulting in a smaller and faster design (which operated at 168 MHz) as compared to the design in [78, 79, 80] (which operated at 25.19 MHz). Most recently, a DES implementation achieved a throughput of 12 Gbps when operating in ECB mode using a Xilinx Virtex-E XCV300E-8 FPGA [140]. This implementation did not make use of run-time reconfiguration — through careful optimization of the DES datapath and the use of a 48-stage pipeline, the implementation was able to achieve a clock frequency of 188.68 MHz, resulting in the improved performance versus the implementation in [113].

2.3.2 IDEA

The first IDEA processor implementation using FPGAs achieved a theoretical throughput of 528 Mbps although it required four Xilinx XC4020XL devices [100]. An FPGA bit-serial implementation of IDEA using a single Xilinx Virtex XCV1000-6 device achieved a throughput of 2 Gbps [91] while an improved version of the implementation achieved a throughput of 2.4 Gbps [33] (see [91] for a more detailed discussion of bit-serial architectures). In [33], a bit-parallel implementation achieved a throughput of 5.25 Gbps on a Xilinx Virtex XCV1000-6 device, the fastest reported implementation of IDEA using a single FPGA.

The CryptoBooster, a generalized architecture for the implementation of block ciphers in FPGAs, uses an algorithm-specific CryptoCore module that interfaces to the other fixed modules within the architecture [104]. This architecture was designed to take advantage of partial reconfigurability, a feature now available in leading-edge FPGAs. Partial reconfigurability allows a portion of the FPGA to be reprogrammed while the remaining elements are left unchanged, resulting in a shorter reconfiguration time. A theoretical IDEA implementation using the CryptoBooster architecture achieved theoretical of 200 Mbps for single-round implementations and 1.5 Gbps for implementations that are both fully unrolled and pipelined [104].

2.3.3 AES

Multiple FPGA implementation studies have been presented for the AES candidate algorithm finalists [38, 39, 49, 50, 57, 141, 145], the results of which are found in Table 2.2. Note that [50] is a subset of the study performed in [49] and that no throughput results are presented in [145]. The studies performed in [49, 50, 57] used a Xilinx Virtex XCV1000-4 as the target FPGA. The study performed in [38, 39] used the Xilinx Virtex family of FPGAs but did not specify which FPGA was used as the target device. Finally, the study performed in [141] used the Altera Flex EPF10K130EQ-1 as the target FPGA.

Algorithm	**Non-Feedback Mode Throughput (Mbps) [49]**	**Feedback Mode Throughput (Mbps) [49]**	**Throughput (Mbps) [38, 39]**	**Throughput (Mbps) [57]**	**Throughput (Mbps) [141]**
MARS	•	•	101.88	39.8	•
RC6	2400	126.5	112.87	103.9	•
Rijndael	1940	300.1	353.00	331.5	232.7
Serpent	5040	444.2	148.95	339.4	125.5
Twofish	2400	127.7	173.06	177.3	81.5

Table 2.2: AES candidate algorithm finalists FPGA implementation studies — best performance results

FPGA implementations of individual candidate algorithms (both finalists and nonfinalists) have also been performed. Implementations of CAST-256 achieved throughputs of 11.03 Mbps using a Xilinx Virtex XCV1000-4 [47] and 13 Mbps using a Xilinx XC4020XV-9 [117]. An RC6 implementation achieved a throughput of 37 Mbps using a Xilinx XC4020XV-9 [117]. A Serpent implementation using a Xilinx Virtex XCV1000-4 achieved a throughput of 4.86 Gbps [48]. When targeted to a Xilinx Virtex-E XCV400E-8, a Serpent implementation achieved a throughput of 17.55 Gbps through the use of the Xilinx run-time reconfiguration software application JBits^{TM} which allowed for real-time key-specific compilation of the bit-stream used to program the FPGA [112]. This run-time reconfiguration resulted in a smaller and faster design (which operated at 137.15 MHz) as compared to the design in [48] (which operated at 37.97 MHz). When implemented using an FPGA from the Altera Flex 10KA family, the Serpent algorithm achieved a maximum throughput of 301 Mbps [108]. However, it is important to note that the implementation in [108] implements eight of the Serpent algorithm's thirty-two rounds while the implementations in [48] and [112] implement all of the rounds of the Serpent algorithm. Note that all of the presented throughput values are for non-feedback modes of operation.

Multiple implementations of Rijndael, the Advanced Encryption Algorithm, have been presented using both Xilinx and Altera FPGAs [54, 99, 109]. The implementation in [99] achieves a throughput of 6.956 Gbps using a Xilinx Virtex-E XCV3200E-8. Utilizing ROM to implement the Rijndael ByteSub operation resulted in a significant increase in throughput and decrease in area as compared to implementations in [38, 39, 49, 50, 57]. A Rijndael implementation achieved a throughput of 268 Mbps when using the Altera Flex EPF10K250 [109]. When targeting the more advanced Altera APEX 20KE200-1, Rijndael implementations achieved throughputs ranging from 570 Mbps to 964 Mbps depending on the implementation methodology [54].

2.4 Reconfigurable Processor Implementations

Reconfigurable processor architectures tend to be targeted towards accelerating the performance of specific applications. These architectures may be classified as one of the following types:

- A *reconfigurable system* consisting of multiple programmable devices, local memory, and an interconnect between the programmable devices.
- A *hybrid architecture* comprised of a microprocessor core, containing much of the functionality of a general purpose processor, combined with one or more reconfigurable function blocks that often support on-the-fly reconfiguration to provide optimized functionality for a given application [133].
- A *generalized reconfigurable architecture* designed to provide near-custom hardware performance with the flexibility of a software implementation through the use of on-the-fly reconfiguration.

The following summarizes block cipher implementations in reconfigurable processors. Reconfigurable processors targeting non-cryptographic applications are also presented, as an examination of these architectures yields a great deal of information for use in developing a specialized reconfigurable architecture optimized for block cipher implementation.

2.4.1 Reconfigurable Systems

A number of systems have been designed to map large applications to reconfigurable systems. These systems are comprised of multiple programmable devices, typically FPGAs, that are configured to contain a component of the partitioned application. Reconfigurable systems typically provide local memory, an interconnect between the programmable devices that is either fixed or programmable, and either an external controller or a dedicated programmable device that acts as a controller [27]. The use of these architectures has been primarily in the systolic array computation domain, targeting applications such as long number arithmetic, RSA cryptography, image classification, image analysis, image processing, motion detection, stereo vision, sound synthesis, signal processing, and genetic algorithms [16, 26, 60, 102, 144, 157]. Examples of reconfigurable systems used in systolic array computation include DECPeRLe, PARTS, SPLASH and SPLASH 2, and WILDSTAR^{TM}.

Logic emulation, where a gate-level design is mapped into configurable hardware, also provides a suitable application for reconfigurable systems. Examples of reconfigurable systems used in logic emulation include the Enterprise Emulation System, Realizer, and Teramac [11, 96, 142].

DECPeRLe

DECPeRLe is an array of Xilinx XC3090 FPGAs used as processing elements. Each FPGA has 16-bit connections to each of its Manhattan neighbors. The FPGAs in each row and each column of the array share two common 16-bit buses, allowing for connections between processing elements that are not Manhattan neighbors. Four independent 32-bit cache banks are provided at each edge of the array of processing elements. The cache bank control signals are generated by two controller FPGAs and the cache data is routed through four switching FPGAs for routing to the appropriate processing element within the array. Three real-time links are provided from the processing element array to external devices and a fourth link is provided for interfacing to a TURBOchannel host [3, 144].

PARTS

PARTS is an array of Xilinx XC4000 FPGAs combined to form a homogeneous array of processing elements with tightly coupled SRAM. Each processing element has an associated dedicated SRAM that serves to maximize memory bandwidth. Three FPGAs external to the array of processing elements are used to control the both the system clock and the datapath while providing a PCI interface to a host computer. The array has both nearest-neighbor mesh connections and eight “superpin” connections on each side of each processing element [67]. Superpin connections enable communication between chips using a single interconnect between adjacent pins, allowing for the creation of distribution networks while minimizing the required routing resources of the processing elements. Multiple boards may be daisy-chained to form larger arrays of processing elements [157].

SPLASH and SPLASH 2

SPLASH and SPLASH 2 are linear arrays of processing elements that are well suited for systolic applications which require limited interconnection between neighboring processing elements. SPLASH and SPLASH 2 support non-systolic applications via a crossbar that interconnects the Xilinx FPGAs that comprise the system. Each FPGA has a dedicated local memory as well as a 36-bit connection to its two nearest neighbors. A final FPGA is used to control the system and this FPGA is in turn initialized and configured via a Sparc host processor. Multiple boards may be daisy-chained to form larger arrays of processing elements [16, 27, 60].

Realizer

Realizer consists of a set of FPGAs used to emulate logic that are connected via a partial crossbar interconnect. A partial crossbar is implemented via a separate FPGA and consists of a set of smaller full crossbar interconnects that connect to the FPGAs that emulate logic but not to each other. The I/O of FPGAs used to emulate logic are divided into proper subsets and are connected to the same subset pins of each FPGA used as a partial crossbar. The number of FPGAs used as partial crossbars matches the number of subsets and the I/O for these FPGAs is equal to the number of pins per subset multiplied by the number of subsets. The result of this configuration is that the size of the partial crossbar increases linearly with the number of FPGAs used to emulate logic, a significant improvement as compared to the size of a full crossbar whose size increases exponentially as the number of FPGAs used to emulate logic increases. Realizer is comprised of Xilinx XC3090-70 FPGAs, which are used for logic emulation, and Xilinx XC2018-100 FPGAs, which are used as the partial crossbar interconnects. A host computer partitions the gate-level design and controls the download of the subcomponents into the FPGAs. A larger scale reconfigurable system implementation that employs the same design techniques as Realizer is the Enterprise Emulation System manufactured by Quickturn Systems [96, 142].

Teramac

Teramac consists of a set of custom FPGAs known as PLASMA [12]. The PLASMA FPGAs are used for both logic emulation and interconnection so as to minimize the number of wires on the printed circuit board. A separate controller board provides static RAM for use in both logic emulation and in interfacing with the host computer. The controller provides configuration information to the PLASMA FPGAs, transfers state data between the system components, and controls the clocking of the network so as to enable the use of breakpoints in software. Multichip modules (MCMs) are used to implement the network of PLASMA FPGAs. Four MCMs are contained in each board and up to sixteen boards may be interconnected along with a controller to form a Teramac system [11].

2.4.2 Hybrid Architectures

Hybrid architectures comprised of a microprocessor core combined with reconfigurable function blocks are typically used to accelerate the performance of a general purpose processor for specific applications. Reconfigurable function blocks may support on-the-fly reconfiguration to provide more optimized implementations and further improve system performance. The mapping of complex functions to adaptable hardware reduces the instruction fetch and execute bottleneck common to a software implementation [26]. However, the delay associated with communication between the microprocessor and the reconfigurable logic block often becomes the bottleneck within the system [27]. This overhead can be reduced by caching multiple configurations within the reconfigurable function blocks at the cost of more expensive and less flexible hardware [68, 81, 151, 152]. Hybrid architectures have been targeted at accelerating applications such as symmetric-key cryptography, digital signal processing, data compression, image processing, video processing, multimedia, block matching, automated target recognition, and wireless communications [9, 20, 21, 30, 35, 68, 69, 73, 81, 116, 120, 123, 134, 153]. Examples of hybrid architectures include A7, Chimaera, ConCISe, DReAM, E5, Garp, MorphoSys, NAPA, OneChip, PRISC, and ReRISC.

A7 and E5

The Triscend A7 32-bit configurable system-on-chip consists of an ARM7TDMI processor core combined with a Configurable System Logic (CSL) programmable matrix. To support the ARM processor, A7 provides 8K bytes of instruction and data cache, a high performance dedicated system bus, four independent DMA channels, and 16K bytes of RAM scratch space. A variety of peripherals are provided, including timers and UARTs. The CSL may be configured as either logic or memory and has access to the same memory space as the ARM processor [27, 35].

The Triscend E5 architecture is similar to that of the A7. E5 is an 8-bit configurable system on chip that incorporates an 8051/8032 microcontroller with a CSL programmable matrix. To support the processor, E5 provides up to 40K bytes of RAM, a high performance dedicated system bus, and two DMA channels. As in the A7 architecture, the CSL may be configured as either logic or memory and has access to the same memory space as the ARM processor [27, 35].

Chimaera

Chimaera combines a reconfigurable function unit (RFU) array that interfaces with a MIPS R4000 processor, a Content Addressable Memory (CAM), and an instruction cache. The CAM evaluates the instruction stream to determine if an instruction is an RFU operation. If an RFU operation is detected and the particular instruction is currently within the RFU array, the CAM allows the RFU output to be written to the result bus. Multiple instructions are contained within the RFU array and these functions operate in parallel upon the input data stream. There are no state-holding elements within the RFU array — sequential computations are implemented within an RFU with results stored in the system's register file. The RFU array may be partially reconfigured during run-time, allowing for the addition or removal of specific instructions. Chimaera also maintains a strictly downward flow through the array and does not allow feedback of signals to upper levels within the system [68].

ConCISe

The ConCISe architecture is targeted towards enhancing RISC processor performance in embedded applications through the use of custom instructions whose execution is based in the reconfigurable block of the architecture. The Reconfigurable Function Unit (RFU) operates in parallel with the processor's ALU and is controlled via an encoded instruction that matches the standard RISC encoding format. While on-the-fly reconfiguration was a system requirement, ConCISe does not implement partial reconfiguration so as to avoid the associated increase in hardware cost and complexity. ConCISe was implemented by combining a Phillips PZ3960 complex programmable logic device (CPLD) and a RISC processor. A DES implementation on the ConCISe platform achieved a 44 % performance increase versus the results of a general processor implementation due to a reduction in the number of instructions required to implement the cipher. Instruction count was reduced by off-loading the bit-level manipulations of DES to the RFU as these operations are highly inefficient when implemented in a microprocessor. Note that the performance improvement is based on the assumption of comparable operating frequencies for the RFU and the RISC processor's ALU [81].

DReAM

DReAM is an array of coarse-grained dynamically reconfigurable processing units (RPUs) designed to accelerate wireless telecommunication systems. The array is assumed to connect via a communication bridge to a general purpose processor and bus structure such as the ARM processor and the AMBA bus. Each RPU consists of arithmetic units, datapath spreading units, dual port RAMs, and a controller. Four RPUs share a configuration memory unit (CMU) that stores configuration data and all CMUs are controlled by a global communication unit (GMU) that interfaces with the general purpose processor. If a requested configuration is not within the associated CMU, the GMU indicates this to the processor who then initiates a transfer of the configuration from system memory to the appropriate CMU via the GMU. RPUs may be configured both on-the-fly and individually, resulting in a highly flexible system [20, 21].

Garp

The Garp architecture combines a standard single-issue MIPS processor with a reconfigurable array used as a hardware accelerator that attaches to the MIPS processor as a coprocessor. The reconfigurable array is composed of a matrix of logic blocks whose configuration is controlled by the MIPS processor and accelerated by a configuration cache. While the operation of the reconfigurable array is implemented via extensions to the MIPS instruction set, the reconfigurable array may also access the data cache or main memory independent of the MIPS processor. Memory buses are used to load configurations or transfer data when the reconfigurable array is idle. When the reconfigurable array is active, the array becomes the bus master and may use the buses to access memory. A theoretical implementation of DES in the Garp architecture operating at 133 MHz achieved a factor 24 speed-up versus an equivalent implementation on a 167 MHz Sun UltraSPARC 1/170. The Garp implementation was able to directly implement the S-Box look-up tables in parallel within the reconfigurable array. By avoiding referencing external memory the algorithm cycle count was greatly reduced as compared to software implementations, which require a memory read to perform each S-Box table look-up [27, 30, 69].

MorphoSys

The MorphoSys architecture is composed of a general purpose RISC processor core tightly coupled with a high-bandwidth memory interface and a reconfigurable processing unit. The reconfigurable cell array (RCA) is comprised of coarse-grain cells to match the granularity of the target applications. The RISC processor core controls the operation of the RCA and the RCA is configured through context words that specify an instruction for a given reconfigurable cell. The RCA follows a SIMD model with all reconfigurable cells within the same row or column sharing the same context word but operating on different data. MorphoSys supports on-the-fly partial reconfiguration and can store multiple configurations for the RCA. An IDEA implementation on the MorphoSys platform achieve a factor 11.8 reduction in cycles per encryption versus an equivalent general processor implementation. The performance improvement is a result of the ability of the MorphoSys architecture to take advantage of the parallelism within the IDEA algorithm, allowing for the system to operate on multiple blocks of plaintext simultaneously. However, the performance improvement does not hold when operating in feedback mode due to the dependence of the chaining of encrypted blocks [134].

NAPA

The NAPA architecture is comprised of an Adaptive Logic Processor (ALP) combined with a 32-bit RISC processor core. A dedicated interface exists between the ALP and the RISC processor and both may access the same memory space. While the ALP has general access to the RISC processor's memory space, high speed memories are dedicated to the ALP to improve performance. The ALP may be partially reconfigured, resulting in a wide range of functions being available to the RISC processor [27, 120].

OneChip

The OneChip architecture encompasses a MIPS-like processor that is tightly coupled to reconfigurable logic termed the programmable functional units (PFUs). OneChip may be configured to not make use of the PFUs so as to maintain binary compatibility with standard MIPS applications. The PFUs are used to implement the resources required to improve system performance and are integrated into the execute stage of the regular datapath of the processor. PFUs are associated with dedicated memories containing the application-specific function configurations. A high bandwidth channel between the PFUs and the processor core results in a net performance speed-up versus a general processor implementation [27, 153].

PRISC

PRISC combines a RISC processor core with a programmable function units (PFUs) to provide application-specific instructions. PFUs augment the processor's datapath by being placed in parallel with existing functional units. Extensions to the processor's instruction set allow for the use of a PFU in a fashion similar to that of other standard datapath elements.

The function for which a PFU is programmed is compared to the function required by the instruction and an exception is raised if the two do not match, resulting in the exception handler loading the required function into the PFU. While a PFU is significantly slower than a highly customized functional unit, its flexibility results in a significant performance improvement as compared to a general processor implementation [116].

ReRISC

The ReRISC architecture consists of a computation array, a programmable NOR plane, and a register file. The computational array consists of computational elements, each of which have eight configuration contexts that are selectable on a per-cycle basis. The computational array is capable of implementing the RISC instruction set with the exception of multiply and divide instructions. The programmable NOR plane is used to implement bit-level functions such as permutations, shifts, and packing/unpacking. ReRISC is dynamically configured such that only the necessary instructions for a given application are implemented within the computational array, with the only hard-coded instructions being those required to fetch data from memory and to store data in the computational array. This serves to minimize overhead and reduce the number of reconfiguration cycles required for a given application [73]

2.4.3 Generalized Reconfigurable Architectures

Generalized reconfigurable architectures consist of an interconnected network of configurable logic and storage elements where the granularity of the architecture is dependent upon the target application. Architectures are typically distinguished based on the control of processing resources — SIMD, MIMD, VLIW, systolic, microcoded, etc. Generalized reconfigurable architectures have been targeted applications such as symmetric-key cryptography, automatic target recognition, and real-time digital signal processing, video processing, and image processing, [31, 41, 42, 61, 76, 85, 103, 138, 139, 151, 152, 161]. Examples of generalized reconfigurable architectures include BYU DISC, DPGAs, MATRIX, PADDI, PCA-1, and PipeRench.

BYU DISC

The BYU DISC architecture makes use of the partial reconfiguration capability available in many currently available FPGAs to implement custom-instruction caching. Instruction modules are implemented as individual configurations that are loaded into the FPGA upon demand by the application. BYU DISC queries the FPGA to determine if a particular instruction module is present. If the required instruction module is not present, an idle module is removed from the FPGA and the required module is configured in its place. This on-the-fly replacement capability allows for an instruction set that is larger than what may be stored in the FPGA at any one time. The use of partial reconfiguration minimizes the idle time between executed instructions while maintaining the current state of the system [151, 152].

DPGAs

DPGAs operate as a multi-context FPGA with the capability of context-switching on a per-cycle basis. DPGAs are comprised of an array of processing elements, each of which may be configured to perform simple logic functions. A configuration look-up-table is coupled with each processing element and serves as a configuration cache, allowing for multiple processing elements to switch context in parallel. Given their similarity to FPGAs in terms of their processing elements and interconnection structure, DPGAs are well-suited to fine-grained applications that require multitasking or multithreading [41, 42, 103, 138].

MATRIX

MATRIX is a coarse-grained reconfigurable architecture that unifies the instruction and datapath resources, effectively forming a resource pool that may be allocated in an application-specific manner. The dynamic allocation of instruction and datapath resources allows MATRIX to support multiple types of architectural control. The MATRIX architecture consists of an array of functional units overlaid with a three-level configurable interconnection network. Each functional unit may serve as data memory, instruction memory, control logic, or an ALU. The configuration of the functional units allows for the implementation of both spatially limited and regular applications. In the case of spatially limited applications, the bulk of the functional units will be configured to operate as control logic. For regular applications, the functional units will be primarily configured as datapath logic and may support architecture options such as loop unrolling and pipelining. Moreover, the tailoring of the instruction set to the given application results in a minimized instruction stream. These features result in high-performance over a wide range of applications [103].

PADDI

PADDI was developed to enable rapid prototyping of DSP computation-intensive, coarse-grained datapaths. The architecture is comprised of a set of concurrently operating execution units (EXUs) that are interconnected via a configurable crossbar network that operates at the word level. EXUs are configured by a global controller that utilizes a VLIW control structure, allowing for on-the-fly reconfiguration. The EXUs are also optimized to perform high speed arithmetic operations typical of DSP applications [31, 103, 161].

PCA-1

The PCA-1 architecture is a fine-grained cell array architecture. The PCA-1 cell is dual structured, composed of a plastic part (PP) and a built-in part (BP). The PP is a configurable look-up-table which is configured by its associated BP, a small controller element. BPs are also responsible for communicating data between PPs. PPs and BPs are independently connected as meshes to form the PP plane and BP plane. The PP plane is therefore a sea of look-up-tables used to configure logic circuits and contains no flip-flops, long routing lines, or switching elements. The PP plane is designed for maximum asynchronous circuit implementation density. The BP plane performs performs message routing for the PP plane and can reconfigure PP cells within the PP plane at run-time [76, 85].

PipeRench

PipeRench, a generalized reconfigurable architecture, was not specifically developed for cryptographic applications. The architecture supports hardware virtualization, pipelined datapaths for word-based computations, and zero apparent configuration time. PipeRench is comprised of a set of physical pipeline stages termed stripes, each of which contains an interconnection network and a set of processing elements (PEs). The interconnection network allows PEs to access operands from the registered outputs of previous stripes as well as outputs from other PEs within the same stripe. Once a stripe has completed it's set of operations it may be reconfigured on-the-fly to implement new functionality. When used to implement IDEA, PipeRench achieved a throughput of 126.6 Mbps. When used to implement RC6, CRYPTON, and Twofish, the PipeRench architecture achieved throughputs of 58.8, 86.8, and 164.7 Mbps respectively. One of the drawbacks of PipeRench is that the architecture targets streaming applications where pipelining drastically increases system throughput once the latency of the pipeline has been met [61, 139]. Systems that require feedback, as is the case with most block ciphers (operating in CBC, CFB, or OFB modes) [126], are difficult to realize within the PipeRench architecture.

Chapter 3

Architecture Development

3.1 Architecture Goals

3.1.1 Algorithm Agility

Algorithm agility is defined as the ability to switch between cryptographic algorithms within a given system. Chief among the reasons for designing a system that supports algorithm agility is the algorithm-independent nature of modern security protocols and standards. For example, the SSL v3.0 and IPSec standards were written to support many cryptographic algorithms and SSL v3.0 is included in most of today's world wide web browsers. During setup of the secure SSL connection, a block cipher is chosen for encryption/decryption of the connection from the following list: RC4, RC2, DES, Triple-DES, 40-bit DES, and Skipjack [56]. IPSec is also defined to be algorithm independent. The IPSec specification discusses the implementation of security mechanisms, which are defined as algorithm independent, for the protection of user traffic. This allows selection of different sets of algorithms without affecting other parts of the implementation [84]. Among the encryption algorithms specified are DES, Triple-DES, Blowfish, CAST, IDEA, RC4, and RC6.

Another argument for algorithm agility is demonstrated when a system is designed with a given algorithm, and that algorithm is broken or rendered insecure. The system must then be redesigned via either hardware or software to support a new algorithm, requiring both time and money. However, if the system was designed with multiple algorithms in mind, the system simply ceases to use the insecure algorithm and continues to operate by using any of the other remaining algorithms. As an example, DES expired as a federal standard and was also rendered insecure in 1998 due to the fact that exhaustive key searches have become technically feasible. Despite the fact that DES is used in SSL v3.0, the protocol is still used to encrypt web sessions since it supports many other algorithms. This argument is further strengthened when considering remote systems that are difficult to access. For instance, in the case of satellites, upgrading the hardware incurs tremendous cost.

3.1.2 Performance

The performance required for the proposed reconfigurable architecture is dependent upon what application is being targeted. In developing the proposed architecture, three applications were targeted: ATM OC-3 networks, running at 155.52 Mbps, 100Base-X (Ethernet) networks running at 100 Mbps, and the average speed of the AES candidate algorithm finalists implemented on commercially available FPGAs.

3.2 Implementation Options

As described in [32], the principle approaches for building a generic block cipher implementation module are:

1. *General-Purpose Processor*: Software implementations in a commercially available, general-purpose processor.

2. *Specialized Processor Architecture*: Software implementations in a custom processor. The system may use an external FPGA or versatile functional blocks to perform functions not efficiently supported in the processor's internal functional blocks.

3. *Commercial Hardware*: Hardware implementations targeted towards ASIC, commercially available FPGA, or custom VLSI platforms.

4. *Specialized Reconfigurable Architecture*: Hardware implementations in a custom reconfigurable architecture.

3.2.1 General-Purpose Processors

A general-purpose processor approach refers to a software implementation of an algorithm in a commercially available processor. These processors may range from a simple 8-bit microcontroller, such as the 8051, to a highly complex 64-bit processor, such as the Alpha AXP 21164A. Numerous software implementations of block ciphers have appeared throughout the literature [14, 15, 19, 22, 23, 25, 36, 44, 55, 58, 83, 87, 94, 97, 98, 105, 114, 118, 122, 128, 132, 136, 137, 147, 148, 149, 155, 158]. General-purpose processors provide algorithm agility but fall short of the performance requirements, especially when considering most modern block ciphers. When high-end processors are considered, the cost per unit also becomes a limiting factor. Processors suffer a performance degradation due to overhead costs as they are designed to handle a broad range of functionality. These overheads costs are typically due to the inherent nature of a von Neumann architecture where fetch, execute, and store operations must occur for each instruction. While faster processors are continuously being designed, it is expected that ASIC implementations will continue to exhibit significantly improved performance at comparable operating frequencies.

To further illustrate the overhead costs of a general-purpose processor architecture when used to implement block ciphers, consider the example of the RC6 encryption round detailed in Section 2.1. The RC6 encryption round flow chart implements the equations:

$$\begin{aligned}
A_{i+1} &= B_i \\
B_{i+1} &= [(([(2D_i^2 + D_i) \lll 5] \oplus C_i) \lll ([2B_i^2 + B_i] \lll 5)] + S[2i+1] \\
C_{i+1} &= D_i \\
D_{i+1} &= [(([(2B_i^2 + B_i) \lll 5] \oplus A_i) \lll ([2D_i^2 + D_i] \lll 5)] + S[2i]
\end{aligned}$$

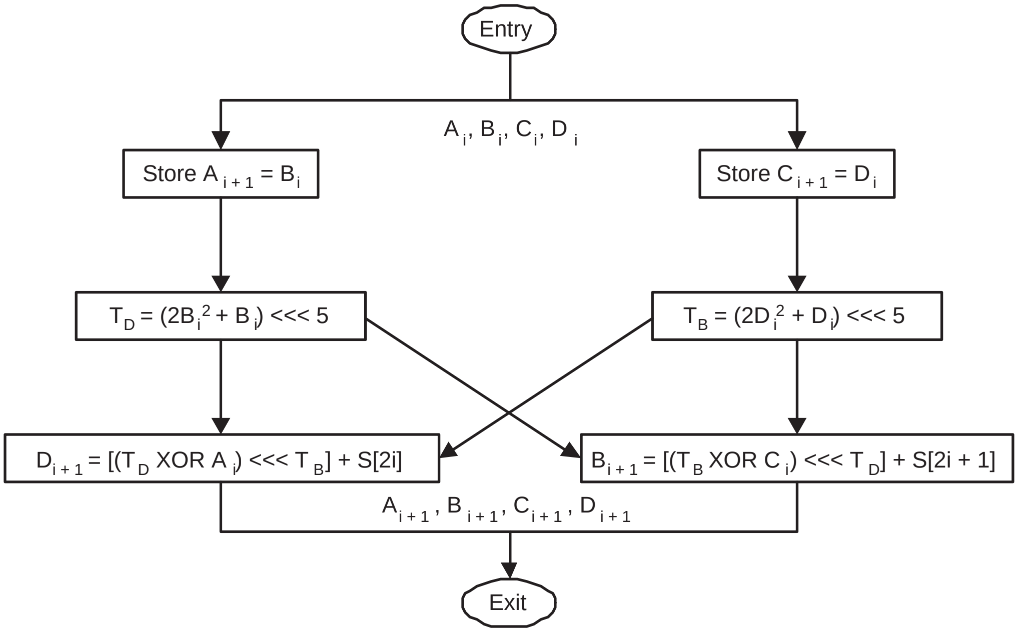

Entry
A i, B i, C i, D i
Store A i + 1 = B i
Store C i + 1 = D i
T D = (2B i 2 + B i) <<< 5
T B = (2D i 2 + D i) <<< 5
D i + 1 = [(T D XOR A i) <<< T B] + S[2i]
B i + 1 = [(T B XOR C i) <<< T D] + S[2i + 1]
A i + 1, B i + 1, C i + 1, D i + 1
Exit

Figure 3.1: RC6 flow chart

SISD Processors

The flow chart shown in Figure 3.1 must be mapped to the targeted general-purpose processor for implementation. In the case of a Single Instruction stream Single Data Stream (SISD) processor, the architecture is assumed to have a single ALU capable only of scalar arithmetic [70]. A data-flow diagram for a SISD processor implementation is shown in Figure 3.2.

The SISD processor is assumed to have enough local register space to hold the data values for A_i, B_i, C_i, and D_i, required for performing RC6 encryption. Should the SISD processor be lacking in local register space, register-based operations would require the loading and storing of data to memory, significantly decreasing implementation performance by creating a memory access bottleneck. Moreover, the SISD processor is assumed to have both local registers and an ALU wide enough to support the 32-bit RC6 internal block size and operation bit-width. The encryption implementation will require multiple sub-operations to perform what are currently represented as single operations in Figure 3.2 if either of these hardware resources fail to meet the 32-bit width requirement, resulting in

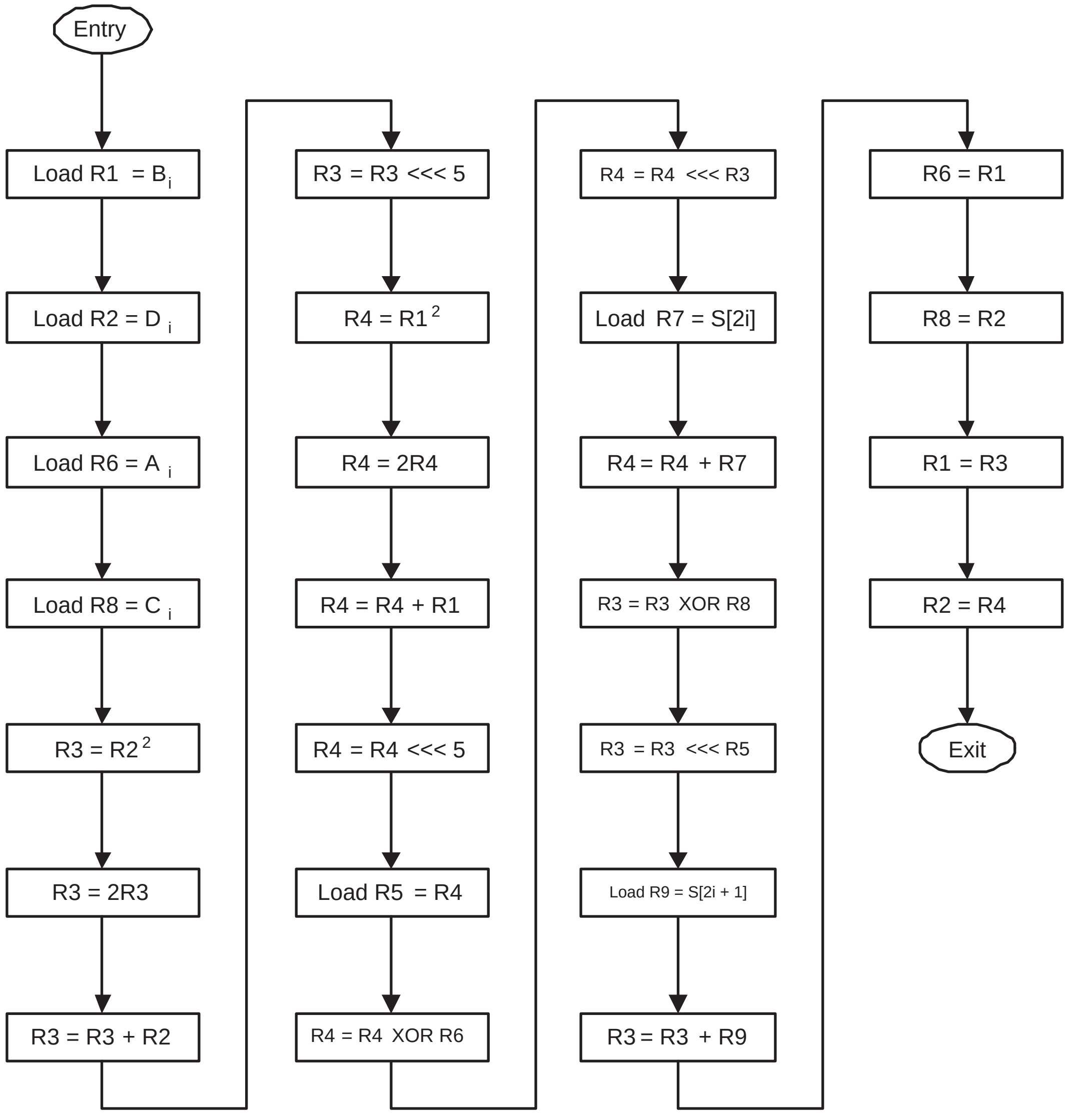


Figure 3.2: RC6 data-flow diagram for a SISD processor implementation

an architecture that is too fine-grained for the targeted application. Finally, it is assumed that the SISD processor is capable of performing each of the RC6 encryption core functions in a single operation. While this assumption typically holds true for modular addition, modular subtraction, shift, rotate, or boolean operations, this is not the case for modular multiplication, multiplication in a Galois field, modular inversion, or look-up-table substitution operations. In the case of RC6 encryption, the *mod* 2^{32} multiplier will most likely be implemented via either a 32-bit full multiplier or a shift-and-add process loop. A full multiplier implementation will be inefficient in terms of logic resources required as compared to a *mod* 2^{32} multiplier. Moreover, given that most full multipliers are pipelined, both the full multiplier and the shift-and-add process loop implementations will be inefficient in terms of the number of cycles required to complete a multiplication.

Even with the aforementioned assumptions, the SISD processor is unable to take advantage of the parallelism inherent in RC6 encryption. As indicated in Figure 3.1, the data streams for calculating B_{i+1} and D_{i+1} may operate independently until the data dependent rotations occur, at which point the data streams must transmit their rotation values to each other. Once the rotation values have been shared, the data streams for calculating B_{i+1} and D_{i+1} may complete their operations independently. However, because the SISD processor is by nature a single data stream processor, the calculation of B_{i+1} and D_{i+1} must be performed sequentially. Finally, given that the subkeys *S[2i]* and *S[2i + 1]* change for each round, the values for the appropriate *S[2i]* and *S[2i + 1]* must be loaded from memory each iteration through the round function. Given that accessing memory requires multiple clock cycles, this results in a memory access bottleneck similar to the bottleneck seen if local registers are not available for the storage of the RC6 encryption data values.

SIMD Processors

In the case of Single Instruction stream Multiple Data Stream (SIMD) parallel processing, also known as array processing, the architecture is assumed to have a single program broadcast to multiple execution units where the multiple data streams are the local data accessed by each individual processor [66, 70]. Examples of SIMD processors include the ILLIAC IV, the Goodyear Aerospace MPP, and the International Computers Ltd. Distributed Array Processor (DAP) [18, 71]. Figure 3.3 details the structure of an array processor similar to the ILLIAC IV [66].

Figure 3.4 details the data-flow diagram for a SIMD implementation of the RC6 encryption flow previously described in Figure 3.1. Note that the two processes differ only in the data being accessed and are the same in all other respects. The SIMD implementation

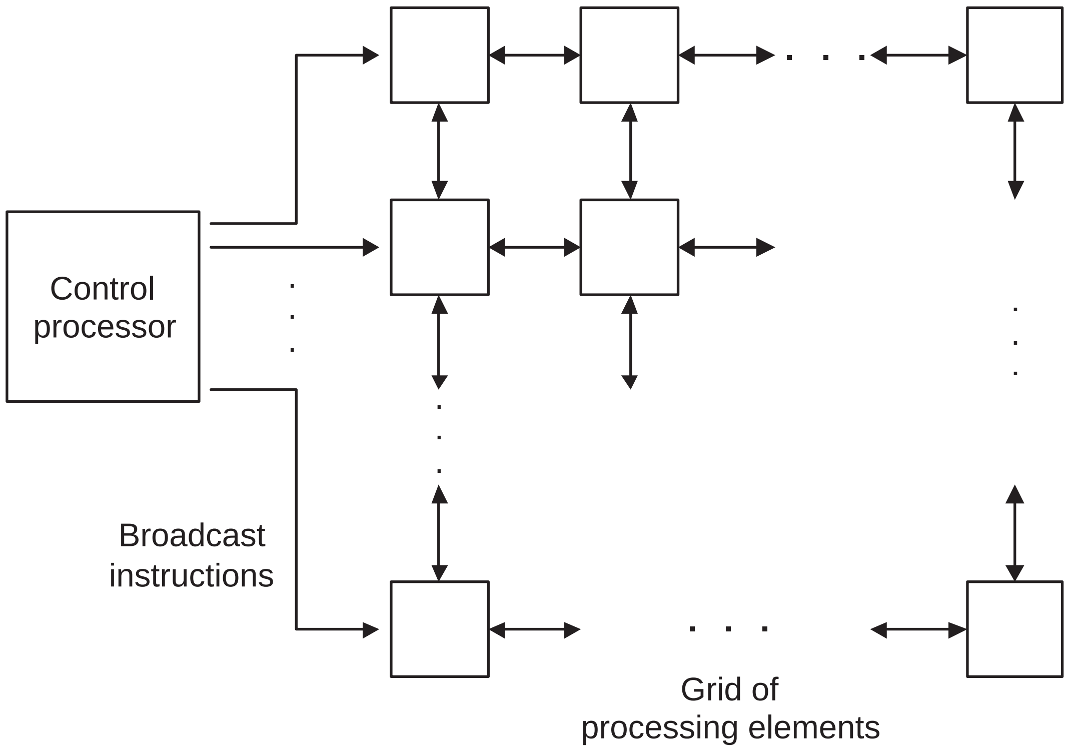


Figure 3.3: Structure of a SIMD array processor [66]

operates under the same assumptions as those made in the SISD processor implementation in terms of the availability of local storage registers, the bit widths of local registers and the ALUs, and the capability of performing the required operations for RC6 encryption. Should these assumptions prove incorrect, performance degradation similar to the degradation described in the SISD processor implementation would be expected.

While the SIMD implementation halves the number of instructions required to implement RC6 encryption in relation to the SISD processor implementation, it still encounters a memory access bottleneck. Given that the subkeys *S[2i]* and *S[2i + 1]* change for each round, the values for the appropriate *S[2i]* and *S[2i + 1]* must be loaded from memory each iteration through the round function. This results in a memory access bottleneck similar to the bottleneck seen if local registers are not available for the storage of the RC6 encryption data values. Moreover, because the two datapaths are interdependent, the interconnection matrix between processing elements becomes a key factor in evaluating system performance. An inflexible interconnection matrix will result in increased overhead being required to transfer the data dependent rotation values between processing elements, resulting in a performance degradation.

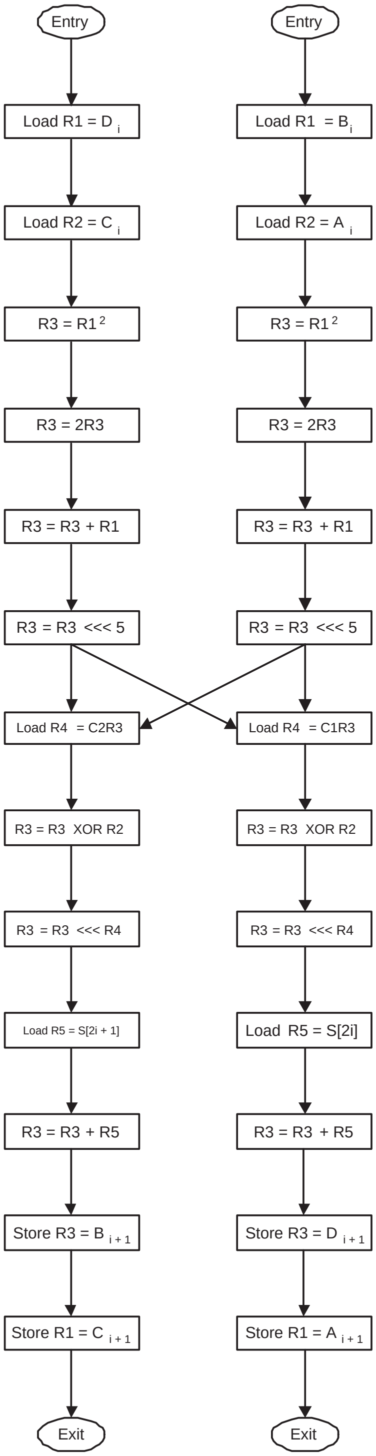


Figure 3.4: RC6 data-flow diagram for a SIMD processor implementation

MIMD Processors

In the case of Multiple Instruction stream Multiple Data Stream (MIMD) systems, also known as multiprocessors, the architecture is assumed to have multiple independent processors, each executing a different program and accessing different data [66, 70]. Pipelined processors may also be categorized as MIMD systems as each pipeline segment has both a distinct data stream as well as a distinct instruction stream. Examples of MIMD processors include the Carnegie-Mellon University Cm* and the NCUBE/ten [52, 77]. Figures 3.5 and 3.6 detail the structure of both a general multiprocessor and a global memory multiprocessor.

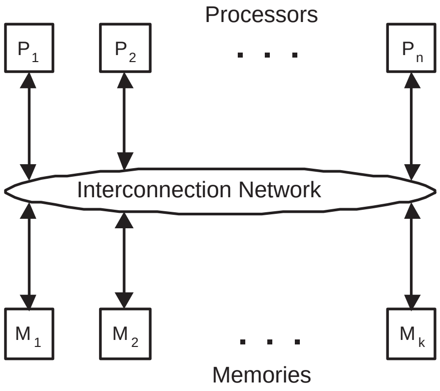


Figure 3.5: Structure of a MIMD general multiprocessor [66]

The MIMD multiprocessor implementation operates under the same assumptions as those made in the SISD and SIMD processor implementations in terms of the availability of local storage registers, the bit widths of local registers and the ALUs, and the capability of performing the required operations for RC6 encryption. Should these assumptions prove incorrect, performance degradation similar to the degradation described in the SISD and SIMD processor implementations would be expected. The configuration of the system shown in Figure 3.5 also results in a memory access bottleneck as the interconnection network introduces significant delays between a given processor and a memory being accessed for instruction and data fetching. The configuration of the system shown in Figure 3.6 alleviates this bottleneck by coupling each processor with its own local memory while the

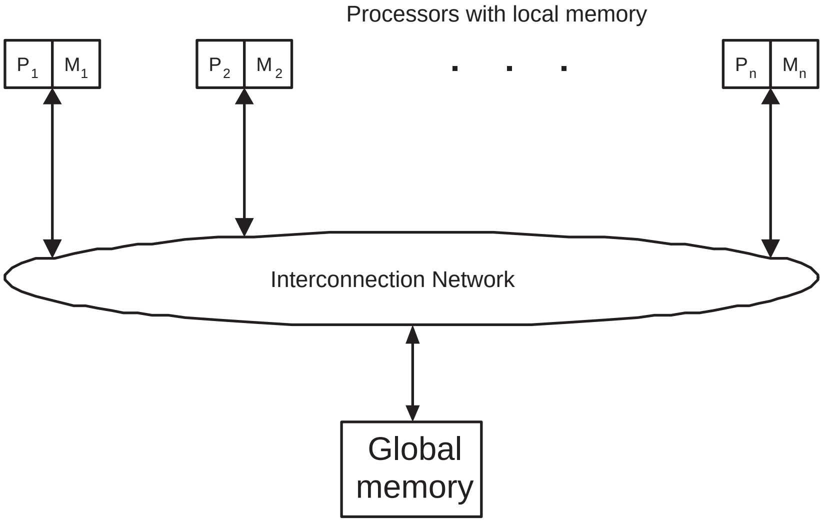


Figure 3.6: Structure of a MIMD global memory multiprocessor [66]

interconnection network allows all processors to access a global memory as well [66]. However, as shown in the SISD and SIMD processor implementations, any memory access becomes a bottleneck when implementing RC6 encryption.

The data-flow diagram for a SIMD implementation of the RC6 encryption flow detailed in Figure 3.4 also applies to a MIMD multiprocessor implementation. However, the MIMD multiprocessor implementation would be able to take advantage of hardware pipelining to increase the overall encryption throughput. Subkeys would be stored in the local memory associated with a given processor while data dependent rotation values would be shared in the global memory, resulting in a memory access performance bottleneck at both points.

3.2.2 Specialized Processors

A specialized processor refers to a processor designed for a given application or families of applications. Certain commercially available processors, such as those designed for digital signal processing (DSP) or network applications, fall into this category. Designing a processor that efficiently implements block ciphers qualifies as a specialized processor. As suggested in [32], an external FPGA may be coupled with such a specialized processor to support functions not efficiently implemented in the processor itself. When an algorithm is implemented, an analysis must be performed to determine which functions are to be executed in the processor and which functions are to be executed in the external FPGA. A block diagram of this architecture is shown in Figure 3.7. While this form of implementation meets the algorithm agility requirements outlined in Section 3.1, it fails to meet the performance requirements outlined in the same section. A rough estimation in [32] suggests that this implementation method yields performance results that are equivalent to the comparison general-purpose processor based software implementations for equivalent clock frequencies [58].

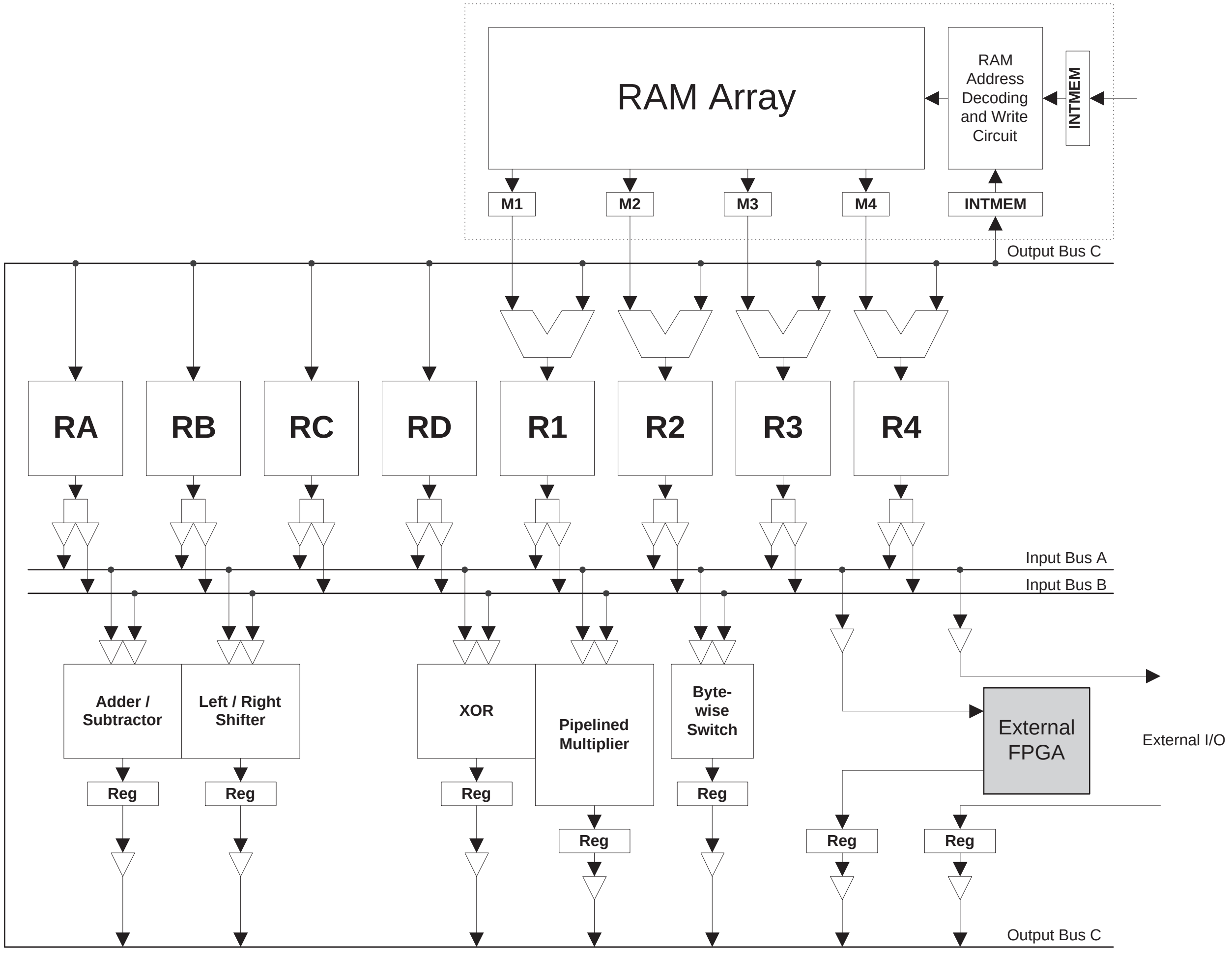


Figure 3.7: Specialized processor for block cipher implementation [32]

In an attempt to improve block cipher implementation performance, versatile function blocks referred to as ”hyperops” were added to the specialized architecture detailed in Figure 3.7. A block diagram of this architecture is shown in Figure 3.8. Hyperops are capable of performing multiple operations within one clock cycle and the operations performed within one hyperop are selected from a given set of functions by the application programmer. In addition, the specialized processor was expanded to support two parallel datapaths, allowing two hyperops to operate concurrently. This implementation method outperforms the comparison software implementations for equivalent clock frequencies by as much as a factor of three. The performance enhancement was primarily due to the use of the hyperops to decrease cycle counts in addition to the outsourcing of operations to the FPGA that do not perform well in a general purpose processor environment. However, even the best results still fall short of the targeted performance rates of 155.52 Mbps ATM OC-3 networks [32].

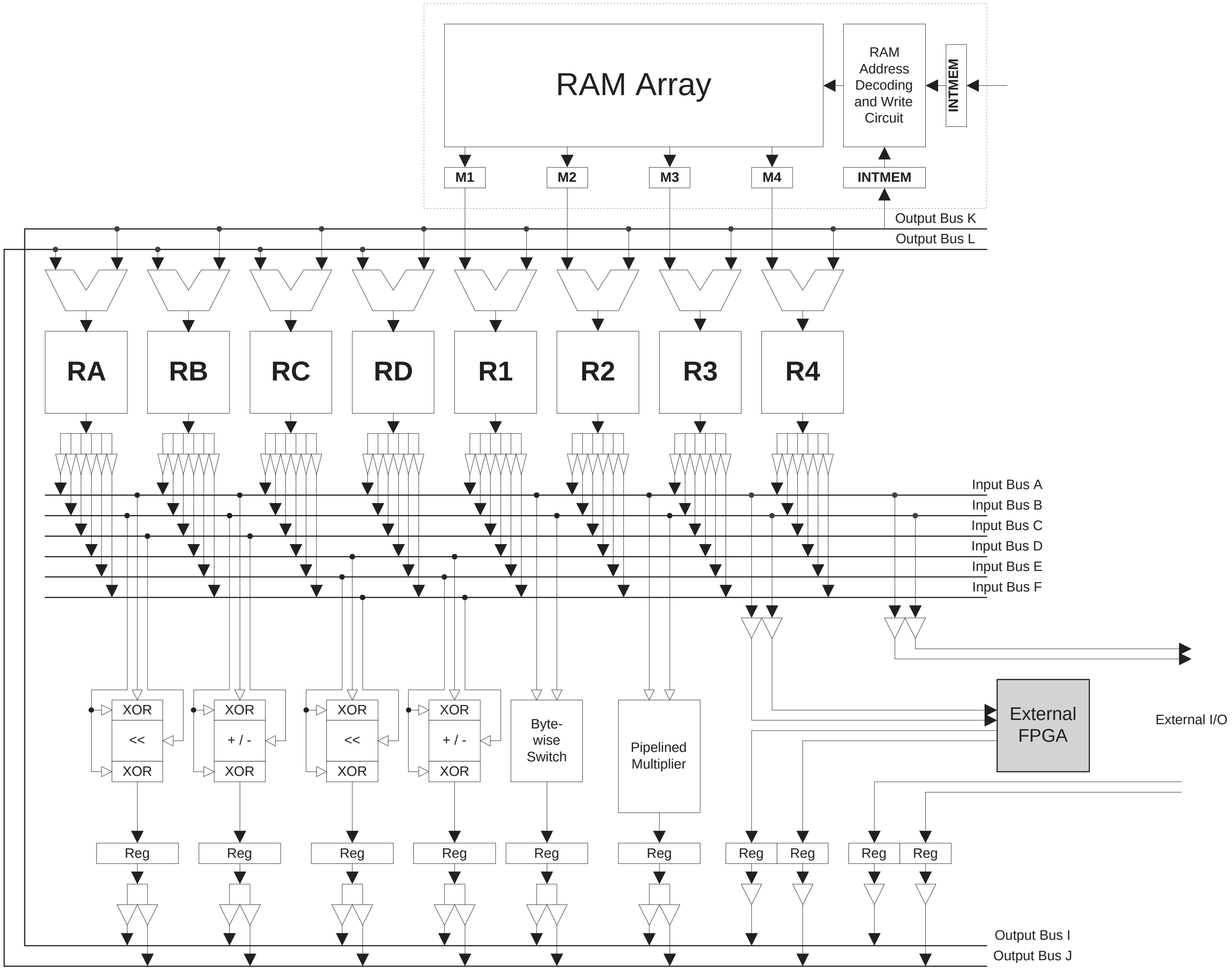


Figure 3.8: Specialized processor with hyperops for block cipher implementation [32]

3.2.3 Commercial Hardware

FPGAs are devices that contain many programmable function units in a reconfigurable interconnect matrix. Several vendors, including Altera, Lucent, and Xilinx, offer FPGAs in a variety of sizes and packages. These devices provide equivalent gate counts ranging from 1,000 to 3,000,000 logic gates [10, 95, 160]. As a result of the look-up-table requirements of many block ciphers, RAM resources are an important attribute when considering an FPGA-based implementation. FPGAs are commonly programmed using a hardware description language such as Verilog or VHDL. The code written in this language is then synthesized and optimized into a gate-level netlist which is then placed and routed, resulting ultimately in a bitstream used to program the function units and the interconnect matrix of the FPGA. Multiple FPGA-based implementations of block ciphers have appeared throughout the literature [38, 47, 48, 49, 50, 54, 57, 78, 79, 80, 99, 104, 111, 112, 113, 117, 140, 145]. FPGAs can provide algorithm agility due to their ability to be reprogrammed as needed. Typically, FPGA implementations do not provide as high a throughput as compared to ASIC implementations and tend to be expensive for high-volume applications.

ASICs are custom devices designed for a specific functionality that is fixed at the time of manufacturing. A variety of ASIC implementations of block ciphers have appeared throughout the literature [45, 46, 72, 146, 150]. ASICs typically have higher clock rates than FPGAs and thus can provide higher throughput. However, given that the functionality of the ASIC is fixed, it does not fulfill our requirement of algorithm agility. Using an ASIC in production level quantities is cheaper than an FPGA. However, initial design and fabrication costs often exceed $100,000. Hence, for low or medium volume applications, an ASIC can be prohibitively expensive.

Both ASICs and 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 [30, 31, 69, 81, 123, 134, 154].

Finally, the interconnection matrices of both FPGAs and ASICs provide 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.

3.2.4 Specialized Reconfigurable Architecture

As demonstrated in Chapter 2, reconfigurable processor architectures offer a great deal of insight towards creating an architecture optimized for block cipher implementation. A specialized reconfigurable architecture that is optimized for the implementation of block ciphers 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 communication are minimized by maintaining most or all of the system's resources on chip. 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 [68, 120, 134, 153]. 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 block cipher space [9, 31, 51, 134, 151, 152, 161, 163]. Because block ciphers 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 [30, 31, 69, 81, 123, 134, 154]. Finally, an interconnect matrix must also be designed to support both Feistel networks (common to most block ciphers [126]) as well as other forms of networks, such as Substitution-Permutation (SP) networks.

3.2.5 Architecture Comparison

Table 3.1 summarizes the characteristics of each of the principle approaches for building a generic block cipher implementation module. Based on these characteristics, a specialized reconfigurable architecture optimized for the implementation of block ciphers appears to be the most balanced solution. The specialized reconfigurable architecture satisfies the requirements for algorithm agility (and in turn the need for algorithm upload and algorithm modification) at a theoretically low cost per unit while maintaining acceptable performance and efficiency results.

Architecture	**Algorithm Agility**	**Efficiency**	**Performance**	**NRE Cost**	**Unit Cost**
General Purpose Processor	Yes	Low	Low	Low	Low
Specialized Processor	Yes	Moderate	Moderate	Low	Low
Commercial FPGA	Yes	High	Moderate	Low	High
Custom ASIC	No	High	High	High	Low
Specialized Reconfigurable Architecture	Yes	Moderate	Moderate	Low	Low

Table 3.1: Characteristics of the principle approaches for building a generic block cipher implementation module

Chapter 4

AES FPGA Implementation Study

The National Institute of Standards and Technology (NIST) initiated a process to develop a Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard (AES), specifying an Advanced Encryption Algorithm to replace DES which expired in 1998 [125]. NIST solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five were selected as finalists. In October 2000, NIST chose Rijndael as the Advanced Encryption Algorithm. For more information about the AES selection process, the reader is referred to the NIST web site [1].

Unlike DES, which was designed specifically for hardware implementations, one of the design criteria for AES candidate algorithms is that they can be efficiently implemented in both hardware and software. Thus, NIST announced that both hardware and software performance measurements would be included in their efficiency testing. However, prior to the third AES conference in April 2000, virtually all performance comparisons were restricted to software implementations on various platforms [106].

The following AES requirements, set forth by NIST, may be found in [4]:

- The algorithm must implement symmetric-key cryptography.
- The algorithm must be a block cipher.
- The algorithm must support key lengths of 128, 192, and 256 bits.

- The algorithm must support block lengths of 128 bits.

As described in Section 1.2, the advantages of a software implementation include ease of use, ease of upgrade, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [43, 125]. Conversely, cryptographic algorithms (and their associated keys) that are implemented in hardware are, by nature, more physically secure as they cannot easily be read or modified by an outside attacker [43]. The down side of traditional (ASIC) hardware implementations are the lack of flexibility with respect to algorithm and parameter switching. Reconfigurable hardware devices such as FPGAs are a promising alternative for the implementation of block ciphers. FPGAs are hardware devices whose function is not fixed and which can be programmed in-system. The potential advantages of encryption algorithms implemented in FPGAs include:

- Algorithm Agility
- Algorithm Upload
- Algorithm Modification
- Architecture Efficiency
- Throughput
- Cost Efficiency

In order to design a reconfigurable architecture that is able to efficiently implement a wide range of block ciphers, a finite set of ciphers must be identified for initial analysis. What follows is an investigation of four of the five AES candidate algorithm finalists to determine the nature of their underlying components. The characterization of the algorithms' components will lead to a discussion of the hardware architectures best suited for implementation of the AES candidate algorithm finalists. A performance metric to measure the hardware cost for the throughput achieved by each algorithm's implementations will be developed and a target FPGA will be chosen so as to yield implementations that are optimized for high-throughput operation within the commercially available device. Finally, multiple architecture options of the algorithms within the targeted FPGA will be discussed and the overall performance of the implementations will be evaluated versus typical software implementations. Note that in this discussion, we will only consider 128-bit key lengths and that key scheduling is not considered. Subkeys and key-dependent S-Boxes are assumed to be precomputed. This assumption does not affect algorithm throughput as the values change only when a new key is loaded.

4.1 Methodology

4.1.1 Design Methodology

There are two basic hardware design methodologies currently available: language based (high level) design and schematic based (low level) design. Language based design relies upon synthesis tools to implement the desired hardware. While synthesis tools continue to improve, they rarely achieve the most optimized implementation in terms of both area and speed when compared to a schematic implementation. As a result, synthesized designs tend to be (slightly) larger and slower than their schematic based counterparts. Additionally, implementation results can vary greatly depending on the synthesis tool as well as the design being synthesized, leading to potentially increased variances in the synthesized results when comparing synthesis tool outputs. This situation is not entirely different from a software implementation of an algorithm in a high-level language such as C, which is also dependent on coding style and compiler quality. As shown in [115], schematic based design methodologies are no longer feasible for supporting the increase in architectural complexity evidenced by modern FPGAs. As a result, a language based design methodology was chosen as the implementation form for the AES candidate algorithm finalists, with VHDL being the specific language chosen.

4.1.2 Implementations — General Considerations

Each AES candidate algorithm finalist was implemented in VHDL, which led to the use of a bottom-up design and test methodology [115]. The same hardware interface was used for each of the implementations. In an effort to achieve the maximum possible efficiency, key scheduling and decryption were not implemented for each of the AES candidate algorithm finalists. Because FPGAs may be reconfigured in-system, the FPGA may be configured for key scheduling and then later reconfigured for either encryption or decryption. This option is a major advantage of FPGA implementations over classical ASIC implementations. Note that should the overall system include an external processor to control FPGA reconfiguration, this processor could be used instead of the FPGA to perform the key scheduling.

Round keys for encryption are loaded from the external key bus and are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. Each implementation was simulated for functional correctness using the test vectors provided in the AES submission package [13, 37, 119, 127]. After verifying the functionality of the implementations, the VHDL code was synthesized, placed and routed, and resimulated with annotated timing using the same test vectors, verifying that the implementations were successful.

4.1.3 Selection of a Target FPGA

Modern FPGAs have a structure comprised of a two-dimensional array of configurable function units interconnected via horizontal and vertical routing channels. Configurable function units are typically comprised of look-up-tables and flip-flops. Look-up-tables may be configured as either combinational logic or memory elements. Additionally, many modern FPGAs provide variable-size SRAM blocks that may be used as either memory elements or look-up-tables [29].

When examining the AES candidate algorithm finalists for hardware implementation within an FPGA, a number of key aspects emerge. First, it is obvious that the implementation will require a large amount of I/O pins to fully support the 128-bit data stream at high speeds, where bus multiplexing is not an option. It is desirable to decouple the 128-bit input and output data streams to allow for a fully pipelined architecture. Since the round keys cannot change during the encryption process, they may be loaded via a separate key input bus prior to the start of encryption. Additionally, to implement a fully pipelined architecture requires 128-bit wide pipeline stages, resulting in the need for a register-rich architecture to achieve a fast, synchronous implementation. Moreover, it is desirable to have as many register bits as possible per each of the FPGA's configurable units to allow for a regular layout of design elements as well as to minimize the routing required between configurable units. Finally, it is critical that fast carry-chaining be provided between the FPGA's configurable units to maximize the performance of AES candidate algorithm finalists that utilize arithmetic operations [47, 117], a feature commonly available in commercial FPGAs at no additional cost.

In addition to architectural requirements, scalability and cost must also be considered. We believe that the chosen FPGA should be the most high-end chip available at that time, capable of providing the largest amount of hardware resources as well as being highly flexible so as to yield optimal performance. Unfortunately, the cost associated with current high-end FPGAs is relatively high (several hundred US dollars per device). However, it is important to note that the FPGA market has historically evolved at an extremely rapid pace, with larger and faster devices being released to industry at a constant rate. This evolution has resulted in FPGA cost-curves that decrease sharply over relatively short periods of time. Hence, selecting a high-end device provides the closest model for the typical FPGA that will be available over the lifespan of AES.

Based on the aforementioned considerations, the Xilinx Virtex XCV1000BG560-4 FPGA was chosen as the target device. The XCV1000 has 128K bits of embedded RAM divided among thirty two RAM blocks that are separate from the main body of the FPGA. The 560-pin ball grid array package provides 512 usable I/O pins. The XCV1000 is comprised of a 64 × 96 array of look-up-table based Configurable Logic Blocks (CLBs), each of which acts as a 4-bit element comprised of two 2-bit slices for a total of 12,288 CLB slices. Additionally, the XCV1000 may be configured to provide a maximum of 384K bits of RAM independant of the embedded RAM [159]. This type of configuration results in a highly flexible architecture that will accommodate the round functions' use of wide-operand functions. Note that the XCV1000 also appears to be a good representative for a modern FPGA and that devices from other vendors are not fundamentally different. It is thus hoped that our results carry over, within limits, to other devices.

4.1.4 Design Tools

FPGA Express by Synopsys, Inc. and Synplify by Synplicity, Inc. were used to synthesize the VHDL implementations of the AES candidate algorithm finalists. As this study places a strong focus on high throughput implementations, the synthesis tools were set to optimize for speed. As will be discussed in Section 4.4, the resultant implementations exhibit the best possible throughputs with the associated cost being an increase in the area required in the FPGA for each of the implementations. Similarly, when the synthesis tools were set to optimize for area, the resultant implementations exhibit reduced area requirements at the cost of decreased throughput. However, this theory does not always hold true for certain algorithms and architectures. This contradiction is caused by the underlying proprietary synthesis tool algorithms — different synthesis algorithms tend to yield different implementations for the same VHDL code. As will be evidenced in later discussions, these synthesis algorithms may lead to unexpected results based on how a design implementation is interpreted.

XACTstep 2.1i by Xilinx, Inc. was used to place and route the synthesized implementations. For the sub-pipelined architectures, a 40 MHz timing constraint was used in both the synthesis and place-and-route processes as it resulted in significantly higher system clock frequencies. However, the 40 MHz timing constraint was found to have little affect on the other architecture types, resulting in nearly identical system clock frequencies to those achieved without the timing constraint.

Finally, Speedwave by Viewlogic Systems, Inc. and Active-HDL^{TM} by ALDEC, Inc. were used to perform behavioral and timing simulations for the implementations of the AES candidate algorithm finalists. The simulations verified both the functionality and the ability to operate at the designated clock frequencies for the implementations.

4.1.5 Evaluation Metrics

When evaluating a given implementation, the throughput of the implementation and the hardware resources required to achieve this throughput are usually considered the most critical parameters. Other aspects, such as power consumption, are not addressed in this study. No established metric exists to measure the hardware resource costs associated with the measured throughput of an FPGA implementation. To address this issue, the Xilinx XCV1000 FPGA must be examined. Two area measurements are readily apparent — logic gates and CLB slices. It is important to note that the logic gate count does not yield a true measure of how much of the FPGA is actually being used. Hardware resources within CLB slices may not be fully utilized by the place-and-route software so as to relieve routing congestion. This results in an increase in the number of CLB slices without a corresponding increase in logic gates. To achieve a more accurate measure of chip utilization, CLB slice count was chosen as the most reliable area measurement. Therefore, to measure the hardware resource cost associated with an implementation's resultant throughput, the Throughput Per Slice (TPS) metric is used. We defined TPS as:

$$TPS \quad := \quad \text{(Encryption Rate)/(\# CLB Slices Used)}$$

When appropriate, the TPS metric will be normalized. Therefore, the optimal implementation will display the highest throughput and have the largest TPS. Note that the TPS metric behaves inversely to the classical time-area (TA) product.

When comparing implementations using the TPS and throughput metrics, it is required that the architectures are implemented on the same FPGA. Different FPGAs within the same family yield different timing results as a function of available logic and routing resources, both of which change based on the die size of the FPGA. Additionally, it is impossible to legitimately compare FPGAs from separate families (such as comparing an implementation on a Xilinx Virtex FPGA with an implementation on a Xilinx XC4000 FPGA) as each family of FPGAs has a unique architecture. Quite obviously, different FPGA architectures will greatly affect the TPS and throughput measurements — each architecture may be comprised of different logic elements and have different amounts of logic and routing resources. Therefore, all of the implementations were performed on the Xilinx Virtex XCV1000BG560-4 to guarantee consistency.

Finally, it is critical to note that TPS and throughput may not scale linearly based on the number of rounds implemented for the three architecture types detailed in Section 4.2.5. Therefore, it is imperative to examine multiple implementations for each architecture type, varying the round count to determine the optimal number of rounds per implementation. As will be shown in Section 4.3, TPS and throughput do not scale linearly for certain algorithms when implemented using some or all of the three architecture types.

The number of CLBs required as well as the maximum operating frequency for each implementation was obtained from the Xilinx place-and-route report files. The computed throughput for implementations that employ any form of hardware pipelining (to be discussed in Section 4.2) are made assuming that the pipeline latency has been met.

4.2 Architecture Options and the AES Candidate Algorithm Finalists

Before attempting to implement the AES candidate algorithm finalists in hardware, it is important to understand the nature of each algorithm as well as the hardware architectures most suited for their implementation. What follows is an investigation into the key components of the AES candidate algorithm finalists. Based on this breakdown, a discussion is presented on the hardware architectures most suited for implementation of the AES candidate algorithm finalists.

4.2.1 RC6

RC6 was submitted as an AES candidate algorithm by Ron Rivest and RSA Laboratories [119]. RC6 is an extension of the RC5 algorithm and contains much of the same structure [119] while being designed to meet AES requirements as well as to provide speed and security improvements over RC5. As a result, much of the security analysis performed on RC5 applies to RC6 [34]. The choice of RC6 as an AES candidate algorithm finalist was due to the algorithm's simplicity, speed and ease of implementation, inherited security from its predecessor RC5, and fast key setup [107].

RC6 is specified as a family of algorithms with three parameters. RC6-$w/r/b$ refers to an algorithm with a word size of w bits, r rounds, and a key size of b bytes. For this discussion, we consider RC6-32/20/16 which conforms to the AES specifications with a key size of 128 bits. Additionally, we assume subkeys to have been precomputed and supplied in a 44-bit × 32-bit array S.

Plaintext is divided into four 32-bits blocks, A, B, C, and D. Encryption begins with key whitening, accomplished by adding keys S[0] and S[1] to plaintext blocks B and D, respectively. Following key whitening, the RC6 round operation is performed on A, B, C, and D. Figure 4.1 shows the structure of an RC6 encryption round, where shifts are denoted by $\ll$, rotations are denoted by $\lll$, and XORs are denoted by $\oplus$. Decryption is accomplished by applying this structure in reverse order with the corresponding keys.

The RC6 round is comprised of a modified Feistel network [23] which operates on two of the four 32-bit data blocks. Blocks B and D are fed into the f function and are shifted left by one bit, thus introducing a zero in the least significant bit (LSB). The LSB is then set to one and this value is multiplied modulo 2^{32} by the original f function input. The result of the f function is left rotated by $\log_2(w)$ bits — for RC6-32/20/16, w is 32, resulting in $\log_2(w)$ being 5. The resultant rotated output is XORed with the adjacent block (block B's result is XORed with block A, and block D's result is XORed with block C). Finally, these results are left rotated by a variable number of bits corresponding to a value determined from the five LSBs of the non-adjacent f function result. Thus, block A is left rotated by 0 to 31 bits based on the five LSBs in the result of block D's f function and fixed $\log_2(w)$-bit left rotation. Block C is modified in a similar manner by block B's result. To complete the round function, blocks A and C are added modulo 2^{32} to their associated subkeys and blocks A, B, C, and D are cyclicly left shifted by one block. After twenty rounds have been performed, output whitening is performed on blocks A and C in a manner similar to input whitening.

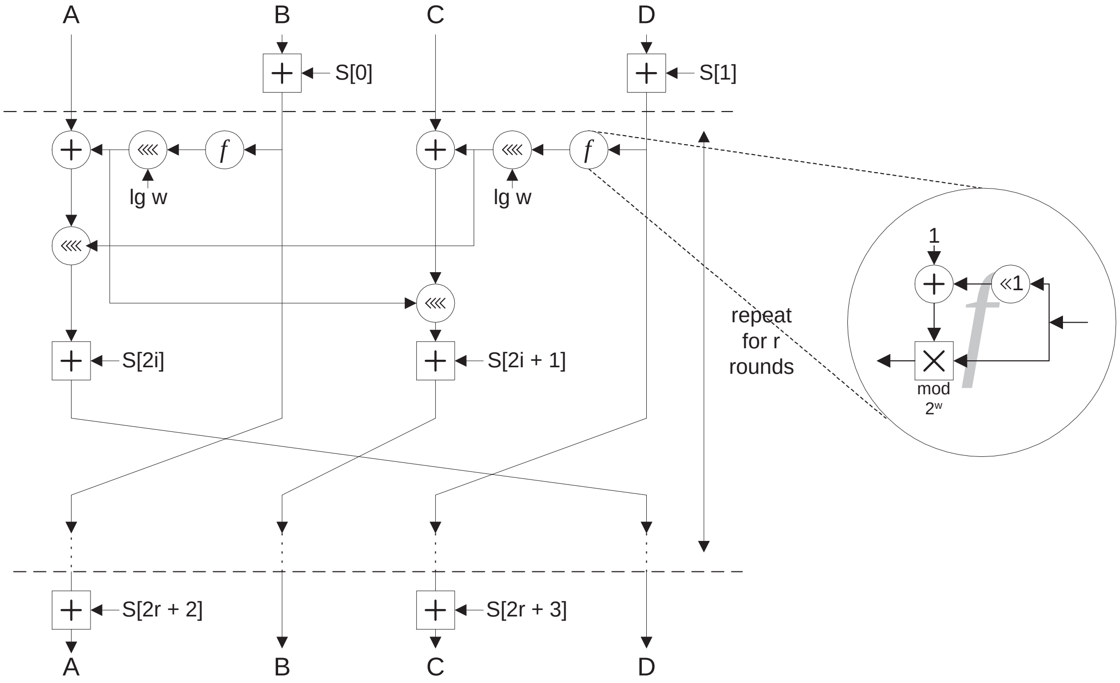
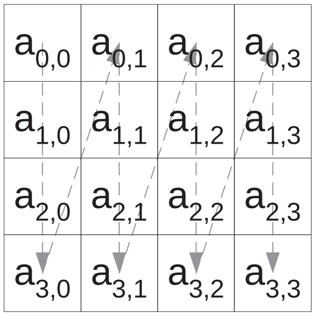


Figure 4.1: Block diagram of an RC6 encryption round

4.2.2 Rijndael

Rijndael was submitted as an AES candidate algorithm by Joan Daemen and Vincent Rijmen [37]. 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 [107].

Rijndael maps plaintext into a rectangular 4 × *Nb* array of bytes as shown in Figure 4.2. *Nb* is defined as the block length divided by thirty two — for AES, *Nb* is set to four. The most significant byte of plaintext is mapped to $a_{0,0}$ while the next significant byte is mapped to $a_{1,0}$. The mapping continues, as the arrows indicate, with the least significant byte being mapped to $a_{3,3}$. Within a given byte in the array, $a_{r,c}$, the first number, r, refers to the row number while the second number, c, refers to the column number. Similarly, the Cipher Key is mapped into a rectangular 4 × *Nk* array of bytes as per Figure 4.2. *Nk* is defined as the key length divided by thirty two — for AES, *Nk* may be four, six, or eight for key lengths of 128, 192, and 256 bits, respectively.



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Figure 4.2: Rijndael plain-text mapping

Figure 4.3 details the Rijndael round structure. For the first nine rounds, the round input passes through the ByteSub, ShiftRow, MixColumn, and AddRoundKey functions. For round ten, the final round, the MixColumn function is eliminated.

The ByteSub transformation performs a bytewise substitution on all bytes of the input array. The substitution is nonlinear and invertible [37]. The ShiftRow transformation performs a bytewise right rotation on the rows of the array output from the ByteSub operation — see Figure 4.4. The first row, row 0, is not rotated, row 1 is rotated by one byte, row 2 by two bytes, and row 3 by three bytes, resulting in the values that were previously aligned by column now being aligned along a right diagonal. This operation is analogous to the data swapping characteristic of the Feistel structure found in other algorithms.

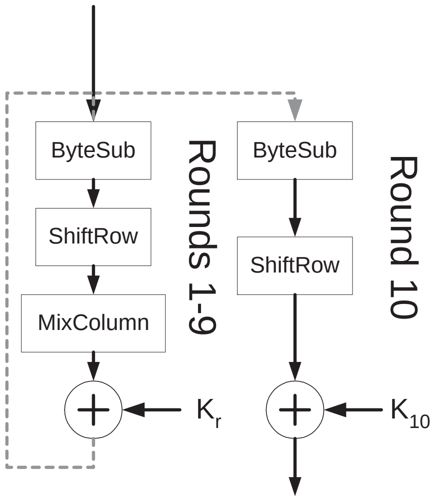


Figure 4.3: Rijndael block diagram

The MixColumn operates on each column independently within the array output from the ShiftRow operation. Within a given column, each byte is considered a polynomial over the Galois Field $GF(2^8)$ (see [37, 92] for a discussion of operations in Galois Fields). Figure 4.5 depicts the MixColumn transformation with input column a and output column b. To illustrate the transformation, the output equation for b_0 is:

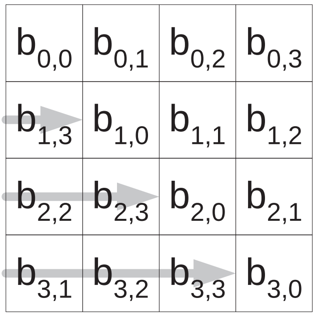
$$b_0 = (02 \times a_0) \oplus (03 \times a_1) \oplus (01 \times a_2) \oplus (01 \times a_3)$$

All multiplications and additions are in the field $GF(2^8)$. Addition within $GF(2^8)$ is defined as the bitwise XOR of both arguments. Multiplication in a Galois Field is dependent upon the field polynomial. For Rijndael, the field polynomial is given as:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \text{ [37]}$$

Note that for $GF(2^8)$ with field polynomial $m(x) = x^8 + x^5 + x^3 + x^2 + 1$, the average optimized complexity of a constant multiplier is 14.4 XOR gates [110].

The final operation, AddRoundKey is the XORing of the previous MixColumn or ShiftRow with the round's associated subkey.



$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,3}$	$b_{1,0}$	$b_{1,1}$	$b_{1,2}$
$b_{2,2}$	$b_{2,3}$	$b_{2,0}$	$b_{2,1}$
$b_{3,1}$	$b_{3,2}$	$b_{3,3}$	$b_{3,0}$

Figure 4.4: Rijndael ShiftRow transformation

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

Figure 4.5: Rijndael MixColumn transformation

4.2.3 Serpent

Serpent was submitted as an AES candidate algorithm by Ross Anderson, Eli Biham, and Lars Knudsen [13]. Serpent was designed by one of the inventors of differential cryptanalysis [24] and has been strengthened against this type of attack. As detailed in [107], Serpent was chosen as an AES candidate algorithm finalist for the algorithm's large security margin, efficiency of the optimized bit-slice implementation, ease of analysis, and low memory requirements. Note that a high security margin implies that an algorithm has been designed with more rounds than are necessary to guarantee security [107].

Figure 4.6 shows the structure of the Serpent algorithm. 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.

The Serpent S-Boxes are 4-bit × 4-bit fixed look-up-tables. For a given round, thirty

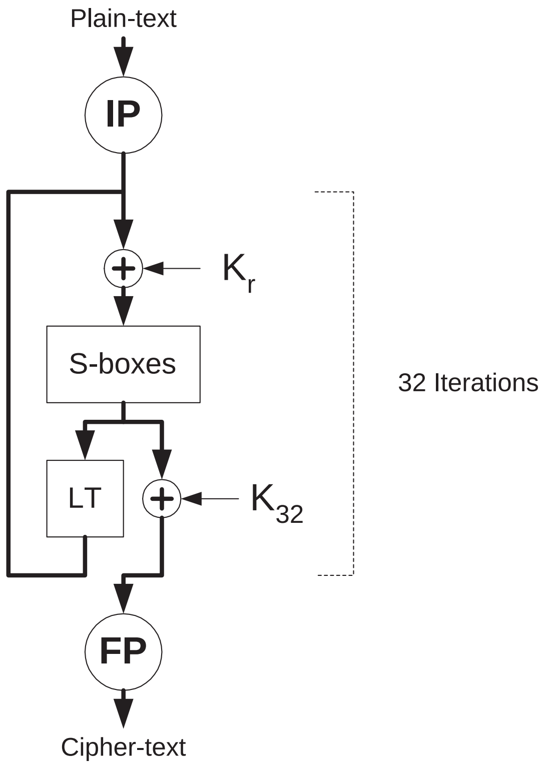


Figure 4.6: Serpent block diagram

two copies of the same S-Box are applied to the 128-bit input. There are a total of eight distinct S-Boxes, S_0 through S_7. Round 0 uses S_0, Round 1 uses S_1, etc. S_0 is used again in Rounds 8, 16, and 24, resulting in each S-Box being used four times. Given the small number of inputs per S-Box, a Boolean logic reduction becomes feasible based on the chosen target technology. The Xilinx Virtex CLB slice contains at least two 4-input look-up-tables [160]. Thus, given that each S-Box has four outputs, one S-Box may be implemented in two CLB slices. This is significant as the number of CLBs required to implement an S-Box increases dramatically as the number of S-Box inputs rises.

Figure 4.7 details the Serpent linear transformation. In this representation, known as bit-slice mode, the 128-bit input is broken into four 32-bit words. Shifts are denoted by $\ll$, rotations are denoted by $\lll$, and XORs are denoted by $\oplus$. Bit-slice mode is an alternative method of implementation that is efficient when operating on 32-bit words [22]. Note that the initial and final permutations are omitted if Serpent is implemented in bit-slice mode.

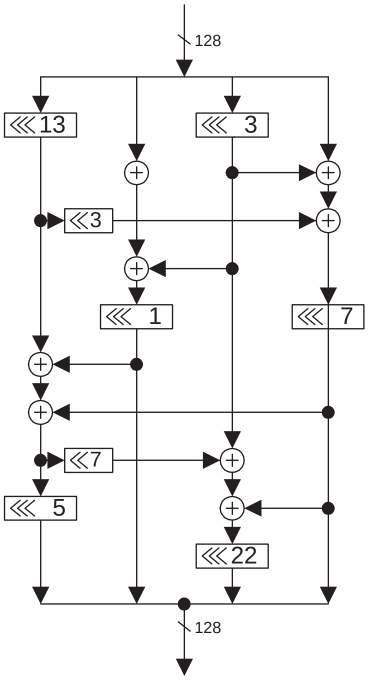


Figure 4.7: Serpent linear transformation — bit-slice mode

4.2.4 Twofish

Twofish was submitted as an AES candidate algorithm finalist by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson [127]. The choice of Twofish as an AES candidate algorithm finalist was due to the algorithm's fast performance across platforms, low memory requirements, and variety of optimization options [107]. Twofish can be optimized for speed, key setup, memory, code size in software, or space when implemented in hardware [127].

As in RC6, Twofish breaks the 128-bit plaintext into the four 32-bit words which enter a modified Feistel network after passing through an input whitening stage. Figure 4.8 details the Twofish encryption process. The plaintext P is divided into four words A, B, C, and D

such that the most significant word of P is loaded into A and the least significant word is loaded into D. Each block passes through the input whitening stage where they are XORed with the appropriate subkeys. Following input whitening, sixteen rounds of processing are performed. A round begins by passing blocks A and B into the function F. Block A passes directly into one of two g functions while block B passes into the other g function after being rotated left eight bits. Each input byte to g passes through a different key-dependent S-Box that is initialized during the key-setup phase.

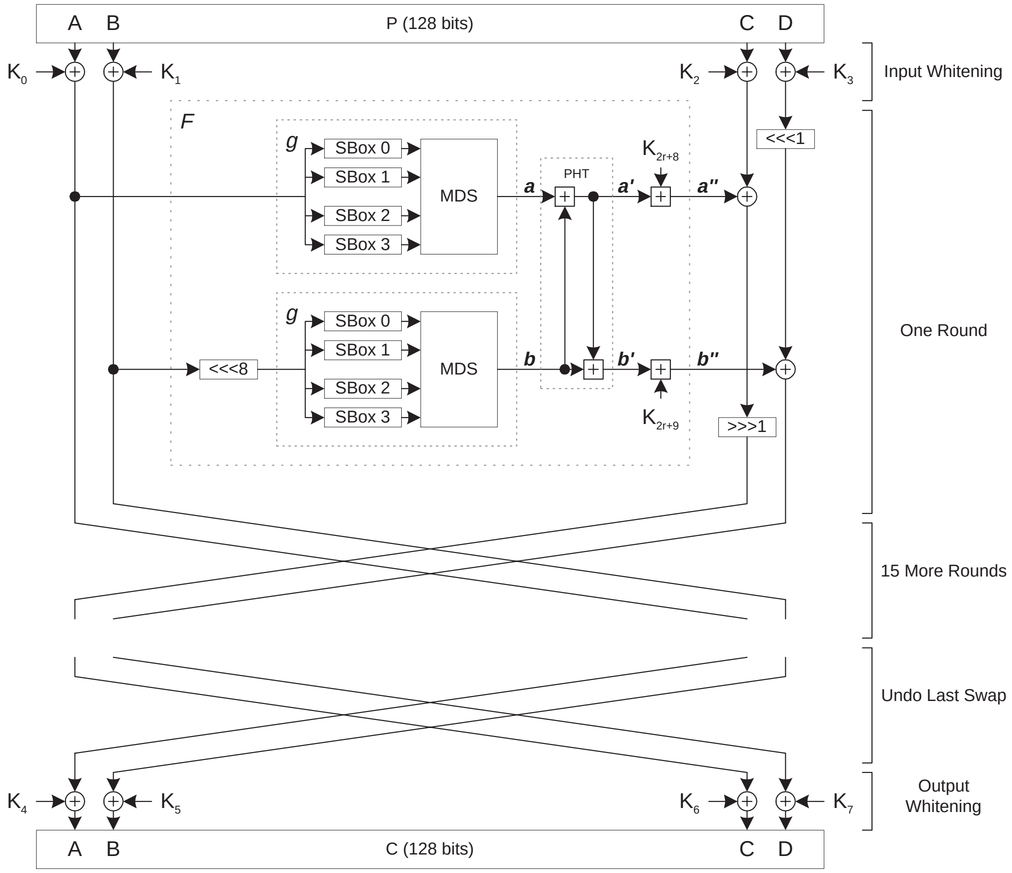


Figure 4.8: Twofish block diagram

The output of the S-Boxes pass into the Maximum Distance Separable (MDS) coder, an operation analogous the Rijndael MixColumn transformation where multiplication over $GF(2^8)$ is performed (see [162, 92] for a discussion of the mathematics of MDS coding).

However, Twofish employs both a different fixed field and a different field polynomial versus Rijndael. Figure 4.9 details the MDS transformation with input column y and output column z.

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} \mathrm{01} & \mathrm{EF} & \mathrm{5B} & \mathrm{5B} \\ \mathrm{5B} & \mathrm{EF} & \mathrm{EF} & \mathrm{01} \\ \mathrm{EF} & \mathrm{5B} & \mathrm{01} & \mathrm{EF} \\ \mathrm{EF} & \mathrm{01} & \mathrm{EF} & \mathrm{5B} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Figure 4.9: Twofish MDS matrix

Each of the elements in the MDS are polynomials defined over $GF(2^8)$ using the field polynomial:

$$v(x) = x^8 + x^6 + x^5 + x^3 + 1$$

The output of both g functions are fed into the Pseudo-Hadamard Transform (PHT). Given inputs a and b, the PHT outputs a' and b' are formed by the following equations:

$$a' = a + b \mod 2^{32}$$
$$b' = a + 2b \mod 2^{32}$$

The outputs of the PHT are added modulo 2^{32} to the associated subkeys of the given round to form the outputs a'' and b''. C is then XORed with a'' and then rotated right one bit. Block D is rotated left one bit and then XORed with b''. The resultant C and D are passed respectively into blocks A and B of the next round. The original A and B are passed respectively into C and D of the next round. After the completion of sixteen rounds, an additional swap of words is performed to undo the swap performed in the sixteenth round. Finally, an output whitening operation is performed where A, B, C, and D are XORed with the corresponding subkeys.

4.2.5 Core Operations of the AES Candidate Algorithm Finalists

Algorithm	XOR	Mod 2^{32} Add	Mod 2^{32} Sub	Fixed Shift	Var. Rot.	Mod 2^{32} Mult	GF(2^8) Constant Mult	S-Box
MARS	●	●	●	●	●	●		●
RC6	●	●		●	●	●		
Rijndael	●			●			●	●
Serpent	●			●				●
Twofish	●	●		●			●	●

Table 4.1: AES candidate algorithm finalists core operations [32]

Table 4.1 details the core operations of each of the AES candidate algorithm finalists. In terms of complexity, the operation detailed in Table 4.1 which requires the most hardware resources as well as computation time is the modulo 2^{32} multiplication [32]. Implementing wide multipliers in hardware is an inherently costly task that requires significant hardware resources. S-Boxes may be implemented in either combinatorial logic or embedded RAM — the advantages of each of these options are discussed in Section 4.2.6. Fast operations such as bit-wise XOR, modulo 2^{32} addition and subtraction, fixed value shifting, and variable rotations are constructed from simple hardware elements. Additionally, the Galois field multiplications required in Rijndael and Twofish can also be implemented very efficiently in hardware as they are multiplications by a constant. Galois field constant multiplication requires simple Boolean linear equations as compared to general multiplications [110].

4.2.6 Hardware Architectures

The AES candidate algorithm finalists are all comprised of a basic looping structure (some form of either Feistel or substitution-permutation network) whereby data is iteratively passed through a round function. Based on this looping structure, the following architecture options were investigated so as to yield optimized implementations:

- Iterative Looping
- Loop Unrolling
- Partial Pipelining
- Partial Pipelining with Sub-Pipelining

Iterative looping over a cipher's round structure is an effective method for minimizing the hardware required when implementing an iterative architecture. When only one round is implemented, an n-round cipher must iterate n times to perform an encryption. This approach has a low register-to-register delay but requires a large number of clock cycles to perform an encryption. This approach also minimizes in general the hardware required for round function implementation but can be costly with respect to the hardware required for round key and S-Box multiplexing. This cost may be alleviated by configuring the 4-bit look-up-tables within the Xilinx Virtex CLB slice to operate in RAM mode and then storing one bit of each round key within the look-up-table. One look-up-table would support up to sixteen round keys, leading to an implementation requiring k look-up-tables to store round keys that are k bits in length, removing the need to store each round key in a separate register. Additionally, multiple banks of look-up-tables may be used to support algorithms that require more than sixteen round keys. These look-up-table banks would then be sequentially addressed based on the current round to access the appropriate key. However, this methodology is less efficient when multiple rounds are unrolled or pipelined as each round requires its own bank of associated look-up-tables for round key storage. This methodology becomes completely inefficient for a fully unrolled or fully pipelined architecture as each look-up-table bank would only contain a single round key. Because this methodology does not provide a performance enhancement for all architectures, its use was not considered for this study to maintain consistency between architectures. Iterative looping is shown in Figure 4.10.

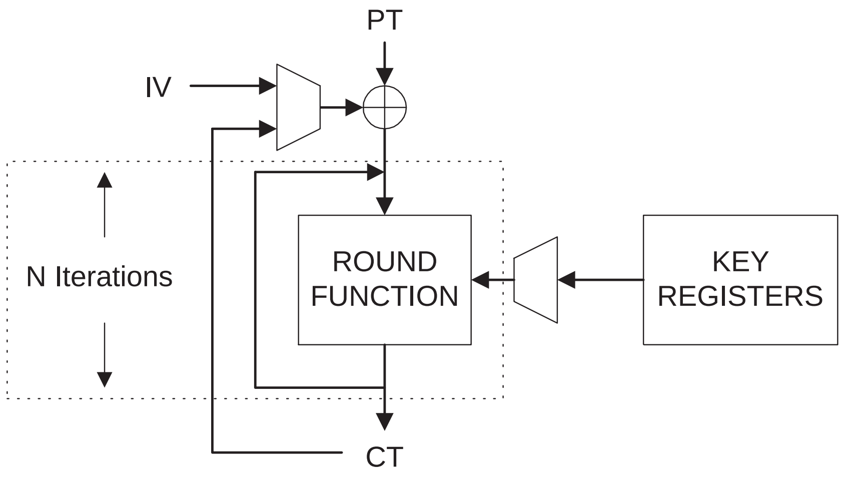


Figure 4.10: Iterative looping

Iterative looping is a subset of loop unrolling in that only one round is unrolled whereas a loop unrolling architecture allows for the unrolling of multiple rounds, up to the total number of rounds required by the cipher. As opposed to an iterative looping architecture, a loop unrolling architecture where all n rounds are unrolled and implemented as a single combinatorial logic block maximizes the hardware required for round function implementation while the hardware required for round key and S-Box multiplexing is completely eliminated. However, while this approach minimizes the number of clock cycles required to perform an encryption, it maximizes the worst case register-to-register delay for the system, resulting in an extremely slow system clock. Partial loop unrolling is shown in Figure 4.11 while full loop unrolling is shown in Figure 4.12.

A partially pipelined architecture offers the advantage of high throughput rates by increasing the number of blocks of data that are being simultaneously operated upon. This is achieved by replicating the round function hardware and registering the intermediate data between rounds. Moreover, in the case of a full-length pipeline (a specific form of a partial pipeline), the system will output a 128-bit block of ciphertext at each clock cycle once the latency of the pipeline has been met. However, an architecture of this form requires significantly more hardware resources as compared to a loop unrolling architecture. In a

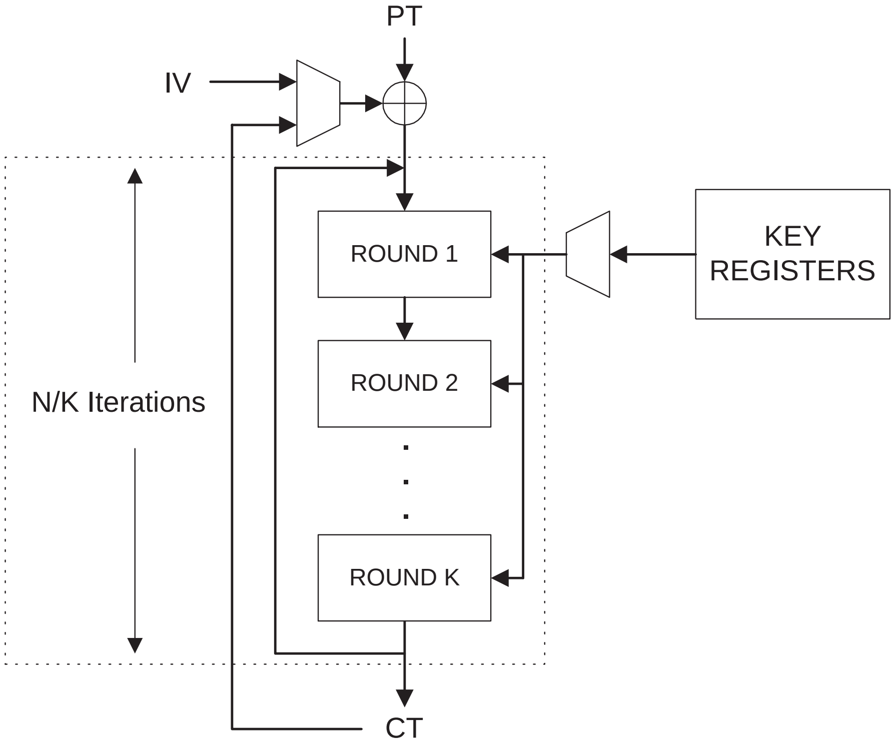


Figure 4.11: Partial loop unrolling

partially pipelined architecture, each round is implemented as the pipeline's atomic unit and the rounds are separated by the registers that form the actual pipeline. However, many of the AES candidate algorithm finalists cannot be implemented using a full-length pipeline due to the large size of their associated round function and S-Boxes, both of which must be replicated n times for an n-round cipher. As such, these algorithms must be implemented as partial pipelines. Additionally, a pipelined architecture can be fully exploited only in modes of operations which do not require feedback of the encrypted data, such as ECB or Counter Mode. When operating in feedback modes such as CFB, CBC, or OFB, the ciphertext of one block must be available before the next block can be encrypted. As a result, multiple blocks of plaintext cannot be encrypted in a pipelined fashion when operating in feedback modes. Partial pipelining is shown in Figure 4.13.

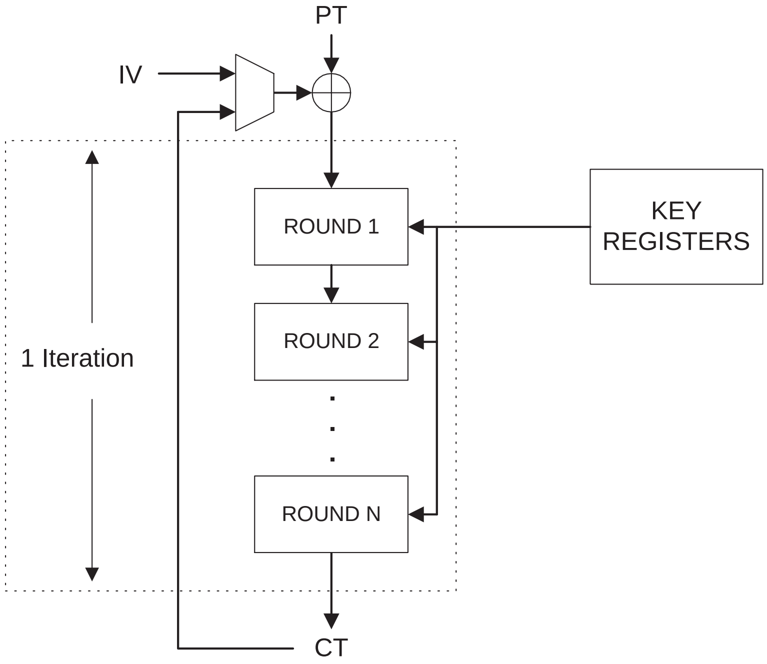


Figure 4.12: Full loop unrolling

Sub-pipelining a (partially) pipelined architecture is advantageous when the round function of the pipelined architecture is complex, resulting in a large delay between pipeline stages. By adding sub-pipeline stages, the atomic function of each pipeline stage is subdivided into smaller functional blocks. This results in a decrease in the pipeline's delay between stages. However, each sub-division of the atomic function increases the number of clock cycles required to perform an encryption by a factor equal to the number of subdivisions. At the same time, the number of blocks of data that may be operated upon in non-feedback mode is increased by a factor equal to the number of sub-divisions. Therefore, for this technique to be effective, the worst case delay between stages must be decreased by a factor of m where m is the number of added sub-divisions. Partial pipelining with sub-pipelining is shown in Figure 4.14.

Many FPGAs provide embedded RAM which may be used to replace the round key and S-Box multiplexing hardware. By storing the keys within the RAM blocks, the appropriate key may be addressed based on the current round. However, due to the limited number

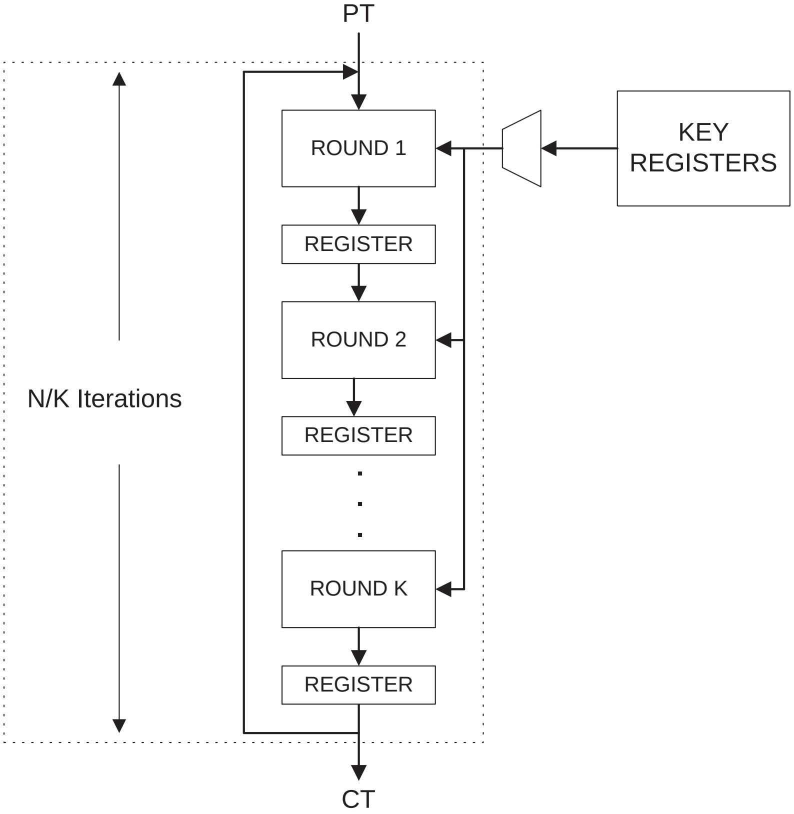


Figure 4.13: Partial pipelining

of RAM blocks, as well as their restricted bit width, this methodology is not feasible for architectures with many pipeline stages or unrolled loops. Those architectures require more RAM blocks than are typically available. Additionally, the switching time for the RAM is more than a factor of three longer than that of a standard Xilinx Virtex CLB slice element, resulting in the RAM element having a lesser speed-up effect on the overall implementation. Therefore, the use of embedded RAM is not considered for this study to maintain consistency between architectural implementations.

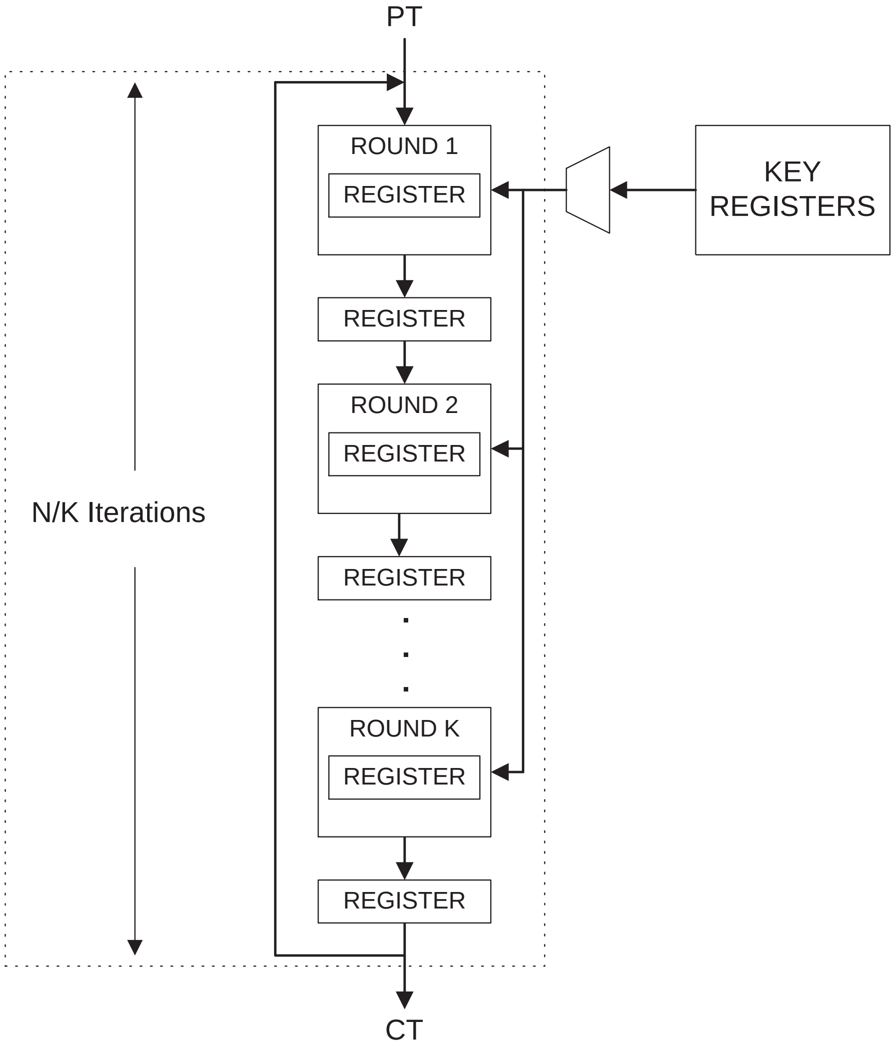


Figure 4.14: Partial pipelining with sub-pipelining

4.3 Architectural Implementation Analysis

For each of the AES candidate algorithm finalists, the four architecture options described in Section 4.2.6 were implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. Round keys are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. These implementations yielded a great deal of knowledge in regards to the FPGA suitability of each AES candidate algorithm finalist. What follows is a discussion of the knowledge gained regarding each algorithm when implemented using the four architecture types.

4.3.1 Architectural Implementation Analysis — RC6

When implementing the RC6 algorithm, it was first determined that the RC6 modulo 2^{32} multiplication was the dominant element of the round function in terms of required logic resources. Each RC6 round requires two copies of the modulo 2^{32} multiplier. However, it was found that the RC6 round function does not require a general modulo 2^{32} multiplier. The RC6 multipliers implement the function $A(2A + 1)$ which may be implemented as $2A^2 + A$. Therefore, the multiplication operation was replaced with an array squarer with summed partial products, requiring fewer hardware resources and resulting in a faster implementation. The remaining components of the RC6 round function — fixed and variable shifting, bit-wise XOR, and modulo 2^{32} addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. Finally, it was found that the synthesis tools could not minimize the overall size of an RC6 round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire twenty rounds of the algorithm within the target FPGA. The results for each of the RC6 implementations using the four architecture types is found in Appendix A.

The 2-stage partial pipeline architecture optimized for speed was found to yield both the highest throughput and TPS when operating in feedback mode, outperforming the single round iterative looping implementation by achieving a significantly higher system clock frequency. When operating in non-feedback mode, a partially pipelined architecture optimized for speed with two additional sub-pipeline stages was found to offer the advantage of extremely high throughput rates and TPS once the latency of the pipeline was met, with the 10-stage partial pipeline implementation displaying the best throughput and TPS results. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly two thirds of the round function's associated delay was attributed to the modulo 2^{32} multiplier. Therefore, two additional pipeline sub-stages were implemented so as to subdivide the multiplier into smaller blocks, resulting in a total of three pipeline stages per round function. Further sub-pipelining was not implemented as this would require subdividing the adders used to sum the partial products (a non-trivial task) to balance the delay between sub-pipeline stages.

4.3.2 Architectural Implementation Analysis — Rijndael

When implementing the Rijndael algorithm, it was first determined that the Rijndael S-Boxes were the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is implemented as an 8-bit to 8-bit look-up-table, requiring significant hardware resources. However, the remaining components of the Rijndael round function — byte swapping, constant Galois field multiplication, and key addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA. The results for each of the Rijndael implementations using the four architecture types are found in Appendix A.

2-stage loop unrolling optimized for speed was found to yield the highest throughput while iterative looping optimized for speed was found to yield the highest TPS when operating in feedback mode. However, the measured throughput for the iterative looping implementation was within 10% of the 2-stage loop unrolled implementation. Due to the probabilistic nature of the place-and-route algorithms, one can expect a variance in performance based on differences in the starting point of the process. When performing this process multiple times, known as multi-pass place-and-route, it is likely that the single round implementation would achieve a throughput similar to that of the 2-stage loop unrolled implementation. However, the use of multi-pass place-and-route was beyond the scope of this investigation and was therefore not performed.

When operating in non-feedback mode, partial pipelining was found to offer the advantage of extremely high throughput rates once the pipeline latency was met. 5-stage partial pipelining with one sub-pipeline stage optimized for speed was found to yield the best throughput results while 2-stage partial pipelining with one sub-pipeline stage optimized for speed was found to yield the best TPS results. However, the measured TPS for the 5-stage partial pipeline with one sub-pipeline stage was within 10% of the 2-stage partial pipeline with one-subpipeline stage. As is the case in feedback mode, it is likely that the 5-stage partial pipeline with one sub-pipeline stage would achieve a TPS similar to the 2-stage partial pipeline with one sub-pipeline stage should multi-pass place-and-route be performed. While Rijndael cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architectures. Sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Rijndael round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

4.3.3 Architectural Implementation Analysis — Serpent

When implementing the Serpent algorithm, it was first determined that since the Serpent S-Boxes are relatively small (4-bit to 4-bit), it is possible to implement them using combinational logic as opposed to clocked memory elements. Additionally, the combinatorial logic that comprises the S-Boxes map extremely well to the Xilinx Virtex CLB slice, which is comprised of 4-bit look-up-tables. This allows one S-Box to be implemented in a total of two CLB slices (requiring four 4-bit look-up-tables), yielding a compact implementation which minimizes routing between CLB slices. Finally, the components of the Serpent round function — key masking, S-Box substitution, and linear transformation — were found to be simple in structure, resulting in the round function requiring few hardware resources. The results for each of the Serpent implementations using the four architecture types is found in Appendix A.

Implementing a single round of the Serpent algorithm provides the greatest area-optimized solution. However, a significant performance improvement was achieved by performing partial loop unrolling. When operating in feedback mode, the 8-round loop unrolled implementation optimized for speed was found to yield both the highest throughput and TPS. By removing the need for S-Box multiplexing (as one copy of each possible S-Box grouping is now included within one of the eight rounds) while minimizing the round key multiplexing, significantly higher performance values were seen as compared to those of the 1-round loop unrolled and 32-round loop unrolled architectures.

When operating in non-feedback mode, a full-length pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, outperforming smaller partially pipelined implementations. In the fully pipelined architecture, all of the elements of a given round function are implemented as combinatorial logic. Other AES candidate algorithm finalists cannot be implemented using a fully pipelined architecture due to the larger round functions. However, due to the small size of the Serpent S-Boxes (4-bit look-up-tables), the cost of S-Box replication is minimal in terms of the required hardware. As a result, the TPS exhibited by the full-length pipeline implementation was significantly higher than that of any other implementation when operating in non-feedback mode. Interestingly, for the full-length pipeline implementation, area optimization yielded the highest throughput while speed optimization yielded the best TPS. This speaks to the relative uncertainty of the synthesis results for a given architecture and algorithm as discussed in Section 4.1.4. Finally, sub-pipelining of the partially pipelined architectures was determined to yield no throughput increase. Because the round function components are all simple in structure, there is little performance to be gained by subdividing them with registers in an attempt to reduce the delay between stages. As a result, the increase in the system's clock frequency would not outweigh the increase in the number of clock cycles required to perform an encryption, resulting in a performance degradation.

4.3.4 Architectural Implementation Analysis — Twofish

When implementing the Twofish algorithm using full keying [127], it was first determined that the synthesis tools were unable to minimize the Twofish S-Boxes to the extent of other AES candidate algorithm finalists due to the S-Boxes being key-dependent. Therefore, the overall size of a Twofish round was too large to allow for a fully unrolled or fully pipelined implementation of the algorithm within the target FPGA. Moreover, the key-dependent S-Boxes were found to require nearly half of the delay associated with the Twofish round function. The results for each of the Twofish implementations using the four architecture types is found in Appendix A.

Iterative looping optimized for area was found to yield both the best throughput and TPS when operating in feedback mode. The iterative looping architecture was able to reach a significantly faster system clock frequency as compared to the other loop unrolling and pipeline implementations, resulting in superior performance results. When operating in non-feedback mode, a partially pipelined architecture optimized for area with two additional sub-pipeline stages was found to offer the advantage of extremely high throughput rates and TPS once the latency of the pipeline was met, with the 8-stage partial pipeline implementation displaying the best throughput and TPS results. While Twofish cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architectures. Sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Twofish round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, an additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in non-feedback mode. Further sub-pipelining was implemented by sub-dividing the S-Boxes with a pipeline sub-stage and moving the previously inserted sub-pipeline stage to balance the delay between sub-pipeline stages. This resulted in an increase in system clock frequency by a factor of nearly 3 as compared to the pipelined implementation with no sub-stages. When additional sub-pipelining was investigated, it was found that the logic functions implemented between the two sub-stages were all simple in structure. As a result, further sub-pipelining was not implemented as the increase in clock frequency achieved by adding an additional sub-stage would not outweigh the number of cycles required to perform an encryption, resulting in a performance degradation.

4.4 Performance Evaluation

The data shown in Tables 4.2, 4.3, 4.4, and 4.5 detail the optimal implementations (in terms of throughput and TPS) for each of the AES candidate algorithm finalists. The architecture types — loop unrolling (LU), full or partial pipelining (PP), and partial pipelining with sub-pipelining (SP) — are listed along with the number of stages and (if necessary) sub-pipeline stages in the associated implementation; e.g., LU-4 implies a loop unrolling architecture with four rounds, while SP-2-1 implies a partially pipelined architecture with two stages and one sub-pipeline stage per pipeline stage. As a result, the SP-2-1 architecture implements two rounds of the given cipher with a total of two stages per round. Throughput is calculated as:

$$Throughput := (\text{128 Bits * Clock Frequency}) \ / \ (\text{Cycles Per Encrypted Block})$$

Note that the implementation of a one stage partial pipeline architecture, an iterative looping architecture, and a one round loop unrolled architecture are all equivalent and are therefore not listed separately. Also note that the computed throughput for implementations that employ any form of hardware pipelining (as discussed in Section 4.2) are made assuming that the pipeline latency has been met.

The number of Xilinx Virtex CLBs required as well as the maximum operating frequency for each implementation was obtained from the Xilinx report files. Note that the Xilinx tools assume the absolute worst possible operating conditions — highest possible operating temperature, lowest possible supply voltage, and worst-case fabrication tolerance for the speed grade of the FPGA [8]. As a result, it is common for actual implementations to achieve slightly better performance results than those specified in the Xilinx report files.

Alg.	**Arch.**	**Opt.**	**Cycles Per Enc. Block**	**Throughput (Gbit/s)**
RC6	SP-10-2	Speed	2	**2.40**
Rijndael	SP-5-1	Speed	2.1	**1.94**
Serpent	PP-32	Area	1	**5.04**
Twofish	SP-8-2	Speed	2	**2.40**

Table 4.2: AES candidate algorithm finalists throughput evaluation — best non-feedback mode implementations

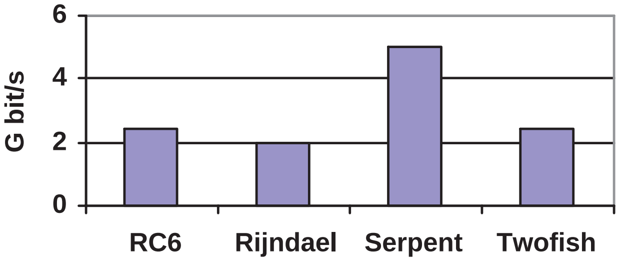


Figure 4.15: Best throughput — non-feedback mode

Alg.	**Arch.**	**Opt.**	**Cycles Per Enc. Block**	**Throughput (Mbit/s)**
RC6	PP-2	Speed	20	**126.5**
Rijndael	LU-2	Speed	6	**300.1**
Serpent	LU-8	Speed	4	**444.2**
Twofish	SP-1-1	Area	32	**127.7**

Table 4.3: AES candidate algorithm finalists throughput evaluation — best feedback mode implementations

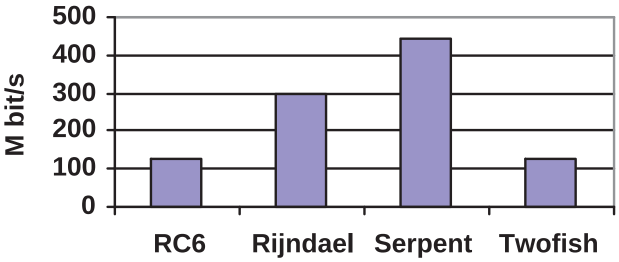


Figure 4.16: Best throughput — feedback mode

Alg.	**Arch.**	**Opt.**	**Slices**	**Key Storage Min. Slices**	**TPS**
RC6	SP-10-2	Speed	10856	704	**220881**
Rijndael	SP-2-1	Speed	4871	704	**194837**
Serpent	PP-32	Speed	9004	2112	**539778**
Twofish	SP-8-2	Area	9486	672	**248666**

Table 4.4: AES candidate algorithm finalists TPS evaluation — best non-feedback mode implementations

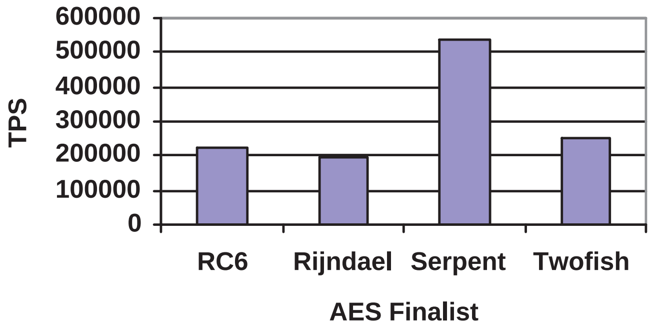


Figure 4.17: Best TPS — non-feedback mode

Alg.	**Arch.**	**Opt.**	**Slices**	**Key Storage Min. Slices**	**TPS**
RC6	PP-2	Speed	3189	704	**39662**
Rijndael	LU-1	Speed	3528	704	**83387**
Serpent	LU-8	Speed	7964	2112	**55771**
Twofish	SP-1-1	Area	2695	672	**47380**

Table 4.5: AES candidate algorithm finalists TPS evaluation — best feedback mode implementations

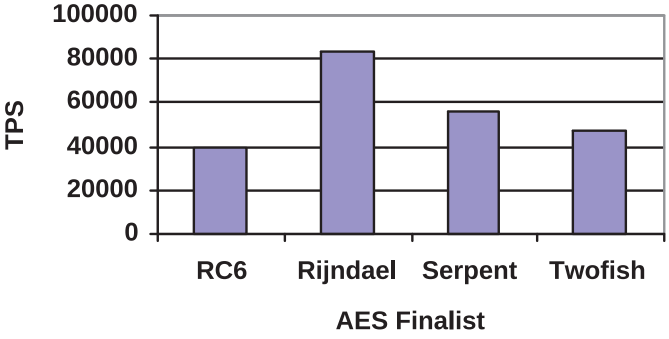


Figure 4.18: Best TPS — feedback mode

Based on the data shown in Tables 4.2 and 4.3, the Serpent algorithm clearly outperforms the other evaluated AES candidate algorithm finalists in both modes of operation in terms of throughput. As compared to its nearest competitor, Serpent exhibits a throughput increase of a factor 2.1 in non-feedback mode and a factor 1.5 in feedback mode. From the data shown in Tables 4.4 and 4.5, the Serpent algorithm outperforms the other evaluated AES candidate algorithm finalists in terms of TPS when operating in non-feedback mode. As compared to it's nearest competitor, Serpent exhibits a TPS increase of a factor of 2.2 in non-feedback mode. When operating in feedback mode, Rijndael outperforms the other evaluated AES candidate algorithm finalists in terms of TPS, outperforming its nearest competitor by a factor of 1.5. Interestingly, RC6, Rijndael, and Twofish all exhibit similar throughput and TPS results in non-feedback mode. Additionally, Rijndael exhibits significantly improved performance in respect to the other evaluated AES candidate algorithm finalists when operating in feedback mode, significantly outperforming its competitors in TPS while outperforming RC6 and Twofish in throughput. However, Rijndael is still 50% slower than Serpent when operating in feedback mode.

One of the main findings of our investigation, namely that Serpent appears to be especially well suited for an FPGA implementation, seems especially interesting considering that Serpent is clearly not the fastest algorithm with respect to most software comparisons [58]. Another major result of our study is that all four algorithms considered easily achieve Gigabit encryption rates with standard commercially available FPGAs. The algorithms are at least one order of magnitude faster than the best reported software realizations when operating in non-feedback mode. These speed-ups are essentially achieved by parallelization (pipelining and sub-pipelining) of the loop structure and by wide operand processing (e.g., processing of 128 bits in once clock cycle), both of which are not feasible on current processors. We would like to stress that the pipelined architectures cannot be used to their maximum ability for modes of operation which require feedback (CFB, OFB, etc.). However we believe that for many applications which require high encryption rates, non-feedback modes (or modified feedback modes such as interleaved CFB) will be the modes of choice. Note that the Counter Mode (in which pseudo-random input is encrypted by the block cipher, the output of which is XORed with the plaintext) grew out of the need for high speed encryption of ATM networks which required parallelization of the encryption algorithm. The reader is referred to the NIST specification for more information regarding AES modes of operation [6].

Chapter 5

Common Block Cipher Elements

5.1 Block Cipher Decomposition

As discussed in Section 2.1, block cipher core operations span a vast range of functionality. An analysis of a wide range of block ciphers was performed in order to develop a hardware architecture optimized for block cipher implementation by identifying their specific core operations. The analysis was 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. Tables 5.1 and 5.2 present the atomic operations of various block ciphers which are defined as:

- Bitwise XOR, AND, or OR.
- Addition or subtraction modulo the table entry value.
- Shift or rotation by a constant number of bits.
- Data-dependent rotation by a variable number of bits.
- Multiplication modulo the table entry value (bullet indicates constant multiplication).
- Multiplication in the Galois field specified by the table entry value.

- Inversion modulo the table entry value.

- Look-up-table substitution (asterisk indicates key dependency).

	XOR	**ADD**				**GF**			**Int.**
	AND	**SUB**	**Fixed**	**Var.**	**MUL**	**Constant**	**Inv.**		**Block**
Algorithm	**OR**	**Mod**	**Shift**	**Rot.**	**Mod**	**MUL**	**Mod**	**LUT**	**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 5.1: Block cipher core operations — 64-bit block ciphers

	XOR	**ADD**				**GF**			**Int.**
	AND	**SUB**	**Fixed**	**Var.**	**MUL**	**Constant**	**Inv.**		**Block**
Algorithm	**OR**	**Mod**	**Shift**	**Rot.**	**Mod**	**MUL**	**Mod**	**LUT**	**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 5.2: Block cipher core operations — 128-bit block ciphers

5.2 Decomposition Analysis

Tables 5.1 and 5.2 were used to generate a list of operations that the specialized reconfigurable architecture must support to guarantee efficient implementation over the range of block ciphers studied.

5.2.1 Boolean Operations

Tables 5.1 and 5.2 indicated that bit-wise Boolean operations — XOR, AND, and OR — exhibit a high frequency of occurrence among the block ciphers studied. As a result, support of bit-wise Boolean operations was determined to be an architecture requirement.

5.2.2 Modular Additions/Subtractions

Tables 5.1 and 5.2 indicated that modular additions and subtractions exhibit a high frequency of occurrence among the block ciphers studied. In particular, these operations are primarily performed $mod\ 2^{16}$ and $mod\ 2^{32}$. As a result, support of addition and subtraction operations $mod\ 2^{16}$ and $mod\ 2^{32}$ was determined to be an architecture requirement. It was also determined that the architecture would support addition and subtraction operations $mod\ 2^{8}$ as this would impose little additional resource requirements upon the architecture while adding to the coverage of the block ciphers studied.

5.2.3 Fixed Shifts

Tables 5.1 and 5.2 indicated that fixed shift operations exhibit a high frequency of occurrence among the block ciphers studied. As a result, support of fixed shift operations was determined to be an architecture requirement.

5.2.4 Variable Rotations

Tables 5.1 and 5.2 indicated that variable rotation operations exhibit a moderate frequency of occurrence among the block ciphers studied. Variable rotation operations occur when a block of data must be rotated based on the value of another block of data. Given that the architecture will have communication channels between datapath elements, support of variable rotation operations was feasible and as a result was determined to be an architecture requirement.

5.2.5 Modular Multiplications

Tables 5.1 and 5.2 indicated that modular multiplication operations exhibit a low frequency of occurrence among the block ciphers studied. In particular, these operations are primarily performed $mod\ 2^{32}$. As a result, support of modular multiplication operations $mod\ 2^{32}$ was determined to be an architecture requirement. It was also determined that the architecture would support modular multiplication operations $mod\ 2^{16}$ and modular squaring operations $mod\ 2^{32}$ as this would impose little additional resource requirements upon the architecture while adding to the coverage of the block ciphers studied. However, because modular multiplication operations require significant logic resources, it was determined that these operations would not be implemented in every block within the architecture in an attempt to reduce the total logic requirements of the architecture.

5.2.6 Galois Field Multiplications

Tables 5.1 and 5.2 indicated that Galois field multiplication operations exhibit a low frequency of occurrence among the block ciphers studied and are primarily in the field GF(2^8). However, these operations are present in a number of the AES candidate algorithm finalists and can be implemented efficiently in hardware due to their multiplications by a constant [110]. As a result, support of Galois field multiplication by a constant was determined to be an architecture requirement.

5.2.7 Modular Inversions

Tables 5.1 and 5.2 indicated that modular inversion operations were required by only one of the block ciphers studied. Inversion $mod\ 2^{32}$ is actually quite simple and requires significantly fewer hardware resources as compared to generalized inversion. However, given that modular inversion operations were required by only one of the block ciphers studied, support of modular inversion operations was eliminated from the architecture requirements.

5.2.8 Look-Up-Tables

Tables 5.1 and 5.2 indicated that look-up-table operations exhibit a high frequency of occurrence among the block ciphers studied. In particular, these operations are primarily performed using either 8-bit to 8-bit or 8-bit to 32-bit look-up-tables. Given that the architecture will have RAM elements for the storage of internal data values, the support of 8-bit to 8-bit look-up-tables was determined to be an architectural requirement with the assumption that the RAM elements will be capable of supporting the 8-bit to 32-bit look-up-tables. It was also determined that the architecture would support 4-bit to 4-bit look-up-tables as this would impose little additional resource requirements upon the architecture while adding to the coverage of the block ciphers studied. To provide better utilization of the memory elements used to implement the look-up-tables, it was determined that a paging mode would be used when an element was configured to use the 4-bit to 4-bit look-up-tables as opposed to the 8-bit to 8-bit look-up-tables.

5.2.9 Internal Block Size

It is important to note that tables 5.1 and 5.2 also present the internal block size for each block cipher. The internal block size parameter is significant as an indication of the internal datapath width that a specialized reconfigurable architecture must support to result in efficient block cipher implementations. From the tables it is evident that the predominant internal block size is thirty two bits, although this value increases when examining block ciphers with 128-bit block lengths.

5.3 Architecture Requirements

As a result of the previous analysis, a list of architecture requirements was developed for use in designing the specialized reconfigurable architecture so as to result in efficient implementation of the block ciphers studied.

- Support of the operations:
  - Bitwise XOR, AND, or OR.
  - Addition or subtraction modulo 2^8, 2^{16}, and 2^{32}.
  - Shift or rotation by a constant number of bits.
  - Data-dependent rotation by a variable number of bits.
  - Multiplication modulo 2^{16} and 2^{32} and squaring modulo 2^{32}.
  - Fixed field constant multiplication in the Galois field GF(2^8).
  - Look-up-table substitution of the forms:
    - 4-bit to 4-bit with paging mode.
    - 8-bit to 8-bit.
    - 8-bit to 32-bit via RAM elements.
- Support of a 32-bit datapath replicated a minimum of four times to support 128-bit block lengths.
- Support of communication between datapaths.

Chapter 6

The COBRA Architecture

A specialized reconfigurable architecture achieves greater flexibility as compared to custom ASIC or specialized processor solutions as well as faster reconfiguration time when compared to a commercial FPGA solution. Additionally, overhead and off-chip communication are minimized when comparing a specialized reconfigurable architecture to a general purpose processor architecture. As demonstrated in Chapter 2, reconfigurable processor architectures offer a great deal of insight towards creating an architecture optimized for block cipher implementation. It is imperative that the reconfigurable datapath be tightly coupled with the processor core so that the overhead associated with their interface does not become the system bottleneck [68, 120, 134, 153]. Ensuring full support of algorithm-specific operations requires maximizing the functional density of the datapath. This results in the need for a datapath that is both generalized and run-time reconfigurable, a requirement that is fulfilled via a processor operating via very long instruction word (VLIW) format that controls the reconfigurable datapath [9, 31, 51, 134, 151, 152, 161, 163]. Moreover, this microcoded approach creates an instruction set that is easily extended and enhanced [62]. Because block ciphers are dataflow oriented, a reconfigurable element with coarse granularity configured via the microcode offers the best solution for achieving maximum system performance, especially when implementing operations such as integer multiplication, fixed field constant multiplication in a Galois field, table look-up, or bit-wise shift and rotation, as these operations do not map well to more traditional, fine-grained reconfigurable architectures [30, 31, 69, 81, 123, 134, 154].

When designing a specialized reconfigurable architecture for efficient block cipher implementation, a number of architecture requirements were established to facilitate achieving an optimized solution. In addition to the functionality requirements specified in Chapter 5, the requirements of the proposed Cryptographic (Optimized for Block Ciphers) Reconfigurable Architecture (COBRA) included:

- Coverage of a wide variety of algorithms.
  - Concentrate on 64-bit and 128-bit algorithms.
  - Ignore algorithms proven to be insecure.
- A generalized datapath suited for most block ciphers.
- A generalized instruction set to control the reconfigurable elements.

6.1 System I/O

The I/O of the COBRA architecture are summarized in Table 6.1. All signals are active high unless specified otherwise.

Signal	**Direction**	**Width in Bits**	**Function**
in_1	In	32	32-bit input to datapath column 0
in_2	In	32	32-bit input to datapath column 1
in_3	In	32	32-bit input to datapath column 2
in_4	In	32	32-bit input to datapath column 3
fclk	In	1	Instruction RAM clock
clk	In	1	Datapath clock
rst	In	1	System reset (active low)
clr	In	1	Configuration clear (active low)
go	In	1	Begin encryption/decryption
load_ctr	In	12	Address to the Instruction RAM when loading a program into COBRA
load_din	In	80	Data to the Instruction RAM when loading a program into COBRA
load_stb	In	1	Strobe to load an instruction into the Instruction RAM
done_load	In	1	Indicator from an external peripheral that the Instruction RAM has been fully programmed
ready	Out	1	System is ready to perform an encryption or decryption
data_valid	Out	1	Data on the output pins is valid
flag_0	Out	1	User-defined flag
flag_1	Out	1	User-defined flag
idle	Out	1	Encryption or decryption in progress
flag_3	Out	1	User-defined flag
flag_4	Out	1	User-defined flag
flag_5	Out	1	User-defined flag
out_1	Out	32	32-bit output from datapath column 0
out_2	Out	32	32-bit output from datapath column 1
out_3	Out	32	32-bit output from datapath column 2
out_4	Out	32	32-bit output from datapath column 3

Table 6.1: COBRA I/O

6.2 Block Diagram and Interconnect

Figure 6.1 details the architecture and interconnect of COBRA. The basic building block for the COBRA architecture is the Reconfigurable Cryptographic Elements (RCEs). All RCEs in columns 1 and 3 have an additional built-in functional unit allowing for the performance of modular multiplication and squaring — these elements are denoted as RCE MULs. Each RCE operates upon a 32-bit data stream within a 128-bit block. Two byte-wise crossbar switches are embedded between rows 0 and 1 and rows 2 and 3 to allow for byte-wise permutations. Data flows from top to bottom, passing through a total of four rows of RCEs and RCE MULs and both crossbar switches. Whitening registers are connected to the outputs of row 3 and the resultant output is both the COBRA output bus and an input to the feedback multiplexor, allowing for iterative operation upon the 128-bit data stream. Whitening registers may be configured to perform either bit-wise XOR or $mod2^{32}$ addition to support the post encryption/decryption key whitening stages required by many block ciphers. Sixteen embedded RAMs (eRAMs) are provided for use as temporary storage space for intermediate values as well as round keys to be used during encryption or decryption. These eRAMs are partitioned such that half are allocated to columns 0 and 1 and the other half are allocated to columns 2 and 3.

Note that the COBRA interconnect need not match the functionality of a generic interconnection matrix typically implemented in commercial FPGAs due to the top to bottom flow of data. Implementing a fixed interconnection network significantly decreases the layout complexity of the COBRA architecture. Also note that each RCE or RCE MUL receives the 32-bit output of either the RCEs and RCE MULs or crossbar switch directly above it. The 32-bit output from the RCE or RCE MUL directly above a given RCE is considered the primary input to that RCE or RCE MUL. In the case of a crossbar being above the RCE or RCE MUL, the crossbar output is partitioned into four 32-bit blocks from left to right with the crossbar output block directly above an RCE or RCE MUL being considered its primary input. The remaining 32-bit blocks are grouped from left to right and are considered secondary inputs to the given RCE or RCE MUL. The left-most block is termed *INB*, the next middle block is termed *INC*, and the right-most block is termed *IND*. The eRAM input to the RCE or RCE MUL is termed *INER*. These terms are used in the custom assembly language created to support the COBRA architecture (see Chapter 7).

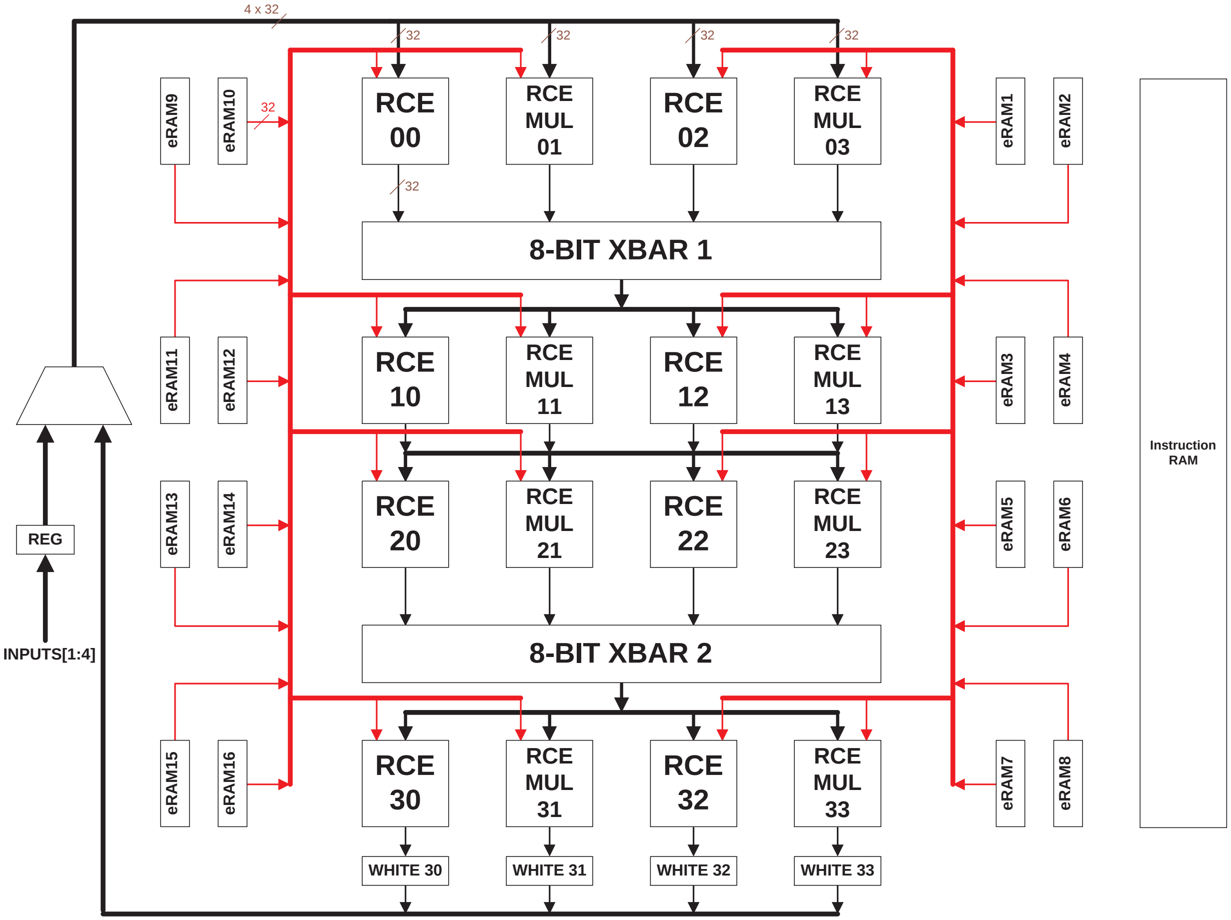


Figure 6.1: COBRA architecture and interconnect

6.3 Reconfigurable Cryptographic Elements

Figures 6.2 and 6.3 detail the functionality of the RCE and RCE MUL. The elements that comprise an RCE and an RCE MUL are:

1. A: Bit-wise XOR, AND, or OR.

2. B: Add or subtract mod $2^8, 2^{16}$, or 2^{32}.

3. C: Look-Up-Tables (LUTs) — either four 8-bit to 8-bit LUTs or eight pages of eight 4-bit to 4-bit LUTs.

4. D: Multiply mod 2^{16} or 2^{32} or Square mod 2^{32}.

5. E: Shift left, shift right, or rotate left. Shift and rotate values may be data dependent.

6. F: GF(2^8) Galois field constant multiplier.

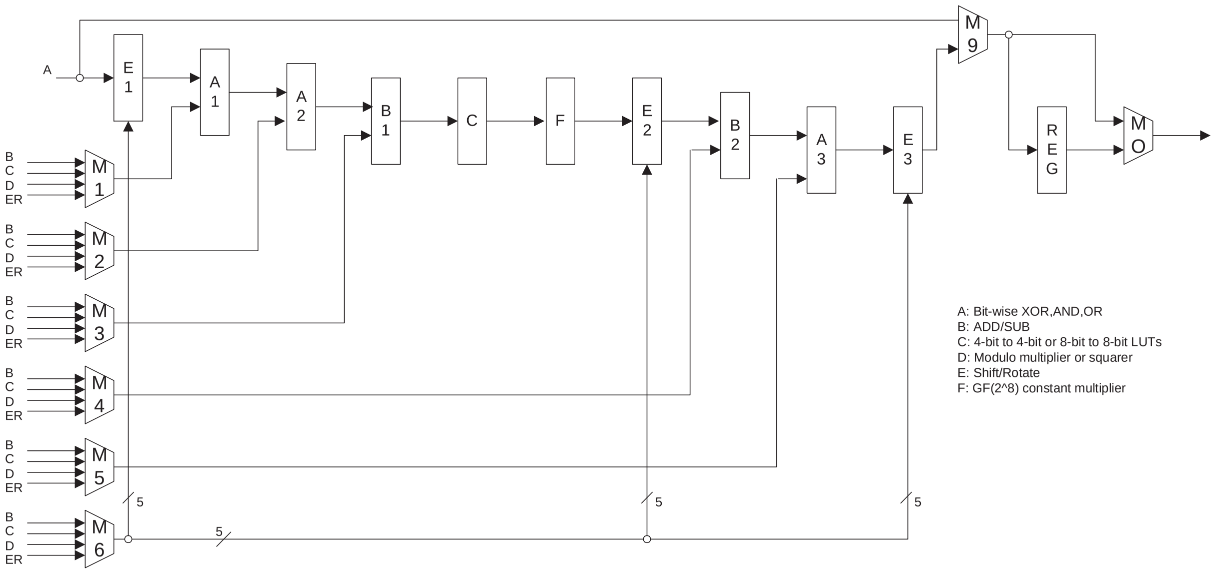
7. M: Multiplexor.

8. REG : Register.

The RCE structures are balanced to provide the greatest efficiency of implementation. E elements are placed at the front, rear, and middle of the structures to allow for flexibility in manipulation of the data. Moreover, the block cipher study indicated that many functions require shifting or rotation at the beginning, middle, and end, necessitating the placement of the E elements. A elements are placed at the front and rear of the structures to allow for Boolean operations at either the beginning or end of a function. Note that two A elements are placed at the front and only one at the rear of the structures as the block cipher study indicated that a greater amount of Boolean operations occur at the beginning of a function. The block cipher study also indicated that Boolean operations tend to occur immediately after shifting or rotation operations as well as at either the beginning or end of the function, necessitating their placement between the E elements. B elements are balanced at the front and rear of the structures and are placed between the A elements. In the case of the RCE, one C and one F element are placed in the middle of the structure, in particular before the central E element. The block cipher study indicated that shifts or rotations that occur in the middle of a function tend to do so after the need for LUTs, Galois field constant multiplication, or standard multiplication. In the case of the RCE MUL, the C elements are balanced within the B elements and one F element and one D element. Also, a bypass multiplexor *M7* is included in the RCE MUL as the block cipher study determined a need to support functions of the form:

$$A = (\mathrm{B} \times \mathrm{C}) + \mathrm{B}$$

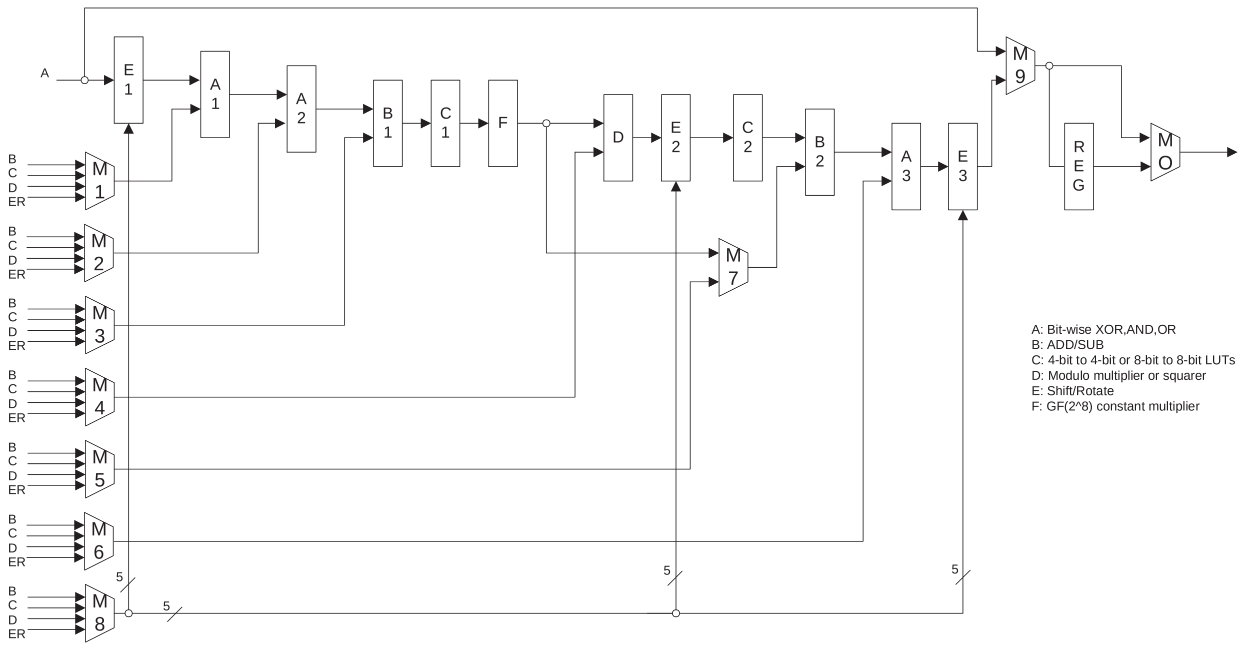
The M elements allow for the selection of any of the secondary bus inputs to the RCE structure (including the eRAM input). Note that the M element connected to the E elements provides the shared data dependent rotation/shift value which overrides a programmed value when data dependent operation of the E elements is enabled. A bypass path through *M9* is provided such that an RCE structure may be disabled and placed in a power-saving mode. Finally, an output multiplexor *MO* is used to select either the registered or unregistered result of the RCE structure.



A: Bit-wise XOR,AND,OR
B: ADD/SUB
C: 4-bit to 4-bit or 8-bit to 8-bit LUTs
D: Modulo multiplier or squarer
E: Shift/Rotate
F: GF(2^8) constant multiplier

All connections are 32-bit bus lines unless otherwise noted.

Figure 6.2: COBRA reconfigurable cryptographic element



All connections are 32-bit bus lines unless otherwise noted.

Figure 6.3: COBRA reconfigurable cryptographic element with multiplier

6.3.1 RCE/RCE MUL Elements: A

The *A* element accepts two 32-bit inputs, *input 1* and *input 2*, and provides one 32-bit output. The *A* element operates as detailed in Table 6.2 based on the *configuration code* specified by the programmer for the RCE or RCE MUL in question.

Configuration Code	**Operation**
00	Bit-wise XOR
01	Bit-wise AND
10	Bit-wise OR
11	Pass through of input 1

Table 6.2: RCE/RCE MUL element functionality — A

6.3.2 RCE/RCE MUL Elements: B

The *B* element accepts two 32-bit inputs, *input 1* and *input 2*, and provides a 32-bit output. The *B* element operates as detailed in Table 6.3 based on the *configuration code* specified by the programmer for the RCE or RCE MUL in question. Note that when multiple additions or subtractions are performed, the operands are divided into either bytes or words and then operated upon in an aligned fashion. As an example, in the case of *configuration code 001*, *input 1* and *input 2* are divided into high order and low order words. The high order words of each operand are then added *mod* 2^{16}, forming the high order word of the resultant output. Similarly, the low order words of each operand are added *mod* 2^{16}, forming the low order word of the resultant output.

6.3.3 RCE/RCE MUL Elements: C

The *C* element accepts one 32-bit input and provides one 32-bit output. The *C* element operates as detailed in Tables 6.4, 6.5, and 6.6 based on the *configuration code* specified by

Configuration Code	**Operation**
000	1 Mod 2^{32} addition
001	2 Mod 2^{16} additions
010	4 Mod 2^{8} additions
011	1 Mod 2^{32} subtraction
100	2 Mod 2^{16} subtraction
101	4 Mod 2^{8} subtraction
110	Pass through of input 1
111	Pass through of input 2

Table 6.3: RCE/RCE MUL element functionality — B

the programmer for the RCE or RCE MUL in question. Note that when the entire 32-bit input is used as the input to the LUTs, the input is divided into either four 8-bit blocks or eight 4-bit blocks depending on the mode of operation. The resultant output blocks from the LUTs retain their ordering when recombined to form the final 32-bit output.

Configuration Code Bits [3:2]	**Operation**
00	Use the four LSBs of the input as the input to all of the 4-input LUTs
01	Use the eight LSBs of the input as the input to all of the 8-input LUTs
10	Use the entire input as the input to the LUTs when operating in either 4-LUT or 8-LUT mode
11	Use the entire input as the input to the LUTs when operating in either 4-LUT or 8-LUT mode

Table 6.4: RCE/RCE MUL element functionality — C configuration code bit [3:2]

Configuration Code Bit 1	**Operation**
0	Use eight 4-bit LUTs (4-LUT mode)
1	Use four 8-bit LUTs (8-LUT mode)

Table 6.5: RCE/RCE MUL element functionality — C configuration code bit 1

Configuration Code Bit 0	**Operation**
0	Pass through
1	Enable LUTs

Table 6.6: RCE/RCE MUL element functionality — C configuration code bit 0

6.3.4 RCE/RCE MUL Elements: D

The *D* element accepts two 32-bit inputs, *input 1* and *input 2*, and provides a 32-bit output. The *D* element operates as detailed in Table 6.7 based on the *configuration code* specified by the programmer for the RCE MUL in question. The *D* element is only available in the RCE MULs. Note that when multiple multiplications are performed, the operands are divided into either bytes or words and then operated upon in an aligned fashion. As an example, in the case of *configuration code 01*, *input 1* and *input 2* are divided into high order and low order words. The high order words of each operand are then multiplied *mod* 2^{16}, forming the high order word of the resultant output. Similarly, the low order words of each operand are multiplied *mod* 2^{16}, forming the low order word of the resultant output.

Configuration Code	**Operation**
00	1 Mod 2^{32} multiplication
01	2 Mod 2^{16} multiplications
10	1 Mod 2^{32} squaring of input 1
11	Pass through of input 1

Table 6.7: RCE/RCE MUL element functionality — D

6.3.5 RCE/RCE MUL Elements: E

The *E* element accepts one 32-bit input and provides one 32-bit output. The *E* element operates as detailed in Tables 6.8 and 6.9 based on the *configuration code* specified by the programmer for the RCE or RCE MUL in question.

Configuration Code Bit 7	**Operation**
0	Normal shift/rotate Use configuration code bits [4:0] for shift/rotate amount
1	Data dependent shift/rotate Use dedicated multiplexor output (M6 for RCE, M8 for RCE MUL) for shift/rotate amount

Table 6.8: RCE/RCE MUL element functionality — E configuration code bit 7

Configuration Code Bits [6:5]	**Operation**
00	Rotate left
01	Shift left inserting zeroes
10	Shift right inserting zeroes
11	Pass through

Table 6.9: RCE/RCE MUL element functionality — E configuration code bits [6:5]

6.3.6 RCE/RCE MUL Elements: F

The *F* element accepts one 32-bit input and provides one 32-bit output. The *F* element operates as detailed in Table 6.10 based on the *configuration code* specified by the programmer for the RCE or RCE MUL in question.

Configuration Code	**Operation**
0	Pass through
1	Fixed field GF(2^8) Galois field constant multiplication

Table 6.10: RCE/RCE MUL element functionality — F

6.3.7 RCE/RCE MUL Elements: M

The *M8* and *M[6:1]* elements accept four 32-bit inputs, *B*, *C*, *D*, and *ER*, and provide either a 32-bit or a 5-bit output. The 5-bit output occurs in the cases of *M6* in an RCE and *M8* in an RCE MUL — all other *M* elements provide 32-bit outputs. The *M8* and *M[6:1]* elements operate as detailed in Table 6.11 based on the *configuration code* specified by the programmer for the RCE or RCE MUL in question. The *M7*, *M9*, and *MO* elements accept two 32-bit inputs and provide a 32-bit output as detailed in Tables 6.12, 6.13, and 6.14 based on the *configuration code* specified by the programmer for the RCE or RCE MUL in question.

Configuration Code	**Operation**
00	Input B
01	Input C
10	Input D
11	Input ER

Table 6.11: RCE/RCE MUL element functionality — M8 and M[6:1]

Configuration Code	**Operation**
0	M5 output
1	F output

Table 6.12: RCE MUL element functionality — M7

Configuration Code	Operation
0	E3 output
1	Input A

Table 6.13: RCE/RCE MUL element functionality — M9

Configuration Code	Operation
0	Registered output
1	M9 output

Table 6.14: RCE/RCE MUL element functionality — MO

6.3.8 RCE/RCE MUL Elements: REG

The *REG* element accepts one 32-bit input from element *E3* and provides a registered version of the input as its 32-bit output. As will be described in Section 6.7, the *REG* element may be enabled or disabled when necessary. The *REG* element is clocked using the datapath clock.

6.4 Instruction RAM

COBRA operates via a Very Long Instruction Word (VLIW) format in which a fixed-length instruction is fetched and executed in two instruction RAM clock cycles. Each instruction is 80 bits and the instruction bus is connected to the control mechanisms for the RCEs, RCE MULs, crossbar switches, eRAMs, whitening registers, and the feedback multiplexor. The instruction bus is driven via the instruction RAM (iRAM) which operates independent from the datapath. This configuration allows the iRAM to reconfigure the datapath during operation. The iRAM is a 12-bit × 80-bit memory which supports programs of up to 4096 total instructions. Instructions are classified as follows:

1. *Configure RCE or RCE MUL*:
    (a) Program LUT data values.
    (b) Program Galois field constant multiplier data values.
    (c) Configure RCE or RCE MUL elements.
    (d) Configure LUT bank address when operating in 4-bit to 4-bit mode.
2. *Enable/disable RCE or RCE MUL outputs.*
3. *Configure crossbar.*
4. *Configure input multiplexor.*
5. *Configure eRAM*:
    (a) Set eRAM address.
    (b) Set eRAM data input source.
    (c) Set RCE or RCE MUL eRAM input source.
    (d) Strobe eRAMs to load data.
6. *Control whitening elements*:
    (a) Configure whitening elements.
    (b) Strobe whitening elements to load data.
7. *Read/write flag register.*
8. *Jump to specified address.*
9. *No operation (NOP).*

COBRA elements are configured by VLIW instructions that write control words to the associated control registers. This configuration format allows for on-the-fly reconfiguration. As an example, to reconfigure the elements within an RCE, an instruction must be executed that contains the control data for each element within the *configuration data* field of the instruction as detailed in Figure 6.4.

6.5 Control Logic

The COBRA architecture control logic performs a variety of functions to guarantee proper system operation. On power-up, the architecture idles until the iRAM and datapath clocks have been synchronized and the external system indicates that the iRAM has been loaded. Once these conditions are met, the loading and executing of instructions within the iRAM is automatically initiated. Generic flags may be used in the COBRA microcode written by the programmer to indicate to the external system when to provide data required for key scheduling or other tasks.

Upon completion of key scheduling, the *ready* flag must be raised by the microcode, indicating to the external system that COBRA is ready to begin either encryption or decryption. COBRA halts upon detection of the *ready* flag and waits for the external system to initiate the encryption or decryption process by raising the *go* signal. Upon detection of the *go* signal, the *idle* flag must be raised by the microcode, indicating to the external system that an encryption or a decryption is in progress. Upon completion of an encryption, the *data_valid* flag must be raised by the microcode, indicating to the external system that the COBRA output data is valid. The microcode must reconfigure COBRA as necessary for the start of a new encryption or decryption operation and then jump to the idle point at the start of the process. If COBRA detects that the *go* signal is still active, a new encryption or decryption operation will commence. If the *go* signal is inactive, COBRA will idle until the *go* signal is raised by the external system.

6.6 Configuration

Through the use of a given instruction, an element within the COBRA datapath may be reconfigured on-the-fly based on the operations required by a block cipher as programmed and stored within the iRAM. Reconfiguration is achieved via a dual clocking scheme. The maximum operating frequency of the datapath may be calculated based upon how it is to be programmed to implement each function required by the given block cipher — key scheduling, encryption, or decryption. The implementation of each of these functions results in a different critical delay path depending on the configuration of the COBRA datapath. The datapath clock is then set to the maximum operating frequency based on a worst case delay analysis performed across these functions. The iRAM is then clocked via a separate clock that is set to a user-specified multiple of the datapath clock. The multiple is determined based upon the number of instructions that must be executed within a given datapath clock period to properly reconfigure the datapath for a given operation and the fact that two iRAM clock cycles are required to load and execute an instruction.

The programmer must determine the optimal number of instructions (not necessarily the maximum number) that must be executed within a datapath clock cycle. When determining the optimal frequency of the iRAM clock, both *overfull* and *underfull* instruction cycles must be considered. An *overfull* instruction cycle occurs when the total number of instructions that must be executed for on-the-fly reconfiguration to be completed is greater than the number of instructions that can be executed in a datapath cycle. An *overfull* instruction cycle is completed by disabling the RCE outputs before the end of the datapath cycle and enabling the outputs when reconfiguration is completed. An *underfull* instruction cycle occurs when the total number of instructions that must be executed for on-the fly reconfiguration to be completed is less than the total number of instructions that can be executed in a datapath cycle, resulting in the insertion of NOP instructions.

As the iRAM clock frequency increases, the number of instructions that may be executed per datapath cycle increases. Ideally, the iRAM clock frequency is set to allow for enough instructions to be executed per datapath cycle for the worst case on-the-fly reconfiguration scenario of a given program, resulting in a complete elimination of *overfull* instruction cycles. However, this frequency is often not feasible for implementation purposes. Increasing the number of instructions that may be executed per datapath cycle increases the number of *underfull* instruction cycles. This in turn grows the size of the overall program such that it may exceed the iRAM instruction space. Moreover, power consumption and heat dissipation both increase as the iRAM clock frequency increases which can result in system-level design difficulties.

As the iRAM clock frequency is decreased, the number of *underfull* instruction cycles decreases, decreasing the overall size of the program. However, the decrease in *underfull* instruction cycles comes at the cost of an increase in the number of *overfull* instruction cycles. As the iRAM clock frequency continues to decrease, the number of *overfull* instruction cycles grows to the point of actually increasing the program size and decreasing the overall throughput of the program. This increase is caused by the instructions required to disable and enable the RCE outputs as well as the extra *underfull* instruction cycles required to fill the empty space created by the completion of an *overfull* instruction cycle before the end of the datapath cycle.

To achieve optimal performance, the programmer must carefully analyze the program when choosing the iRAM clock frequency. The iRAM clock frequency must be chosen to achieve a balance whereby the number of *underfull* instruction cycles is minimized but not past the point where program throughput is decreased as a result of the increase in *overfull* instruction cycles. This results in a minimized instruction space and a maximized program throughput.

6.7 Instruction Mapping

Figure 6.4 details the COBRA instruction mapping. The interpretation of the *configuration data* is based on the values of the *operation code*, *slice address*, *element address*, and *LUT address* fields. Table 6.15 details the *operation code* types and the used portion of the *slice address* bit range. Table 6.16 details the *element address* types and the used portion of the *LUT address* bit range. Table 6.17 details the size and functionality of the elements within the *configuration data* field for each operation.

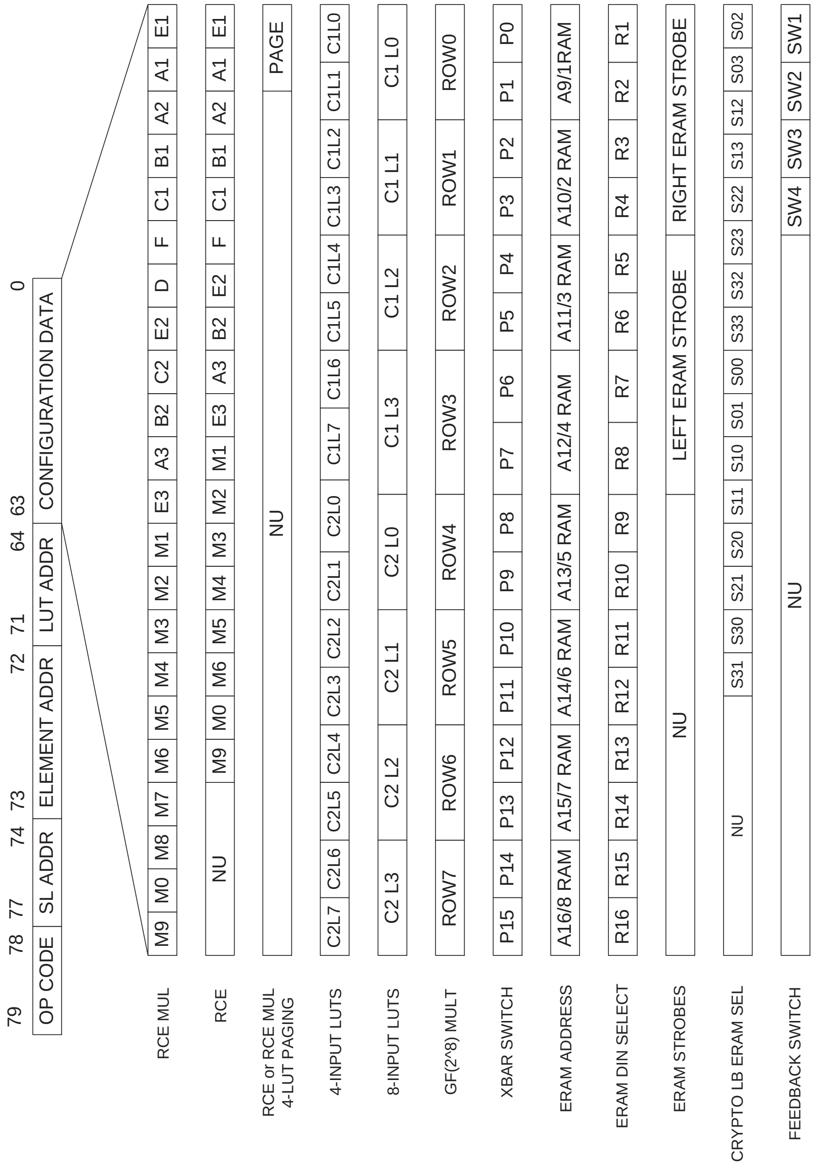

79
78
77
74
73
72
71
64
63
0
OP CODE
SL ADDR
ELEMENT ADDR
LUT ADDR
CONFIGURATION DATA
RCE MUL
M9 M0 M8 M7 M6 M5 M4 M3 M2 M1 E3 A3 B2 C2 E2 D F C1 B1 A2 A1 E1
RCE
NU M9 M0 M6 M5 M4 M3 M2 M1 E3 A3 B2 E2 F C1 B1 A2 A1 E1
RCE or RCE MUL
4-LUT PAGING
NU PAGE
4-INPUT LUTS
C2L7 C2L6 C2L5 C2L4 C2L3 C2L2 C2L1 C2L0 C1L7 C1L6 C1L5 C1L4 C1L3 C1L2 C1L1 C1L0
8-INPUT LUTS
C2 L3 C2 L2 C2 L1 C2 L0 C1 L3 C1 L2 C1 L1 C1 L0
GF(2^8) MULT
ROW7 ROW6 ROW5 ROW4 ROW3 ROW2 ROW1 ROW0
XBAR SWITCH
P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
ERAM ADDRESS
A16/8 RAM A15/7 RAM A14/6 RAM A13/5 RAM A12/4 RAM A11/3 RAM A10/2 RAM A9/1RAM
ERAM DIN SELECT
R16 R15 R14 R13 R12 R11 R10 R9 R8 R7 R6 R5 R4 R3 R2 R1
ERAM STROBES
NU LEFT ERAM STROBE RIGHT ERAM STROBE
CRYPTO LB ERAM SEL
NU S31 S30 S21 S20 S11 S10 S01 S00 S33 S32 S23 S22 S13 S12 S03 S02
FEEDBACK SWITCH
NU SW4 SW3 SW2 SW1

Figure 6.4: COBRA instruction mapping

Operation Code	Operation Type	Slice Address Range
00	Program RCE or RCE MUL	0000 to 1111
01	Program Crossbar or Feedback Switch	0000 to 0010
10	Program eRAM Control	0000 to 0100
11	Program Flags or Whitening Registers or Perform Jump or NOP	1000 to 1111

Table 6.15: COBRA operation code decoding

When the *operation code* is set to *00*, one of the sixteen RCEs may be addressed for configuration based on the *slice address*, which is broken into two 2-bit blocks. In this case, the upper two bits of the *slice address* represent the row address and the lower two bits represent the column address of the RCE to be addressed where RCE *00* is located in the upper left corner of Figure 6.1.

Element Address	Operation Type	LUT Address Range
00	Program Look-Up-Tables	00000000 to 11111111
01	Program Standard Elements	Not Used
10	Program 1 of 16 8 × 8 Galois Field Constant Multipliers	00000000 to 00001111
11	Set 4-bit Look-Up-Table Paging Address	Not Used

Table 6.16: COBRA element address decoding

When the *element address* is set to *00*, the *C* elements within an RCE are to be configured. When operating in 8-bit LUT mode, the entire *LUT address* is used as the input address to each of the 8-bit LUTs within the *C* element. In the case of an RCE MUL, this address is used by both the *C1* and *C2* elements. When operating in 4-bit LUT mode, the four LSBs of the *LUT address* are used as the input address to each of the 4-bit LUTs within the *C* element. Additionally, the three MSBs of the *LUT address* are used to indicate the bank address when operating in 4-bit LUT mode. As in 8-bit LUT mode, the 4-bit address is used by both the *C1* and *C2* elements of an RCE MUL when operating in 4-bit LUT mode.

Element	**Size in Bits**	**Functionality**
A[3:1]	2	Bit-Wise Boolean
B[2:1]	3	Modular Addition/Subtraction
C[2:1]	4	LUT
D	2	Modular Multiplication/Squaring
E[3:1]	8	Fixed or Variable Shift/Rotation
F	1	Fixed Field GF(2^8) Galois Field Constant Multiplication
MO	1	Output Multiplexing
M[6:1]	2	Internal Bus Multiplexing
M7	1	Modular Multiplication Bypass Multiplexing
M8	2	Internal Bus Multiplexing
M9	1	RCE or RCE MUL Internal Bypass Multiplexing
PAGE	3	4-LUT Page Selection
C[2:1] L[7:0]	4	Data for Storage in 4-LUTs
C[2:1] L[3:0]	8	Data for Storage in 8-LUTs
ROW[7:0]	8	Data for Storage in 8 × 8 Fixed Field GF(2^8) Galois Field Constant Multiplier
P[15:0]	4	Input Byte Location for Byte-Wise Crossbar Switch
A[16:9/8:1] RAM	8	eRAM Address
R[16:1]	4	Source of eRAM Input
LEFT ERAM STROBES	8	eRAM 09-16 Write Strobes
RIGHT ERAM STROBES	8	eRAM 01-08 Write Strobes
S[3:0,3:0]	3	Source of RCE eRAM Input
SW[3:0]	1	Feedback Switch
NU	Variable	Not Used

Table 6.17: COBRA configuration data elements — size and functionality

6.8 Instruction Set

A COBRA instruction is assembled from the *operation code*, *slice address*, and *element address*. Tables 6.18, 6.19, and 6.20 detail the COBRA instruction set.

Instruction	Operation
00h	Store configuration data into RCE 00 LUT C1
01h	Configure RCE 00 elements
02h	Store configuration data into RCE 00 Galois Field Constant Multiplier F
03h	Configure RCE 00 LUT C1 bank address when in 4-LUT mode
04h	Store configuration data into RCE MUL 01 LUTs C1 and C2
05h	Configure RCE MUL 01 elements
06h	Store configuration data into RCE MUL 01 Galois Field Constant Multiplier F
07h	Configure RCE MUL 01 LUTs C1 and C2 bank address when in 4-LUT mode
08h	Store configuration data into RCE 02 LUT C1
09h	Configure RCE 02 elements
0Ah	Store configuration data into RCE 02 Galois Field Constant Multiplier F
0Bh	Configure RCE 02 LUT C1 bank address when in 4-LUT mode
0Ch	Store configuration data into RCE MUL 03 LUTs C1 and C2
0Dh	Configure RCE MUL 03 elements
0Eh	Store configuration data into RCE MUL 03 Galois Field Constant Multiplier F
0Fh	Configure RCE MUL 03 LUTs C1 and C2 bank address when in 4-LUT mode
10h	Store configuration data into RCE 10 LUT C1
11h	Configure RCE 10 elements
12h	Store configuration data into RCE 10 Galois Field Constant Multiplier F
13h	Configure RCE 10 LUT C1 bank address when in 4-LUT mode
14h	Store configuration data into RCE MUL 11 LUTs C1 and C2
15h	Configure RCE MUL 11 elements
16h	Store configuration data into RCE MUL 11 Galois Field Constant Multiplier F
17h	Configure RCE MUL 11 LUTs C1 and C2 bank address when in 4-LUT mode
18h	Store configuration data into RCE 12 LUT C1
19h	Configure RCE 12 elements
1Ah	Store configuration data into RCE 12 Galois Field Constant Multiplier F
1Bh	Configure RCE 12 LUT C1 bank address when in 4-LUT mode
1Ch	Store configuration data into RCE MUL 13 LUTs C1 and C2
1Dh	Configure RCE MUL 13 elements
1Eh	Store configuration data into RCE MUL 13 Galois Field Constant Multiplier F
1Fh	Configure RCE MUL 13 LUTs C1 and C2 bank address when in 4-LUT mode

Table 6.18: COBRA instruction set — instructions 00h–1Fh

Instruction	Operation
20h	Store configuration data into RCE 20 LUT C1
21h	Configure RCE 20 elements
22h	Store configuration data into RCE 20 Galois Field Constant Multiplier F
23h	Configure RCE 20 LUT C1 bank address when in 4-LUT mode
24h	Store configuration data into RCE MUL 21 LUTs C1 and C2
25h	Configure RCE MUL 21 elements
26h	Store configuration data into RCE MUL 21 Galois Field Constant Multiplier F
27h	Configure RCE MUL 21 LUTs C1 and C2 bank address when in 4-LUT mode
28h	Store configuration data into RCE 22 LUT C1
29h	Configure RCE 22 elements
2Ah	Store configuration data into RCE 22 Galois Field Constant Multiplier F
2Bh	Configure RCE 22 LUT C1 bank address when in 4-LUT mode
2Ch	Store configuration data into RCE MUL 23 LUTs C1 and C2
2Dh	Configure RCE MUL 23 elements
2Eh	Store configuration data into RCE MUL 23 Galois Field Constant Multiplier F
2Fh	Configure RCE MUL 23 LUTs C1 and C2 bank address when in 4-LUT mode
30h	Store configuration data into RCE 30 LUT C1
31h	Configure RCE 30 elements
32h	Store configuration data into RCE 30 Galois Field Constant Multiplier F
33h	Configure RCE 30 LUT C1 bank address when in 4-LUT mode
34h	Store configuration data into RCE MUL 31 LUTs C1 and C2
35h	Configure RCE MUL 31 elements
36h	Store configuration data into RCE MUL 31 Galois Field Constant Multiplier F
37h	Configure RCE MUL 31 LUTs C1 and C2 bank address when in 4-LUT mode
38h	Store configuration data into RCE 32 LUT C1
39h	Configure RCE 32 elements
3Ah	Store configuration data into RCE 32 Galois Field Constant Multiplier F
3Bh	Configure RCE 32 LUT C1 bank address when in 4-LUT mode
3Ch	Store configuration data into RCE MUL 33 LUTs C1 and C2
3Dh	Configure RCE MUL 33 elements
3Eh	Store configuration data into RCE MUL 33 Galois Field Constant Multiplier F
3Fh	Configure RCE MUL 33 LUTs C1 and C2 bank address when in 4-LUT mode

Table 6.19: COBRA instruction set — instructions 20h–3Fh

Instruction	**Operation**
40h	Configure Crossbar Switch 1
44h	Configure Crossbar Switch 2
48h	Configure Input Switch
80h	Store configuration data into Right eRAM (eRAM 01 — eRAM 08) address registers
84h	Store configuration data into Left eRAM (eRAM 09 — eRAM 16) address registers
88h	Set eRAM input source
8Ch	Set RCE and RCE MUL eRAM input source
90h	Set/Clear eRAM write strobes
C0h	No operation
E0h	Configure RCE, RCE MUL, and whitening block output enables
E4h	Configure whitening blocks
E8h	Store RCE 30 output in whitening block 30
ECh	Store RCE MUL 31 output in whitening block 31
F0h	Store RCE 32 output in whitening block 32
F4h	Store RCE MUL 33 output in whitening block 33
F8h	Store configuration data into flag register
FCh	Jump to address specified in configuration data

Table 6.20: COBRA instruction set — instructions 40h–FFh

6.8.1 Configuring RCE Elements

The format of the *configuration data* used to configure the basic elements within an RCE is detailed in table 6.21. This set of instructions does not use the *LUT address* field.

Configuration Data Bit(s)	**Element Bit(s)**
[63:55]	Not Used
54	M9
53	MO
[52:51]	M6 [1:0]
[50:49]	M5 [1:0]
[48:47]	M4 [1:0]
[46:45]	M3 [1:0]
[44:43]	M2 [1:0]
[42:41]	M1 [1:0]
[40:33]	E3 [7:0]
[32:31]	A3 [1:0]
[30:28]	B2 [2:0]
[27:20]	E2 [7:0]
19	F
[18:15]	C1 [3:0]
[14:12]	B1 [2:0]
[11:10]	A2 [1:0]
[9:8]	A1 [1:0]
[7:0]	E1 [7:0]

Table 6.21: RCE elements configuration data format

6.8.2 Configuring RCE MUL Elements

The format of the *configuration data* used to configure the basic elements within an RCE MUL is detailed in Table 6.22. This set of instructions does not use the *LUT address field*.

Configuration Data Bit(s)	**Element Bit(s)**
63	M9
62	MO
[61:60]	M8 [1:0]
59	M7
[58:57]	M6 [1:0]
[56:55]	M5 [1:0]
[54:53]	M4 [1:0]
[52:51]	M3 [1:0]
[50:49]	M2 [1:0]
[48:47]	M1 [1:0]
[46:39]	E3 [7:0]
[38:37]	A3 [1:0]
[36:34]	B2 [2:0]
[33:30]	C2 [3:0]
[29:22]	E2 [7:0]
[21:20]	D [1:0]
19	F
[18:15]	C1 [3:0]
[14:12]	B1 [2:0]
[11:10]	A2 [1:0]
[9:8]	A1 [1:0]
[7:0]	E1 [7:0]

Table 6.22: RCE MUL elements configuration data format

6.8.3 Storing Configuration Data in LUTs C1 and C2

When operating in 4-LUT mode, the four LSBs of the *LUT address* are used to address each of the eight 4-bit LUTs while the three MSBs are used to select the bank address within C1 and C2. The lower thirty two bits of the *configuration data* provide the data to be stored at the indicated address within the appropriate bank of 4-LUTs within C1. The upper thirty two bits of the *configuration data* provide the data to be stored at the indicated address within the appropriate bank of 4-LUTs within C2. Note that the upper thirty two bits of the *configuration data* are only used by instructions configuring RCE MULs.

When operating in 8-LUT mode, the entire eight bits of the *LUT address* are used to address each of the 8-bit LUTs within C1 and C2. The lower thirty two bits of the *configuration data* provide the data to be stored at the indicated address within C1. The upper thirty two bits of the *configuration data* provide the data to be stored at the indicated address within C2. Note that the upper thirty two bits of the *configuration data* are only used by instructions configuring RCE MULs.

6.8.4 Configuring the Bank Address for LUTS C1 and C2

When operating in 4-LUT mode, bits [2:0] of the *configuration data* indicate which of the eight banks within C1 is to be used during system operation. Similarly, bits [5:3] of the *configuration data* indicate which of the eight banks within C2 is to be used during system operation when operating in 4-LUT mode. This set of instructions does not use the *LUT address* field. Note that bits [5:3] of the *configuration data* are only used by instructions configuring RCE MULs.

6.8.5 Storing Configuration Data in the GF Constant Multiplier F

The goal of the Galois Field constant multiplier F is to perform a fixed field multiplication over GF(2^8) where $[A_3 : A_0]$ are the input bytes and $[B_3 : B_0]$ are the output bytes:

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} K_{00} & K_{01} & K_{02} & K_{03} \\ K_{10} & K_{11} & K_{12} & K_{13} \\ K_{20} & K_{21} & K_{22} & K_{23} \\ K_{30} & K_{31} & K_{32} & K_{33} \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}. \tag{6.1}$$

Note that Tables 6.23 and 6.24 detail the bit ordering of $[A_3 : A_0]$ and $[B_3 : B_0]$ during system operation.

32-Bit Input Word Bits	**GF Element**
[31:24]	A_0
[23:16]	A_1
[15:8]	A_2
[7:0]	A_3

Table 6.23: GF constant multiplier bit ordering during operation — inputs $[A_3 : A_0]$

32-Bit Output Word Bits	**GF Element**
[31:24]	B_0
[23:16]	B_1
[15:8]	B_2
[7:0]	B_3

Table 6.24: GF constant multiplier bit ordering during operation — inputs $[B_3 : B_0]$

The core operation in this fixed field multiplication is an 8-bit inner product that must be performed sixteen times, four per row. The four inner products of each row are then XORed to form the final output word. For a known primitive polynomial $p(x)$, $k(x)$ (representing the 8-bit constant), and a generic input $a(x)$, we create a polynomial equation of the form $b(x) = a(x)k(x)$ *mod* $p(x)$ where each coefficient of $b(x)$ is a function of $a(x)$. This results in an 8-bit × 8-bit matrix representing the coefficients of $b(x)$ in terms of $a(x)$. To illustrate the creation of this matrix, the following example is provided. Let:

$$\begin{aligned} k(x) &= (02)_{16} = (00000010)_2 = x \\ p(x) &= x^8 + x^4 + x^3 + x + 1 \\ a(x) &= a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \end{aligned}$$

Therefore, we see that:

$$\begin{aligned} b(x) &= a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x \ mod \ p(x) \\ b(x) &= a_6x^7 + a_5x^6 + a_4x^5 + [a_3 + a_7]x^4 + [a_2 + a_7]x^3 + a_1x^2 + [a_0 + a_7]x + a_7 \end{aligned}$$

This yields the resultant mapping:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}. \tag{6.2}$$

An 8 × 8 matrix must be generated for each K_{xy}, resulting in a total of sixteen matrices. The *LUT address* is used to indicate the K_{xy} element to be configured as detailed in Table 6.25.

LUT Address	**K Element**
00h	K_{00}
01h	K_{01}
02h	K_{02}
03h	K_{03}
04h	K_{10}
05h	K_{11}
06h	K_{12}
07h	K_{13}
08h	K_{20}
09h	K_{21}
0Ah	K_{22}
0Bh	K_{23}
0Ch	K_{30}
0Dh	K_{31}
0Eh	K_{32}
0Fh	K_{33}

Table 6.25: LUT addresses for GF constant multiplier K_{xy} elements

Table 6.26 details the format of the *configuration data* when configuring an 8 × 8 matrix for a given K_{xy} element. The rows are assigned numbers from 0 to 7 where 0 is the top row and 7 is the bottom row of the 8 × 8 matrix. The word representing a given row is formed by concatenating the bits in the row from left to right with the left-most bit forming the MSB of the word and the right-most bit forming the LSB of the word.

Configuration Data Bit(s)	**Row**
[63:56]	7
[55:48]	6
[47:40]	5
[39:32]	4
[31:24]	3
[23:16]	2
[15:8]	1
[7:0]	0

Table 6.26: GF constant multiplier constant matrix configuration data format

6.8.6 Configuring the Byte-Wise Crossbar Switches

The byte-wise crossbar switches operate on bytes within the 128-bit datapath. The *configuration data* is divided into sixteen 4-bit mapping codes. The code associated to a given output byte indicates which input byte will be mapped to that output byte. This set of instructions does not use the *LUT address* field. Note that it is possible to map a single input byte to multiple output bytes. Table 6.27 details the format of the *configuration data* when configuring the byte-wise crossbar switches.

Configuration Data Bit(s)	**Code for Output Byte**
[63:60]	15
[59:56]	14
[55:52]	13
[51:48]	12
[47:44]	11
[43:40]	10
[39:36]	9
[35:32]	8
[31:28]	7
[27:24]	6
[23:20]	5
[19:16]	4
[15:12]	3
[11:8]	2
[7:4]	1
[3:0]	0

Table 6.27: Byte-wise crossbar switches mapping codes configuration data format

6.8.7 Configuring the Input Switch

The input switch determines if the datapath takes it's input from the system inputs or the feedback of the system outputs. Bits [3:0] of the *configuration data* indicate from which location the datapath will take its inputs. A one indicates that the datapath will take the associated 32-bit input from the feedback of the system output while a zero indicates that the datapath will take the associated 32-bit input from the system input. Note that the system inputs are registered and Table 6.29 details the mapping of the buses to the RCE and RCE MUL columns. This instruction does not use the *LUT address* field. Table 6.28 details the format of the *configuration data* when configuring the input switch.

Configuration Data Bit	**Column Controlled**
0	0
1	1
2	2
3	3

Table 6.28: Input switch configuration data format

Input Switch Output	**Input to RCE or RCE MUL**
Registered system input 1	RCE 00
Whitening block 30 output	RCE 00
Registered system input 2	RCE MUL 01
Whitening block 31 output	RCE MUL 01
Registered system input 3	RCE 02
Whitening block 32 output	RCE 02
Registered system input 4	RCE MUL 03
Whitening block 33 output	RCE MUL 03

Table 6.29: Input switch output to RCE or RCE MUL input mapping

6.8.8 Configuring the eRAM Address Registers

The eRAM address registers provide an 8-bit address to each of the eRAMs and are configured in two groups via two instructions — the right eRAMs (eRAM 01 — eRAM 08) and the left eRAMs (eRAM 09 — eRAM 16). These instructions do not use the *LUT address* field. Table 6.30 details the format of the *configuration data* when configuring the eRAM address registers.

	Right eRAM	**Left eRAM**
Configuration Data Bits	**Address Configured**	Address Configured
[63:56]	eRAM 08	eRAM 16
[55:48]	eRAM 07	eRAM 15
[47:40]	eRAM 06	eRAM 14
[39:32]	eRAM 05	eRAM 13
[31:24]	eRAM 04	eRAM 12
[23:16]	eRAM 03	eRAM 11
[15:8]	eRAM 02	eRAM 10
[7:0]	eRAM 01	eRAM 9

Table 6.30: eRAM address registers configuration data format

6.8.9 Configuring the eRAM Input Source

The input source for each of the eRAMs is configured using a 4-bit mapping code. Tables 6.31, 6.32, 6.33, and 6.34 detail the format of the input source mapping code for the eRAMs. Note that mapping codes 1001_2 through 1111_2 are not used. This instruction does not use the *LUT address* field. Table 6.35 details the format of the *configuration data* when configuring the eRAM input source.

Mapping Code	**Input Source**
0000	RCE 02 output
0001	RCE MUL 03 output
0010	RCE 12 output
0011	RCE MUL 13 output
0100	RCE 22 output
0101	RCE MUL 23 output
0110	RCE 32 output
0111	RCE MUL 33 output
1000	System input 3

Table 6.31: eRAM 01 — eRAM 04 input source configuration data format

Mapping Code	**Input Source**
0000	RCE 02 output
0001	RCE MUL 03 output
0010	RCE 12 output
0011	RCE MUL 13 output
0100	RCE 22 output
0101	RCE MUL 23 output
0110	RCE 32 output
0111	RCE MUL 33 output
1000	System input 4

Table 6.32: eRAM 05 — eRAM 08 input source configuration data format

Mapping Code	**Input Source**
0000	RCE 00 output
0001	RCE MUL 01 output
0010	RCE 10 output
0011	RCE MUL 11 output
0100	RCE 20 output
0101	RCE MUL 21 output
0110	RCE 30 output
0111	RCE MUL 31 output
1000	System input 2

Table 6.33: eRAM 09 — eRAM 12 input source configuration data format

Mapping Code	**Input Source**
0000	RCE 00 output
0001	RCE MUL 01 output
0010	RCE 10 output
0011	RCE MUL 11 output
0100	RCE 20 output
0101	RCE MUL 21 output
0110	RCE 30 output
0111	RCE MUL 31 output
1000	System input 1

Table 6.34: eRAM 13 — eRAM 16 input source configuration data format

Configuration Data Bits	**eRAM Input Source Configured**
[63:60]	eRAM 16
[59:56]	eRAM 15
[55:52]	eRAM 14
[51:48]	eRAM 13
[47:44]	eRAM 12
[43:40]	eRAM 11
[39:36]	eRAM 10
[35:32]	eRAM 09
[31:28]	eRAM 08
[27:24]	eRAM 07
[23:20]	eRAM 06
[19:16]	eRAM 05
[15:12]	eRAM 04
[11:8]	eRAM 03
[7:4]	eRAM 02
[3:0]	eRAM 01

Table 6.35: eRAM input source configuration data format

6.8.10 Configuring the eRAM Input Source to the RCEs and RCE MULs

The eRAM input source for each of the RCEs and RCE MULs is configured using a 3-bit mapping code. Table 6.36 details the format of the eRAM input source mapping code for the RCEs and RCE MULs in columns 0 and 1. Table 6.37 details the format of the eRAM input source mapping code for the RCEs and RCE MULs in columns 2 and 3. This instruction does not use the *LUT address* field. Table 6.38 details the format of the *configuration data* when configuring the RCE and RCE MUL eRAM input source.

Mapping Code	**RCE/RCE MUL eRAM Input Source**
000	eRAM 09
001	eRAM 10
010	eRAM 11
011	eRAM 12
100	eRAM 13
101	eRAM 14
110	eRAM 15
111	eRAM 16

Table 6.36: RCE/RCE MUL eRAM input source mapping codes — columns 0 and 1

Mapping Code	RCE/RCE MUL eRAM Input Source
000	eRAM 01
001	eRAM 02
010	eRAM 03
011	eRAM 04
100	eRAM 05
101	eRAM 06
110	eRAM 07
111	eRAM 08

Table 6.37: RCE/RCE MUL eRAM input source mapping codes — columns 2 and 3

Configuration Data Bits	RCE or RCE eRAM Input Source Configured
[63:48]	Not used
[47:45]	RCE MUL 31
[44:42]	RCE 30
[41:39]	RCE MUL 21
[38:36]	RCE 20
[35:33]	RCE MUL 11
[32:30]	RCE 10
[29:27]	RCE MUL 01
[26:24]	RCE 00
[23:21]	RCE MUL 33
[20:18]	RCE 32
[17:15]	RCE MUL 23
[14:12]	RCE 22
[11:9]	RCE MUL 13
[8:6]	RCE 12
[5:3]	RCE MUL 03
[2:0]	RCE 02

Table 6.38: RCE and RCE MUL eRAM input source configuration data format

6.8.11 Configuring the eRAM Write Strobes

The eRAM write strobes are set or reset using a single instruction. A one indicates the write strobe being set and a zero indicates the write strobe being reset. Therefore, to load data into an eRAM requires one instruction to set the appropriate eRAM write strobe followed by another instruction to reset the appropriate eRAM write strobe. This instruction does not use the *LUT address* field. Table 6.39 details the format of the *configuration data* when configuring the eRAM write strobes.

Configuration Data Bit(s)	**eRAM Write Strobe Configured**
[63:16]	Not used
15	eRAM 16
14	eRAM 15
13	eRAM 14
12	eRAM 13
11	eRAM 12
10	eRAM 11
9	eRAM 10
8	eRAM 09
7	eRAM 08
6	eRAM 07
5	eRAM 06
4	eRAM 05
3	eRAM 04
2	eRAM 03
1	eRAM 02
0	eRAM 01

Table 6.39: eRAM write strobe configuration data format

6.8.12 Configuring the Whitening Blocks

Each whitening block accepts one 32-bit inputs and provides one 32-bit output. The whitening blocks operate upon both the 32-bit input and a stored 32-bit value using a 3-bit configuration code as detailed in Table 6.40. This instruction does not use the *LUT address* field. Table 6.41 details the format of the *configuration data* when configuring the whitening blocks.

Configuration Code	**Operation**
00	Bit-wise XOR
01	Modular 2^{32} addition
10	Pass through of input
11	Pass through of input

Table 6.40: Whitening block functionality

Configuration Data Bits	**Whitening Block Configured**
[63:12]	Not used
[11:9]	Whitening Block 33
[8:6]	Whitening Block 32
[5:3]	Whitening Block 31
[2:0]	Whitening Block 30

Table 6.41: Whitening blocks configuration data format

6.8.13 Storing Data in the Whitening Blocks

Data is stored in the whitening blocks via four instructions. Each whitening block is loaded from the RCE or RCE MUL directly above it as detailed in Table 6.42. These instructions do not use the *LUT address* or *configuration data* fields.

Whitening Block	**Input Source**
30	RCE 30
31	RCE MUL 31
32	RCE 32
33	RCE MUL 33

Table 6.42: Whitening block input source

6.8.14 Configuring the Output Enables for the RCEs, RCE MULs, and Whitening Blocks

The output enables for the RCEs, RCE MULs, and whitening blocks may be disabled to support *overfull* instruction cycles. When the output enable for a given element is disabled, the output register of the element will not update on the datapath clock edge. Enabling the output enable for a given element allows the output register of the element to update on the datapath clock edge. An output is enabled when its output enable is a one and disabled when its output enable is a zero. This instruction does not use the *LUT address* field. Table 6.43 details the format of the *configuration data* when configuring the RCE, RCE MUL, and whitening block output enables.

Configuration Data Bit(s)	**Output Enable Configured**
[63:20]	Not used
19	Whitening Block 33
18	Whitening Block 32
17	Whitening Block 31
16	Whitening Block 30
15	RCE MUL 33
14	RCE 32
13	RCE MUL 31
12	RCE 30
11	RCE MUL 23
10	RCE 22
9	RCE MUL 21
8	RCE 20
7	RCE MUL 13
6	RCE 12
5	RCE MUL 11
4	RCE 10
3	RCE MUL 03
2	RCE 02
1	RCE MUL 01
0	RCE 00

Table 6.43: Output enable configuration data format

6.8.15 Configuring the Flag Register

The flag register provides both user-defined and hard-coded flags to external peripherals for use in controlling the COBRA architecture. The flag register may be written to as specified by the programmer. Table 6.44 details the functionality of each flag within the flag register. Note that user-defined flags may be used as specified by the programmer — examples include indicators to external peripherals that the COBRA architecture is ready to receive data or that a particular operation is in process. A change in a flag bit is assumed to be evaluated by the external peripherals on the next datapath clock rising edge. Any necessary action by the peripherals is assumed to occur on the second datapath clock rising edge following the change in the flag bit. This instruction does not use the *LUT address* field. Table 6.45 details the format of the *configuration data* when configuring the flag register.

Flag Register Bit	Flag Name	Flag Function
7	flag 5	User defined
6	flag 4	User defined
5	flag 3	User defined
4	idle	1 - encryption/decryption in progress 0 - no encryption/decryption in progress
3	flag 1	User defined
2	flag 0	User defined
1	data valid	1 - data on output pins is valid 0 - data on output pins is not valid
0	ready	1 - system is ready to perform encryption/decryption and waits for *go* to become active 0 - system is not ready to perform encryption/decryption

Table 6.44: Flag register functionality

Configuration Data Bit(s)	Flag Configured
[63:8]	Not used
7	flag 5
6	flag 4
5	flag 3
4	idle
3	flag 1
2	flag 0
1	data valid
0	ready

Table 6.45: Flag register configuration data format

6.8.16 Configuring the Jump Address

Once an encryption or decryption has been completed, the jump address specifies the address in the iRAM to set the program counter to so as to begin another encryption or decryption operation. The address is one less than the location at which the system waits for the *go* input signal to become active as the program counter will be automatically incremented by one following the jump operation. This instruction does not use the *LUT address* field. Table 6.46 details the format of the *configuration data* when configuring the jump address.

Configuration Data Bits	Jump Address Configured
[63:12]	Not used
[11:0]	Jump address

Table 6.46: Jump address configuration data format

Chapter 7

COBRA Software Tools

A number of software tools were developed to support hardware implementations in the COBRA architecture. These include:

- An assembler to ease coding.
- A timing analyzer to provide datapath operating frequency information.

7.1 Command Line Execution

The COBRA software tool suite is command-line driven and has been coded in C to guarantee portability across all possible software platforms.

7.2 Assembler

The COBRA assembler was developed to ease programming of the architecture by eliminating the need for the programmer to manually enter bit patterns. Instead, a custom assembly language and assembler were created that fully support all of the instructions detailed in Section 6.8. The assembler parses the input file and generates a vector file used to configure the COBRA architecture.

The COBRA assembly language is subdivided based on operation type. A description of each language construct, examples of construct usage, and input file format requirements may be found in Appendix B.

7.3 Timing Analyzer

The COBRA timing analyzer was developed to ease the determination of the maximum operating frequency of the COBRA architecture for a given program. The timing analyzer evaluates the various configurations of the COBRA architecture as specified by a given program and generates a maximum frequency operating frequency for each configuration as well as for the set of configurations. The maximum operating frequency is used by the programmer in determining the iRAM clock frequency — given that two iRAM clock cycles are required to load and execute an instruction, the iRAM clock frequency is set to be twice that of the maximum operating frequency determined by the timing analyzer.

7.3.1 Program Analysis

The COBRA timing analyzer operates in conjunction with the COBRA assembler. The timing analyzer creates COBRA mapping structures which contain the delay values for the RCEs, RCE MULs, Byte-Wise Crossbars, and Whitening Blocks. Within the mapping structure, sub-structures are created for the RCEs and RCE MULs which contain delay values for their sub-elements. A mapping structure is dynamically allocated and each element within the structure is initialized to a delay value of zero. As each assembly instruction is compiled by the assembler, the timing analyzer populates the mapping structure with delay values for the elements that are enabled. The timing analyzer creates a new mapping structure whenever an instruction that updates a COBRA element is detected. When the new structure is created, all of the delay values for the COBRA elements that were not updated are copied into the new structure while the delay values for the updated COBRA element are calculated based on the instruction being assembled. This process results in multiple mappings for a given program based on the different configurations of the COBRA architecture that are used to implement the desired block cipher.

Each mapping structure maintains a *unique* flag. A mapping structure is considered to be unique if a reconfiguration instruction that creates a mapping structure is followed by a run-time instruction. The *unique* flag is used by the timing analyzer to eliminate redundant mappings. Redundant mappings occur when multiple reconfiguration instructions are executed in sequence. Each reconfiguration instruction results in the creation of a new mapping structure but the intermediate structures are not individually unique as no run-time operations occur using those specific mappings. Only the final mapping that is created after the last instruction in the sequence is assembled is actually unique. Therefore, the timing analyzer removes mapping structures that are not considered to be unique before performing the system timing analysis.

7.3.2 Delay Values

The delay values used by the timing analyzer in evaluating the mapping structures are detailed in Table 7.1. These estimates were obtained by synthesizing the COBRA atomic elements using Synopsys' Design Compiler targeting a 0.35 micron process technology [64].

Element	**Delay (ns)**
RCE/RCE MUL A	0.5
RCE/RCE MUL B	1.2
RCE/RCE MUL C	5.0
RCE MUL D	5.5
RCE/RCE MUL E	2.0
RCE/RCE MUL F	2.5
Byte-Wise Crossbar	1.3
Whitening Block	2.0
Input Multiplexor	Unspecified
Routing	Unspecified

Table 7.1: Timing analyzer delay values

7.3.3 System Timing Calculations

The timing analyzer begins the system timing analysis by calculating the associated delay of each RCE and RCE MUL for every unique mapping. The value is calculated by adding the delay values of the sub-elements in the sub-structure associated with the given RCE or RCE MUL. The timing analyzer then calculates the worst case delay for each row in each mapping structure by selecting the delay value of the RCE or RCE MUL that is the longest of the four possible values. Similarly, the worst case delay for the Whitening Block row is calculated for each mapping structure by selecting the delay value of the Whitening Block that is the longest of the four possible values. The delay values for rows one (RCE/RCE MUL elements 10, 11, 12, and 13) and three (RCE/RCE MUL elements 30, 31, 32, and 33) of each mapping structure are then updated by adding the delay values for the associated Byte-Wise Crossbars 1 and 2 respectively. Finally, the delay value for row three of each mapping structure is updated by adding the delay value for the associated Whitening Block row.

Once the delay values for all of the rows in each of the mapping structures has been calculated, the maximum delay of each mapping structure may be calculated. However, the maximum delay value is not simply the sum of the row values as each RCE or RCE MUL elements may have registered outputs. Each mapping structure is evaluated to determine the location of any registered outputs. If registered outputs are found, a delay value must be calculated for each register-to-register block. To illustrate this concept, if rows one and three have registered outputs, a delay value must be calculated for the block between row three and row one (encompassing the Whitening Blocks, and row zero, Byte-Wise Crossbar 1, and row one) and for the block between row one and row three (encompassing row two, Byte-Wise Crossbar 2, and row three). Once all of the blocks are calculated, the maximum delay of the mapping structure is calculated as the maximum delay of the blocks. Note that if there are either no registered outputs or one row of registered outputs then the maximum delay of the mapping structure is simply the sum of all of the row delays.

The final output of the timing analyzer is a report of the delays for each unique mapping structure broken down by row. The delay associated to Byte-Wise Crossbar 1, Byte-Wise

Crossbar 2, and the Whitening Blocks are also listed so that the programmer can determine how much of the delay in rows one and three are associated with these COBRA elements versus the RCEs and RCE MULs in the associated row. Additionally, the timing analyzer calculates the worst case system clock period by comparing the maximum delay of each mapping structure. From this value the timing analyzer calculates the minimum period and maximum frequency of the iRAM clock. The iRAM clock period is calculated to be half that of the worst case system clock period so as to be able to both fetch and execute an instruction, which requires two iRAM clock cycles, within the worst case system clock period. This allows the datapath to stabilize before the next rising edge of the datapath clock.

Chapter 8

Results

The COBRA architecture was implemented in VHDL using a bottom-up design and test methodology [115]. Key scheduling and encryption for a number of block ciphers were either coded in COBRA assembly language and assembled into microcode or written directly as COBRA microcode. The resulting microcode was then used during simulation to perform both key scheduling and encryption. System operation was controlled via a VHDL test bench that served to load the iRAM with the appropriate microcode, apply the appropriate control signals to the architecture, and examine the encrypted data for validity. The iRAM clock frequency was set in the test bench based on the output of the timing analyzer. The datapath clock frequency was set after examining the microcode and optimizing the instruction stream to minimize both *underfull* and *overfull* instruction cycles. Finally, COBRA implementations of RC6, Rijndael, and Serpent were tested using the test vectors provided in the AES submission package [13, 37, 119].

8.1 Targeted Ciphers

A subset of the block ciphers studied was chosen for implementation in the COBRA architecture. In particular, ciphers were chosen based on the orthogonality of their core functions in an effort to exercise as many of the architecture's features as possible. The 128-bit block ciphers RC6, Rijndael, and Serpent were selected for implementation based on their core element requirements spanning the set of required functionality for the COBRA architecture. Table 8.1 details the core operations of the targeted block ciphers and is a subset of Tables 5.1 and 5.2, detailed in Chapter 5.

	XOR	**ADD**						**Int.**
	AND	**SUB**	**Fixed**	**Var.**	**MUL**	**GF**		**Block**
Algorithm	**OR**	**Mod**	**Shift**	**Rot.**	**Mod**	**MUL**	**LUT**	**Size**
RC6	●	2^{32}	●	●	2^{32}			32
Rijndael	●		●			GF(2^8)	8-to-8	32
Serpent	●		●				4-to-4	32

Table 8.1: Core operations of the block ciphers implemented in the COBRA architecture

Given the large number of implementations of DES and IDEA, both block ciphers were also considered for implementation but proved to be poorly suited to the COBRA architecture. DES requires both an initial and a final permutation, each of which is a bit-wise reordering of the 64-bit data. The COBRA architecture currently supports byte-wise permutations at it's lowest level. While bit-wise shifts and rotations are possible, bit-wise permutations are extremely difficult to implement. However, in the case of DES, the initial and final permutations occur at the start and end of encryption. A solution to this problem would be to assume that the permutations are implemented via wiring external to the chip such that the data arrives already permuted and will be permuted via the wiring on the output pins upon leaving the chip. While this solves the issue of the initial and final permutations, the P permutation within the DES f-function that performs a bit-wise reordering of the 32-bit data remains an open issue. While a 32-bit crossbar switch was considered for addition to the RCE or RCE MUL structures, the decision was made to avoid the implementation of special purpose hardware for specific ciphers and to maintain the coarse-grained nature of the COBRA architecture. The E expansion function within the DES f-function also poses an implementation problem. The E function expands the

32-bit datapath into a 48-bit datapath that is reduced back to a 32-bit datapath through the S-Boxes. Once again, bit-wise expansion and collapse of the datapath was determined to be special purpose hardware that was outside the bounds of a coarse-grained architecture and was therefore not supported. The lack of support for bit-wise permutations and expansion functions resulted in DES no longer being considered for implementation in the COBRA architecture.

In the case of IDEA, all core operations are supported via standard COBRA functionality except for the $mod\ 2^{16}\ +\ 1$ multiplication. Hardware implementations of IDEA implement the $mod\ 2^{16}\ +\ 1$ multiplication as a $mod\ 2^{16}$ multiplication and then increment the result if it is nonzero. This type of implementation is non-trivial in the COBRA architecture because it requires a data dependent addition based on the output of the $mod\ 2^{16}$ multiplier. This type of operation would also be highly specific to IDEA and would require the addition of internal flags that could be evaluated by the elements within the RCE or RCE MUL structures to determine if an the result needed to be incremented. As in the case of bit-wise expansion and permutation functions in DES, support of $mod\ 2^{16}\ +\ 1$ multiplication was not implemented in the COBRA architecture. The lack of support for $mod\ 2^{16} + 1$ multiplication resulted in IDEA no longer being considered for implementation in the COBRA architecture.

8.2 Performance Analysis

Table 8.2 details the performance in terms of cycles per encryption for RC6, Rijndael, and Serpent when implemented in software, specifically either C or C++, for 128-bit blocks and 128-bit master keys. As expected, the performance for each algorithm varies greatly based on the targeted platform. It has also been shown that performance varies significantly based on both the software compiler and the operating system used to implement a given cipher [17]. Due to this wide variance in performance, the best software results for each algorithm were used to represent software performance for the purpose of performance comparisons.

Platform	**RC6**	**Rijndael**	**Serpent**
Intel 486DX2	2,680 [59]	2,370 [59]	12,900 [59]
Cyrix 6x86MX	706 [59]	847 [59]	2,042 [59]
Intel Pentium	243 [158]	284 [158]	900 [158]
Intel Pentium II	**223** [14]	237 [14]	952 [59]
Intel Pentium Pro	340 [17]	682 [17]	1,285 [17]
Intel Pentium III	318 [17]	681 [17]	1,261 [17]
TurboSparc 170 MHz	867 [59]	884 [59]	1,785 [59]
SuperSparc 50 MHz	480 [59]	440 [59]	845 [59]
UltraSparcIIi 270 MHz	1,164 [59]	333 [93]	979 [59]
Alpha EV45	1,247 [59]	907 [59]	1,785 [59]
Alpha EV56	559 [63]	439 [148]	984 [148]
Alpha EV6	382 [148]	285 [63]	854 [148]
HP PA7000	1,085 [59]	735 [59]	1,345 [59]
HP PA8500	580 [158]	168 [158]	580 [158]
Itanium (Merced)	490 [158]	**124** [158]	**565** [158]
ARM	790 [98]	1,467 [98]	N/A
8051	14,500 [98]	3,168 [59]	N/A
6805	106,000 [82]	9,500 [82]	126,000 [82]
Z80	34,736 [122]	25,494 [122]	71,924 [122]

Table 8.2: Software implementations — best cycle counts based on platform

8.2.1 RC6

As detailed in Chapter 4, RC6 is specified as a family of algorithms with three parameters. RC6-$w/r/b$ refers to an algorithm with a word size of w bits, r rounds, and a key size of b bytes. For the COBRA implementation, we consider RC6-32/20/16 which conforms to the AES specifications with a key size of 128 bits [119].

RC6 is easily partitioned into three stages — stage one key scheduling, stage two key scheduling, and encryption. Stage one key scheduling computes the *S[i]* array by initializing *S[0]* to P_w and then computing $S[i] = S[i-1] + Q_w$. The COBRA implementation utilizes the generic flag bits in the flag register to indicate to the external system when the P_w, Q_w, and *L[0:3]* parameters must be provided. Note that the *L[0:3]* parameters are used in the second stage of RC6 key scheduling and are stored in eRAM 10. The P_w parameter is stored at address 00H and the Q_w parameter is stored at address 01H in eRAM 13. eRAM 13 is also configured as the embedded RAM input to RCE 00. RCE 00 is initially configured to pass the eRAM value as its output and the address of eRAM 13 is set to 00H. RCE 10, RCE 20, and whitening block 30 are configured as pass-through elements and LB30 is configured as a registered pass-through element, the output of which is stored in eRAM 09. After the first iteration through the array, P_w is stored in eRAM 09 at address 00H and the addresses of eRAM 09 and eRAM 13 are set to 01H, resulting in eRAM 13 outputting the value of Q_w. The input multiplexors are configured to feedback mode, resulting in the primary input to RCE 00 being the registered output of LB30, in this case *S[0]*. RCE 00 is reconfigured to output the sum of the primary input and the embedded RAM input, adding Q_w to *S[i]*. The registered result of this addition is seen at the output of LB30 and stored in eRAM 09. This process continues with the address of eRAM 09 being incremented to store the next *S[i]* until all forty four values of the S array have been calculated. Figure 8.1 details the COBRA architecture mapping for the first stage of RC6 key scheduling.

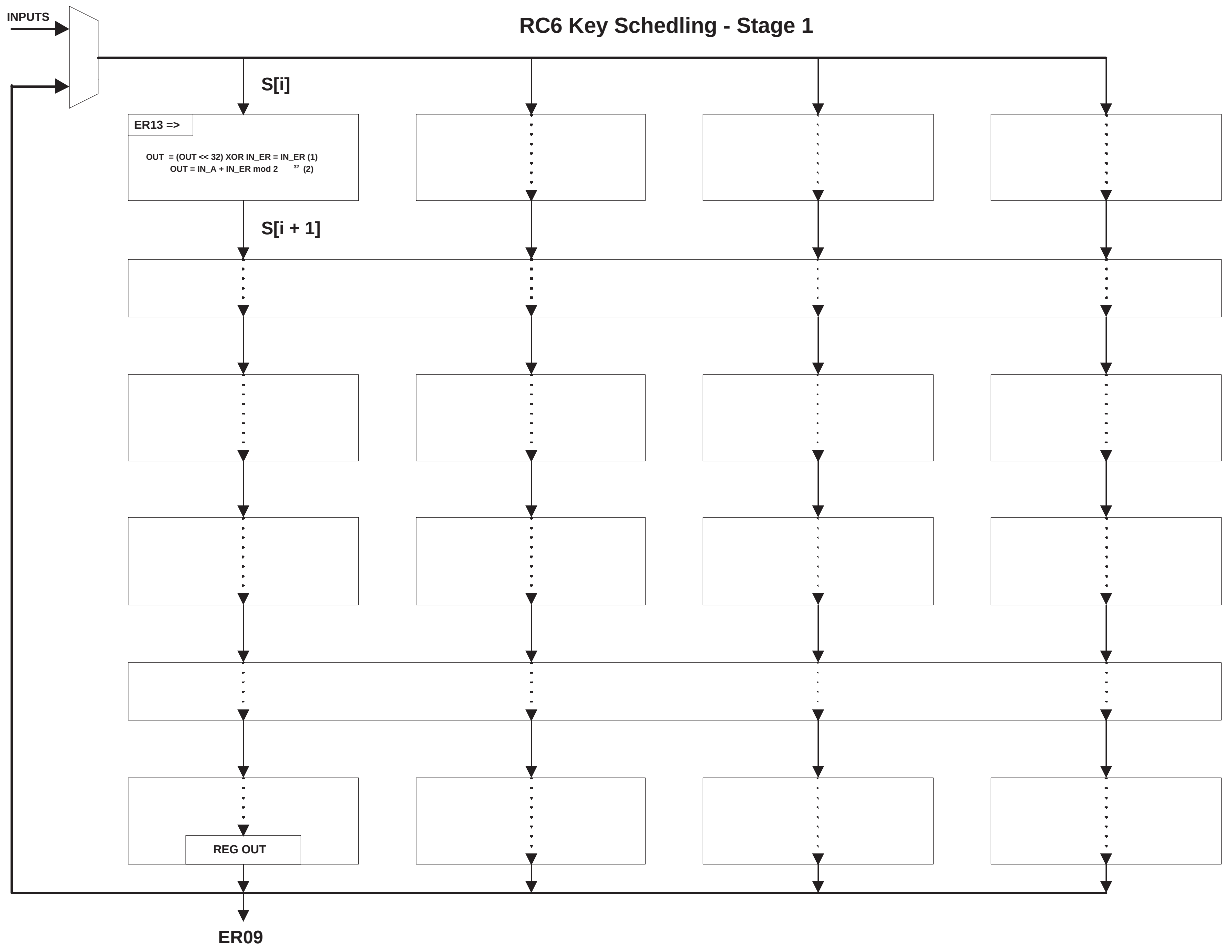

INPUTS
RC6 Key Schedling - Stage 1
S[i]
ER13 =>
OUT = (OUT << 32) XOR IN_ER = IN_ER (1)
OUT = IN_A + IN_ER mod 2 32 (2)
S[i + 1]
REG OUT
ER09

Figure 8.1: RC6 stage one key scheduling — COBRA mapping

Stage two of RC6 key scheduling calculates the final values of *S[i]* based on the procedure provided in the RC6 specification [119], iterating 132 times over the loop counter i to calculate the final values of the subkeys *S[i]*. RCE 00 loads the current value of *S[i]* from eRAM 09 and adds this value to the current value of the internal variables A and B before rotating the resultant 32-bit word left three bits. Note that for the first iteration, A, B, and i are zero and RCE 00 is configured to output the value of *S[0]* which is obtained from the embedded RAM input. RCE 30 is configured to output the value of B and this output is forced to zero for the first iteration. RCE 10 is configured to calculate the intermediate value of *L[j]* by adding the new value of A to the old value of *L[j]* (from eRAM 10) and the current value of B. However, because we need to maintain the value of A for the next iteration, RCE 11 is configured to output the value of A calculated by RCE 00. Note that the j loop counter is set to zero for the first iteration. RCE 13 calculates the data dependent rotation value for *L[j]* by adding the newly calculated value of A to the current value of B. RCE 20 is configured to rotate the intermediate value of *L[j]* by the lower five bits of the output from RCE 13. The resultant output of RCE 20 is the final value of both B and *L[j]*. RCE 30 is configured to output the registered value of the updated A (which is also *S[i]*) which is stored in eRAMs 09 and 15 for odd values of i and eRAMs 13 and 14 for even values of i. Note that the output of RCE 30 is also fed back to RCE 00. RCE 31 and RCE 33 are configured to output the registered value of the updated B (which is also *L[j]*) and the output of RCE 31 is stored in eRAM 10, updating the value of *L[j]*. Note that while the loop counter i is incremented at each iteration, the loop counter j is incremented *mod* c, where c is the master key length divided by the word length. For RC6-32/20/16, c is equal to four. The updated value of B output from RCE 33 is fed back to RCE 03 for the next iteration. Finally, RCE 32 is configured to output the registered value of the updated A which is stored in eRAMs 05, 06, and 07 in the case of odd subkeys. The choice of eRAMs for the storage of subkeys was made to facilitate the encryption process. The stage two key scheduling process is repeated until all *S[i]* have been calculated. Figure 8.2 details the COBRA architecture mapping for the second stage of RC6 key scheduling.

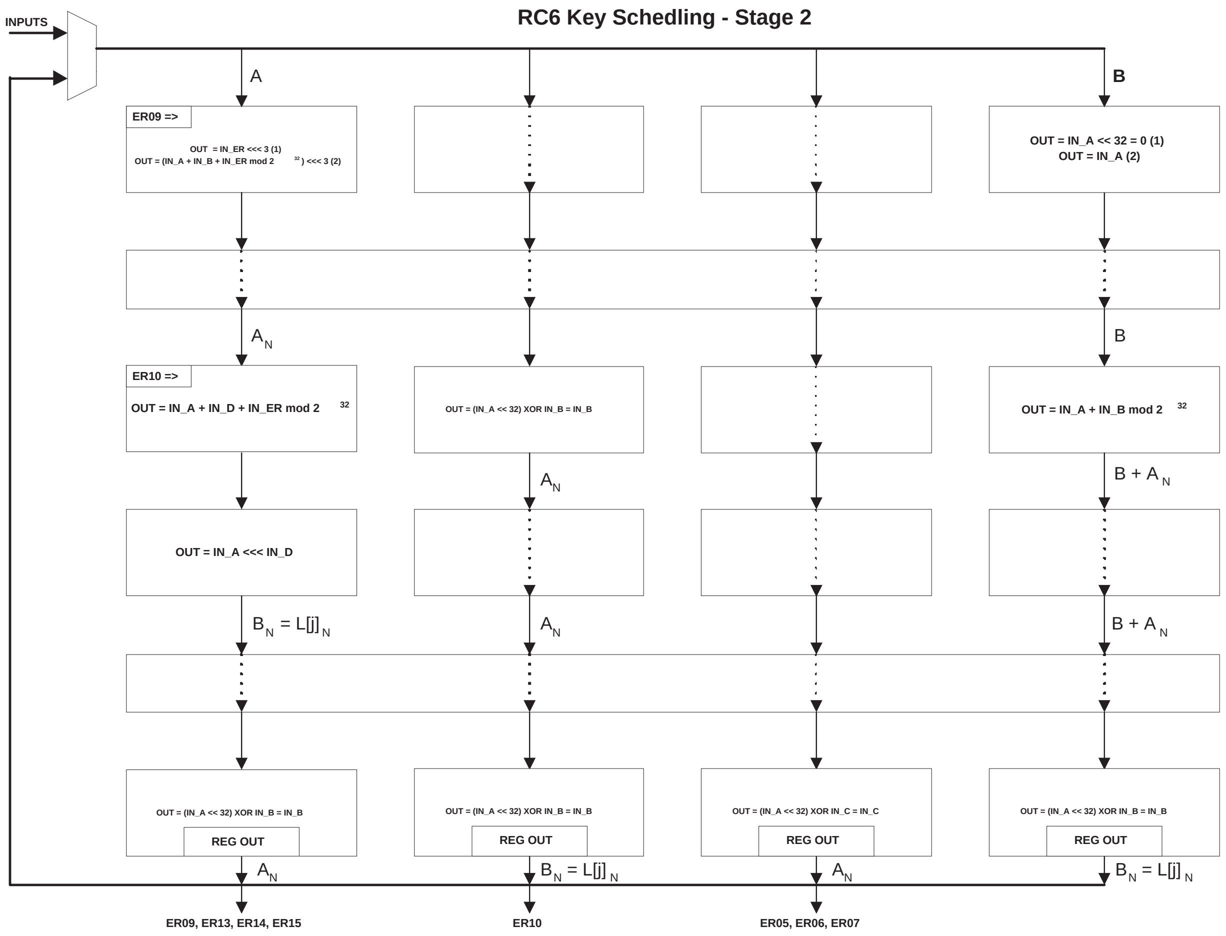

RC6 Key Schedling - Stage 2
INPUTS
A
B
ER09 =>
OUT = IN_ER <<< 3 (1)
OUT = (IN_A + IN_B + IN_ER mod 2 32) <<< 3 (2)
OUT = IN_A << 32 = 0 (1)
OUT = IN_A (2)
A N
B
ER10 =>
OUT = IN_A + IN_D + IN_ER mod 2 32
OUT = (IN_A << 32) XOR IN_B = IN_B
OUT = IN_A + IN_B mod 2 32
A N
B + A N
OUT = IN_A <<< IN_D
B N = L[j] N
A N
B + A N
OUT = (IN_A << 32) XOR IN_B = IN_B
OUT = (IN_A << 32) XOR IN_B = IN_B
OUT = (IN_A << 32) XOR IN_C = IN_C
OUT = (IN_A << 32) XOR IN_B = IN_B
REG OUT
REG OUT
REG OUT
REG OUT
A N
B N = L[j] N
A N
B N = L[j] N
ER09, ER13, ER14, ER15
ER10
ER05, ER06, ER07

Figure 8.2: RC6 stage two key scheduling — COBRA mapping

At the start of encryption, key whitening is performed by adding *S[0]* is to B and *S[1]* to D. We then iterate over the round function twenty times before performing a final key whitening operation. This key whitening is accomplished by adding $S[2r\ +\ 2]$ to A and $S[2r\ +\ 3]$ to C, where r is twenty, the number of rounds in the cipher.

The RC6 round function divides the 128-bit plaintext into four 32-bit blocks (as described in Section 4.2.1) and performs the following operations in parallel as per the RC6 specification [119]:

$$\begin{aligned} A_{i+1} &= B_i \\ B_{i+1} &= [(([(2D_i^2 + D_i) \lll 5] \oplus C_i) \lll ([2B_i^2 + B_i] \lll 5)] + S[2i+1] \\ C_{i+1} &= D_i \\ D_{i+1} &= [(([(2B_i^2 + B_i) \lll 5] \oplus A_i) \lll ([2D_i^2 + D_i] \lll 5)] + S[2i] \end{aligned}$$

The format of the round function effectively results in a 192-bit datapath. Each calculation of B and D require two parallel datapaths, one for the primary computation and one for the computation of the data dependent rotation value. Two additional datapaths are required to maintain the old values of B and D as these become the new values of A and C, respectively. Given that COBRA operates via a 128-bit datapath and that determining the new values of A and C require no actual computation, the 192-bit datapath is implemented by storing the old values of B and D in eRAM scratch space while the COBRA datapath is used to perform the computations required to determine the new values of B and D.

Implementation of RC6 encryption requires two rows of the COBRA architecture. Therefore, a total of two rounds of RC6 may be implemented in the COBRA architecture, as detailed in Figure 8.3. The RC6 encryption configuration assumes that the blocks A, B, C, and D are the primary inputs to RCE 00, RCE MUL 01, RCE 02, and RCE MUL 03, respectively. The values of B and D are stored in eRAM 09 and 08 respectively, on the datapath clock edge that indicates the start of a round. RCE 00 and RCE 02 are configured to pass through A_i and C_i, respectively. For the first round, A_i and C_i are the primary inputs to RCE 00 and RCE 02, respectively. However, in all other rounds A_i and C_i are the primary inputs to RCE 02 and RCE 00, respectively. This is as a result of the first RC6 encryption round swapping the location within the COBRA datapath of B_{i+1} and D_{i+1} versus the location of B_i and D_i. Correcting this datapath swap is achieved in the second round of RC6 encryption but results in the swapping of the location within the COBRA datapath of A_{i+2} and C_{i+2} versus the location of A_i and C_i. These swapped A and C datapaths require that RCE 00 and RCE 02 be reconfigured after the first two rounds have been completed to output the appropriate secondary input corresponding to A_i and C_i, respectively.

RCE MUL 01 and RCE MUL 03 are used to calculate the data dependent rotation values for B_{i+1} and D_{i+1}. The data dependent rotation values are also part of the primary computation for both B_{i+1} and D_{i+1}. This allows RCE MUL 11 to XOR the primary computation value of D_{i+1} with A_i and then rotate the result based on the five least significant bits of the primary computation value of B_{i+1} before adding in the appropriate subkey. Similarly, RCE MUL 13 XORs the primary computation value of B_{i+1} with C_i and then rotates the result based on the five least significant bits of the primary computation value of D_{i+1} before adding in the appropriate subkey. The outputs of RCE MUL 11 and RCE MUL 13 are the new values for D_{i+1} and B_{i+1}, respectively. RCE 10 and RCE 12 output the values of B_i and D_i stored in eRAM 09 and eRAM 08, respectively, which become the new values for A_{i+1} and C_{i+1}, respectively. The RCEs and RCE MULs in rows two and three of the COBRA architecture operate in a similar fashion to those of rows zero and one and both crossbar switches are configured as pass-through elements. Note that during rounds one through eighteen, the whitening elements are configured as pass-through elements. When the system is performing rounds nineteen and twenty, whitening elements 30 and 32 are configured to perform the final key whitening operations on the final A and C.

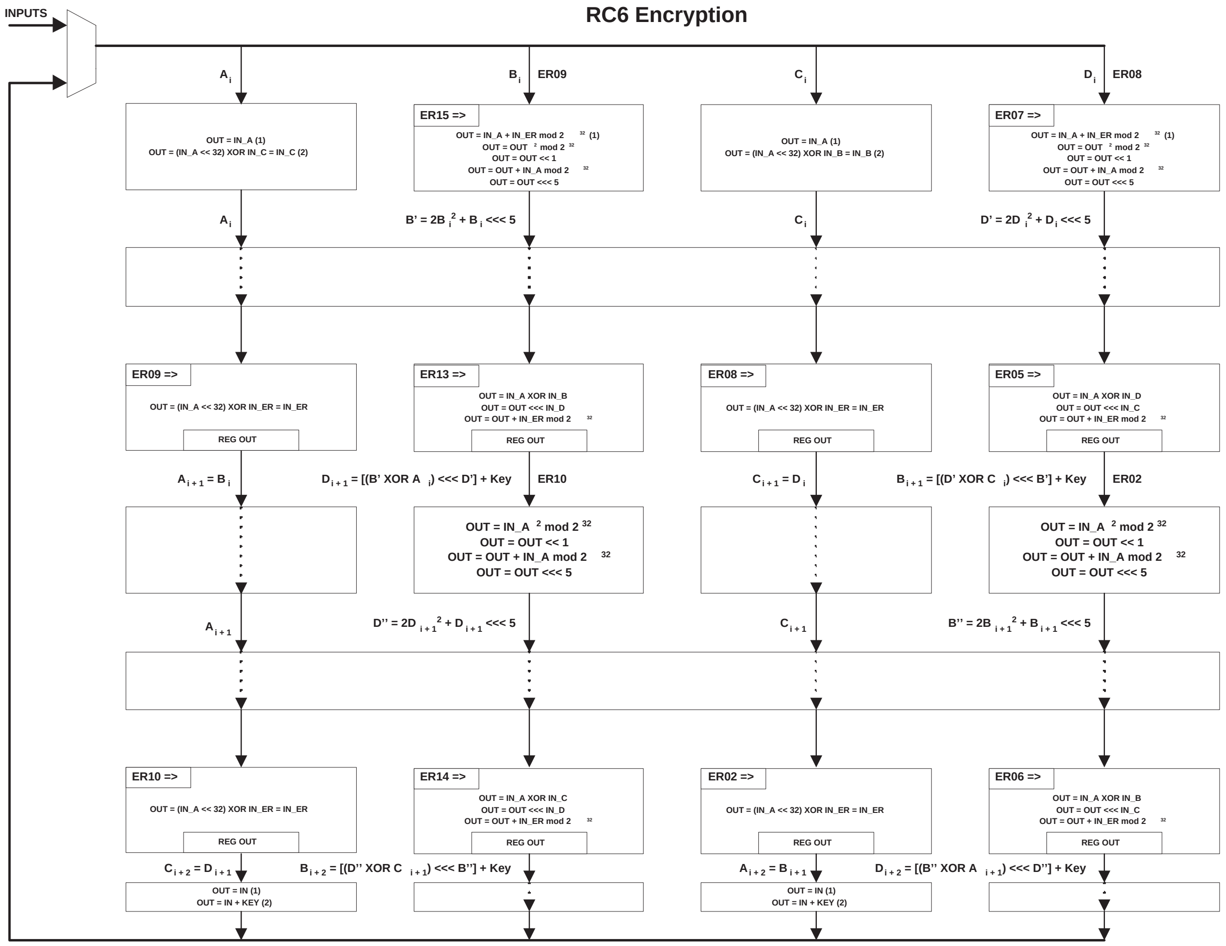

INPUTS
RC6 Encryption
A_i
B_i ER09
C_i
D_i ER08
OUT = IN_A (1)
OUT = (IN_A << 32) XOR IN_C = IN_C (2)
ER15 =>
OUT = IN_A + IN_ER mod 2^{32} (1)
OUT = OUT^2 mod 2^{32}
OUT = OUT << 1
OUT = OUT + IN_A mod 2^{32}
OUT = OUT <<< 5
OUT = IN_A (1)
OUT = (IN_A << 32) XOR IN_B = IN_B (2)
ER07 =>
OUT = IN_A + IN_ER mod 2^{32} (1)
OUT = OUT^2 mod 2^{32}
OUT = OUT << 1
OUT = OUT + IN_A mod 2^{32}
OUT = OUT <<< 5
A_i
$B' = 2B_i^2 + B_i$ <<< 5
C_i
$D' = 2D_i^2 + D_i$ <<< 5
ER09 =>
OUT = (IN_A << 32) XOR IN_ER = IN_ER
REG OUT
ER13 =>
OUT = IN_A XOR IN_B
OUT = OUT <<< IN_D
OUT = OUT + IN_ER mod 2^{32}
REG OUT
ER08 =>
OUT = (IN_A << 32) XOR IN_ER = IN_ER
REG OUT
ER05 =>
OUT = IN_A XOR IN_D
OUT = OUT <<< IN_C
OUT = OUT + IN_ER mod 2^{32}
REG OUT
$A_{i+1} = B_i$
$D_{i+1} = [(B'$ XOR $A_i)$ <<< D'] + Key
ER10
$C_{i+1} = D_i$
$B_{i+1} = [(D'$ XOR $C_i)$ <<< B'] + Key
ER02
OUT = IN_A^2 mod 2^{32}
OUT = OUT << 1
OUT = OUT + IN_A mod 2^{32}
OUT = OUT <<< 5
OUT = IN_A^2 mod 2^{32}
OUT = OUT << 1
OUT = OUT + IN_A mod 2^{32}
OUT = OUT <<< 5
A_{i+1}
$D'' = 2D_{i+1}^2 + D_{i+1}$ <<< 5
C_{i+1}
$B'' = 2B_{i+1}^2 + B_{i+1}$ <<< 5
ER10 =>
OUT = (IN_A << 32) XOR IN_ER = IN_ER
REG OUT
ER14 =>
OUT = IN_A XOR IN_C
OUT = OUT <<< IN_D
OUT = OUT + IN_ER mod 2^{32}
REG OUT
ER02 =>
OUT = (IN_A << 32) XOR IN_ER = IN_ER
REG OUT
ER06 =>
OUT = IN_A XOR IN_B
OUT = OUT <<< IN_C
OUT = OUT + IN_ER mod 2^{32}
REG OUT
$C_{i+2} = D_{i+1}$
$B_{i+2} = [(D''$ XOR $C_{i+1})$ <<< B''] + Key
$A_{i+2} = B_{i+1}$
$D_{i+2} = [(B''$ XOR $A_{i+1})$ <<< D''] + Key
OUT = IN (1)
OUT = IN + KEY (2)
OUT = IN (1)
OUT = IN + KEY (2)

Figure 8.3: RC6 encryption — COBRA mapping

Table 8.3 and Figure 8.4 detail the performance comparison of commercial hardware, software, and COBRA implementations of RC6. As shown in Figure 8.3, up to two rounds of RC6 may be mapped to the COBRA architecture in its current form. Data for implementations with more than two rounds assumes implementation within an architecture expanded by increasing both the iRAM address space and the number of rows, crossbar switches, and eRAMs with corresponding additions to the instruction set to support configuration of the new hardware elements. Table 8.4 details the additional hardware required for RC6 implementations assuming that for each two rows added to the hardware a crossbar switch will be added as well, regardless of whether or not it is required by the algorithm.

The COBRA implementation of RC6 significantly outperforms software implementations as the number of rounds implemented increases. Because RC6 requires that the old values of B and D be stored and recalled as the new values of A and C, respectively, the four 32-bit words output from each round must be registered to maintain synchronization of the data. The cycle counts shown in Figure 8.4 assume a pipelined implementation operating on multiple blocks of data where the round function is used as the atomic unit of the pipeline. This type of implementation does not achieve similar performance numbers in the case of a mode of operation that requires feedback, such as CBC, CFB, or OFB modes. For these modes of operation, the performance of a COBRA implementation will be constant, resulting in performance data roughly equivalent to the best software results.

RC6 Rounds	Software Implementations Cycles Per Encrypted Block	COBRA Pipelined Implementations Cycles Per Encrypted Block	Commercial Hardware Implementations Cycles Per Encrypted Block
1	223	290	20
2	223	145	10
4	223	75	5
5	223	60	4
10	223	30	2
20	223	4	1

Table 8.3: RC6 encryption — performance comparison

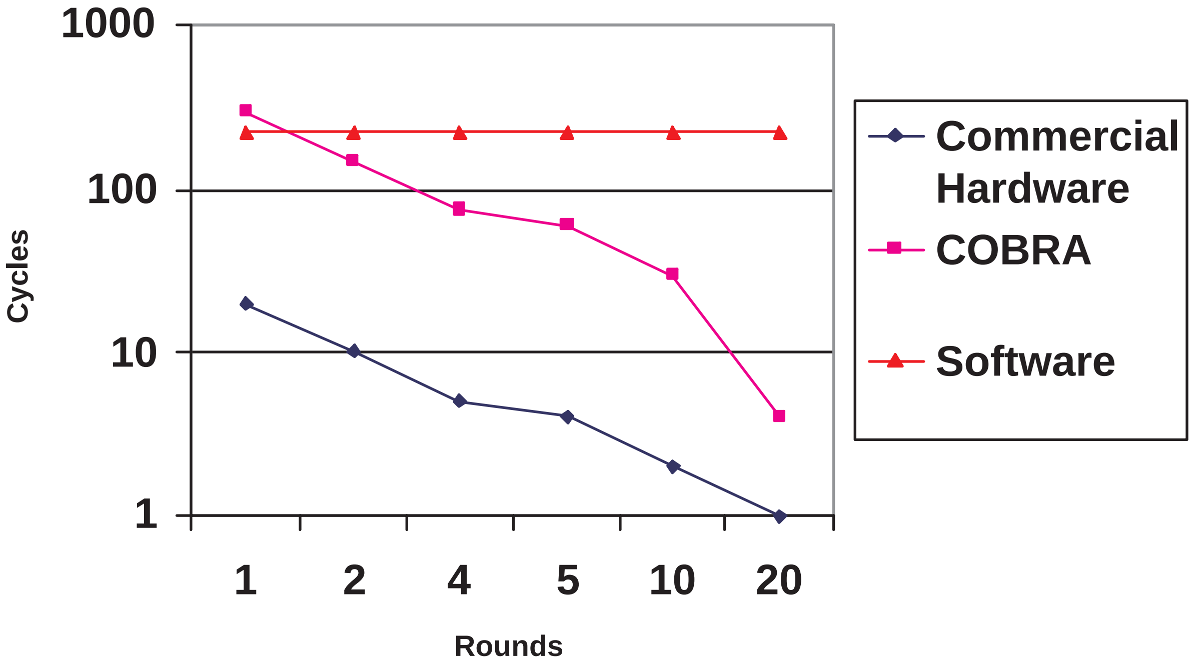


Figure 8.4: RC6 encryption — performance comparison

RC6 Rounds	**Rows**	**Crossbar Switches**	**eRAMs**	**iRAM Address Space**
1	0	0	0	12 bits
2	0	0	0	12 bits
4	4	2	2	12 bits
5	6	3	6	12 bits
10	16	8	26	12 bits
20	36	18	66	12 bits

Table 8.4: Additional COBRA hardware required for RC6 implementations

8.2.2 Rijndael

As detailed in Chapter 4, Rijndael is specified as a family of algorithms with three parameters, the number of columns in the state array *Nb*, the number of columns of the master key *Nk*, and the number of rounds *Nr*. For the COBRA implementation, we consider *Nb = 4*, *Nk = 4*, and *Nr = 10*, which conforms to the AES specifications with a key size of 128 bits.

Rijndael is easily partitioned into two stages — key scheduling and encryption. The COBRA implementation utilizes the generic flag bits in the flag register to indicate to the external system when the *Rcon[1:10]* parameters and the master key must be provided. Note that the master key is loaded from least significant word to most significant word but in byte reversed order. The *Rcon[1:10]* parameters are stored in eRAM 13 and the master key is stored in eRAM 14. Key scheduling then computes the *W[i]* array as follows:

$$\begin{aligned} W_i &= W_{i-4} \oplus (S[W_{i-1} \lll 8] \oplus Rcon[i/4]) && \text{for (i mod 4)} = 0 \\ W_i &= W_{i-4} \oplus W_{i-1} && \text{for (i mod 4)} \neq 0 \end{aligned}$$

RCE 00 is initially configured to load from eRAM 14 to pass the master key into the eRAMs which will contain the final values for the subkeys. Subkeys are stored in eRAMs 01, 02, 05, 06, 09, 10, 14, and 15. The choice of eRAMs for the storage of subkeys was made to facilitate the encryption process. RCEs 10 and 20 are initially configured as pass-through elements and RCE 30 is configured as a registered pass-through element. Once *W[0:3]* have been passed to the appropriate eRAMs, RCE 00 is reconfigured to calculate W_i where i is initialized to the value of *Nk* and has a maximum value of $[Nb\ *\ (Nr\ +\ 1)]\ -\ 1$ as per the Rijndael specification [37]. RCE 10 is configured to XOR the intermediate value of W_i, termed *W'*, with W_{i-4} to generate the final value for W_i. RCE MULs 31 and 33 and RCE 32 are configured to shift out their primary inputs and output the value of W_i so that the appropriate eRAMs may be loaded. Figure 8.5 details the COBRA architecture mapping of Rijndael key scheduling.

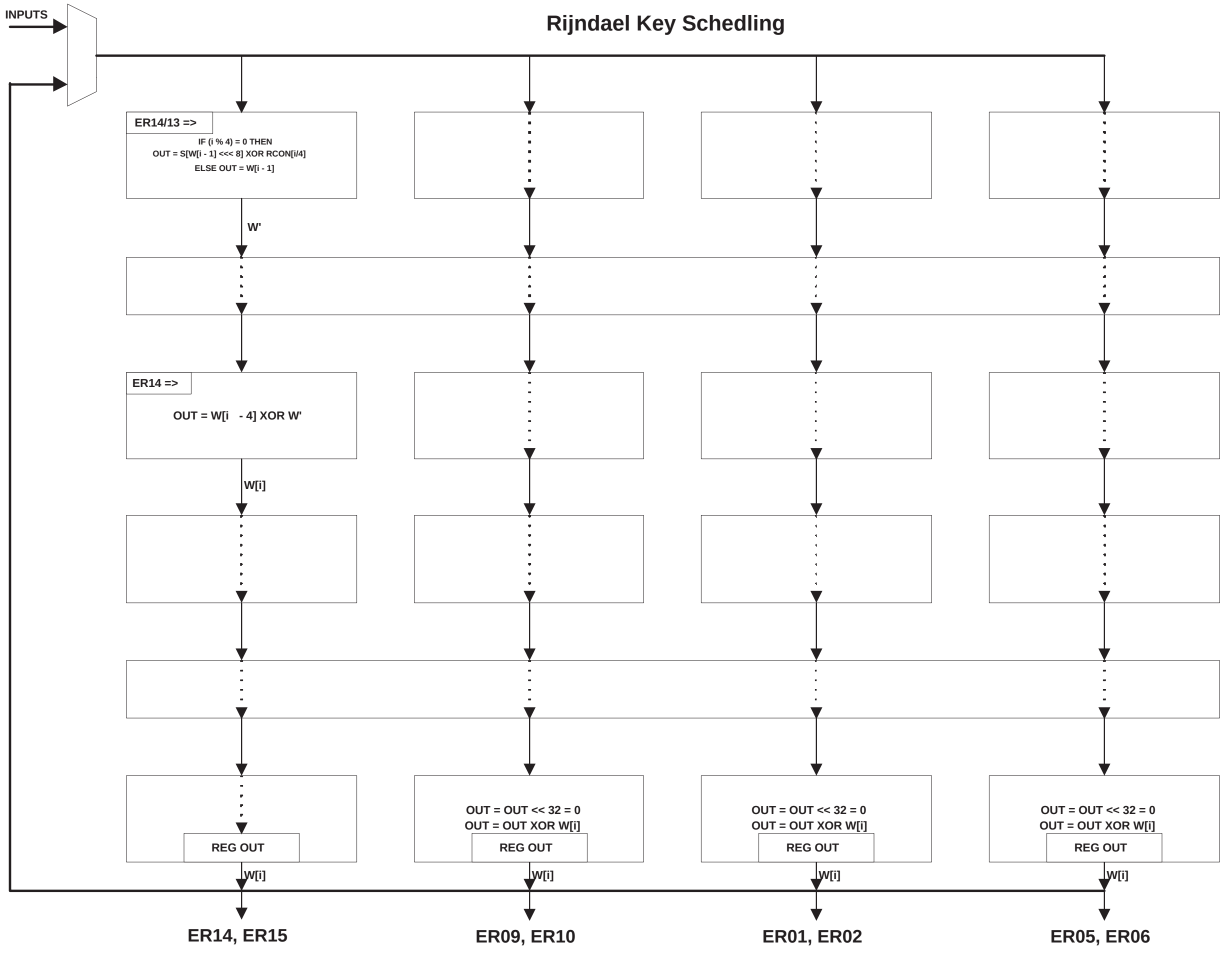


Figure 8.5: Rijndael key scheduling — COBRA mapping

At the start of encryption, key whitening is performed by XORing the plaintext block with the first four subkeys located in eRAMs 15, 10, 02, and 06, which are the embedded RAM inputs to RCE 00, RCE MUL 01, RCE 02, and RCE MUL 03, respectively. The RCEs and RCE MULs in row zero are also configured to perform the ByteSub operation through the use of the 8-bit to 8-bit look-up-tables. The ShiftRow operation is implemented within crossbar switch one and the RCEs and RCE MULs in row one perform both the MixColumn Galois Field fixed field constant multiplication and the XORing of the result with the associated round's subkeys. Subkeys are located in eRAMs 14, 09, 01, and 05, which are the embedded RAM inputs to RCE 10, RCE MUL 11, RCE 12, and RCE MUL 13, respectively. The RCEs and RCE MULs in row two, crossbar switch two, and the whitening elements are configured as pass-through elements. The RCEs and RCE MULs in row three are configured as registered pass-through elements.

After the first round has been completed, the RCEs and RCE MULs in row zero are reconfigured to remove the key whitening operation. Encryption iterates over the round function a total of ten times. For the final round, the RCEs and RCE MULs in row one are reconfigured to remove the Galois Field fixed field constant multiplication as per the Rijndael specification [37]. Figure 8.6 details the implementation of a single round of Rijndael. A total of two rounds may be implemented in the COBRA architecture. However, even with additional hardware to support a larger datapath, more than three rounds cannot be implemented due to limitations on the size of the iRAM. Each additional Rijndael round would require configuration of another set of 8-bit to 8-bit look-up-tables in support of the ByteSub operation as well as another set of Galois Field fixed field constant multipliers in support of the MixColumn operation. The additional instructions required for these configuration operations would result in the microcode exceeding the 12-bit address space of the iRAM for an implementation of more than three rounds.

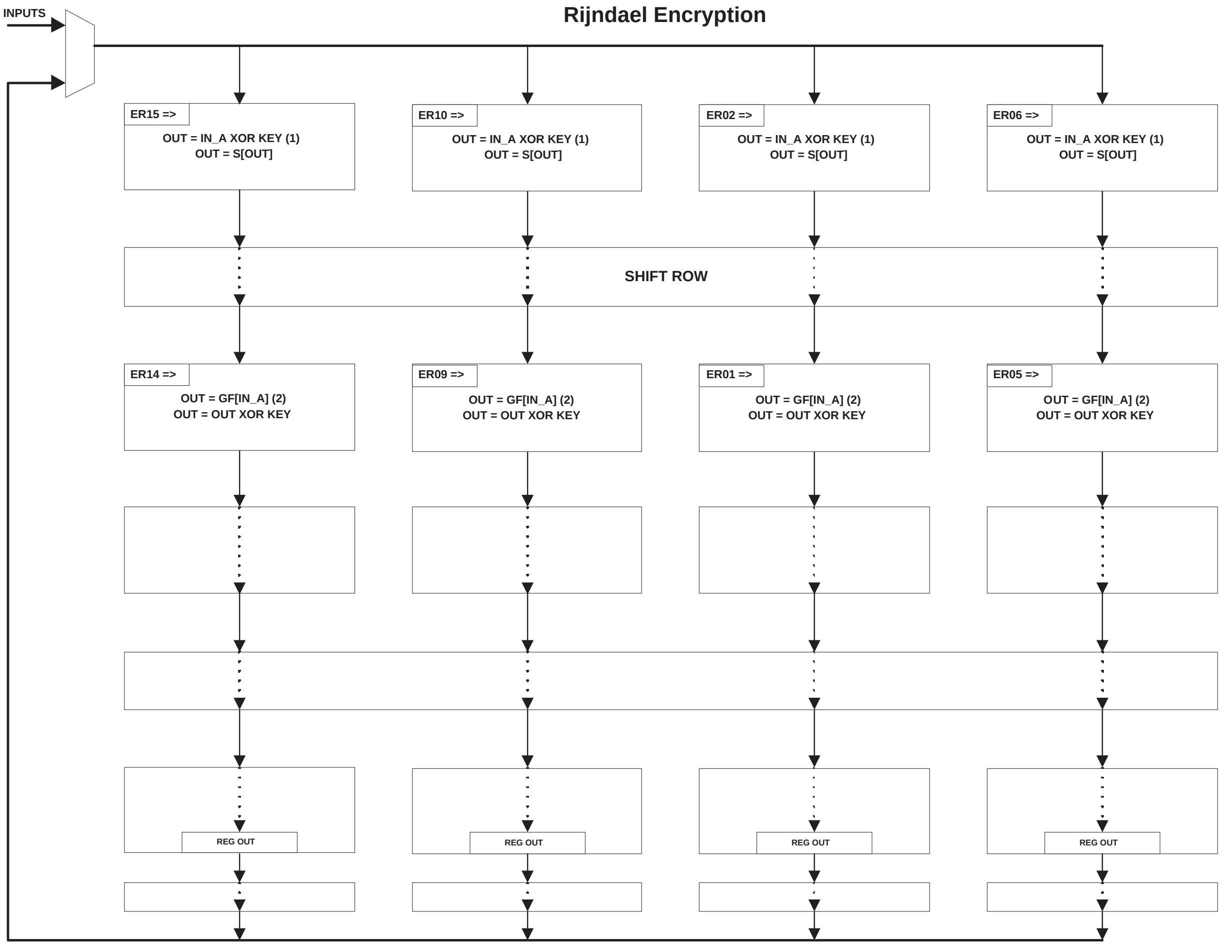


Figure 8.6: Rijndael encryption — COBRA mapping

Table 8.5 and Figure 8.7 detail the performance comparison of commercial hardware, software, and COBRA implementations of Rijndael. As shown in Figure 8.6, a single round of Rijndael may be mapped to the COBRA architecture in its current form. Data for implementations with more than one round assumes implementation within an architecture expanded by increasing both the iRAM address space and the number of rows, crossbar switches, and eRAMs with corresponding additions to the instruction set to support configuration of the new hardware elements. Table 8.6 details the additional hardware required for Rijndael implementations assuming that for each two rows added to the hardware a crossbar switch will be added as well, regardless of whether or not it is required by the algorithm.

The COBRA implementation of Rijndael significantly outperforms software implementations as the number of rounds implemented increases to match the total number of rounds in the algorithm, with pipelined implementations outperforming loop unrolled implementations. The encryption code size is dominated by the instructions required for reconfiguration of the RCEs and RCE MULs to enable and disable the key whitening and Galois Field fixed field constant multiplication operations. This accounts for the small performance increases seen when increasing the number of rounds implemented from one to two or to five. When all ten rounds of the algorithm are implemented within the COBRA architecture, the instructions required for the aforementioned reconfiguration are eliminated, resulting in a significant decrease in the code size which translates to a greatly reduced cycle count for the implementation, as shown in Figure 8.7.

		COBRA	**COBRA**	**Commercial**
	Software	**Loop Unrolled**	**Pipelined**	**Hardware**
	Implementations	**Implementations**	**Implementations**	**Implementations**
Rindael	**Cycles Per**	**Cycles Per**	**Cycles Per**	**Cycles Per**
Rounds	**Encrypted Block**	**Encrypted Block**	**Encrypted Block**	**Encrypted Block**
1	124	114	114	10
2	124	84	44	5
5	124	72	44	2
10	124	6	18	1

Table 8.5: Rijndael encryption — performance comparison

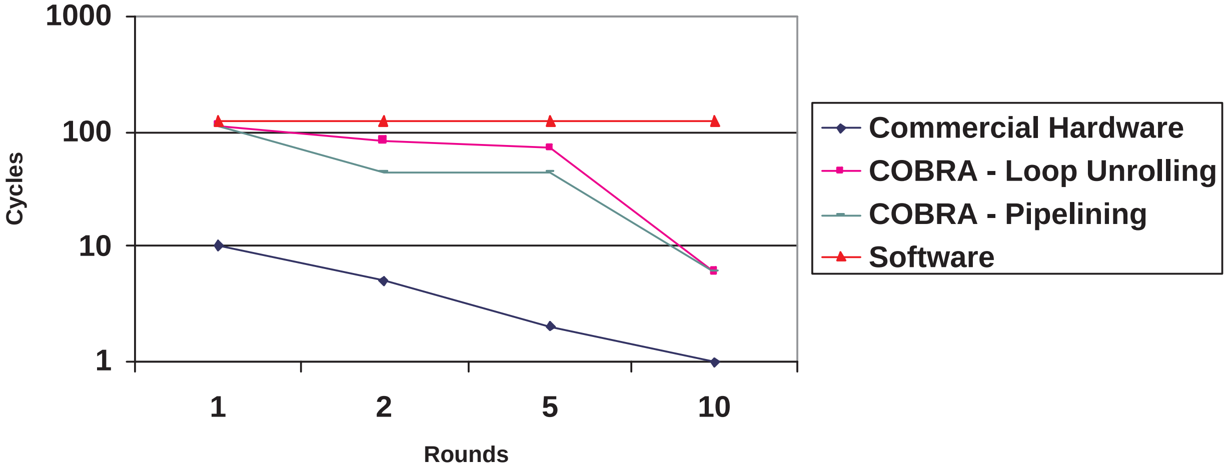


Figure 8.7: Rijndael encryption — performance comparison

Rijndael Rounds	**Rows**	**Crossbar Switches**	**eRAMs**	**iRAM Address Space**
1	0	0	0	12 bits
2	0	0	0	12 bits
5	6	3	6	13 bits
10	16	8	26	14 bits

Table 8.6: Additional COBRA hardware required for Rijndael implementations

8.2.3 Serpent

As detailed in Chapter 4, Serpent is a thirty two round Substitution-Permutation network. Serpent is easily partitioned into three stages — stage one key scheduling, stage two key scheduling, and encryption. The COBRA implementation utilizes the generic flag bits in the flag register to indicate to the external system when the padded master key and ϕ must be provided. The padded master key is comprised of eight 32-bit blocks, denoted as w_{-8} through w_{-1}, which are stored in eRAMs 13, 14, 15, and 16. The value *0x00000001* is stored in eRAM 12 for use in stage one key scheduling as an increment value and ϕ is stored in eRAM 08. Stage one key scheduling then computes the w_i array as follows:

$$w_i \quad = \quad (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11$$

RCE MUL 01 is initially configured to output zero and then reconfigured to increment the feedback value i from RCE MUL 31 using the increment value stored in eRAM 12. RCE 02 is configured to load ϕ from eRAM 08 while RCE 00 is configured to load w_{i-8} from eRAM 16. The output of RCE 00 is passed to RCE 10 and XORed with w_{i-5}, loaded from eRAM 15. The output of RCE 10 is passed to RCE 20 and XORed with w_{i-3}, loaded from eRAM 14. The output of RCE 20 is passed to RCE 30 and XORed with w_{i-1}, loaded from eRAM 13, ϕ, and i. The output of RCE 30 is registered and stored in eRAMs 13, 14, 15, and 16 for use in later iterations. Figure 8.8 details the COBRA architecture mapping of Serpent stage one key scheduling.

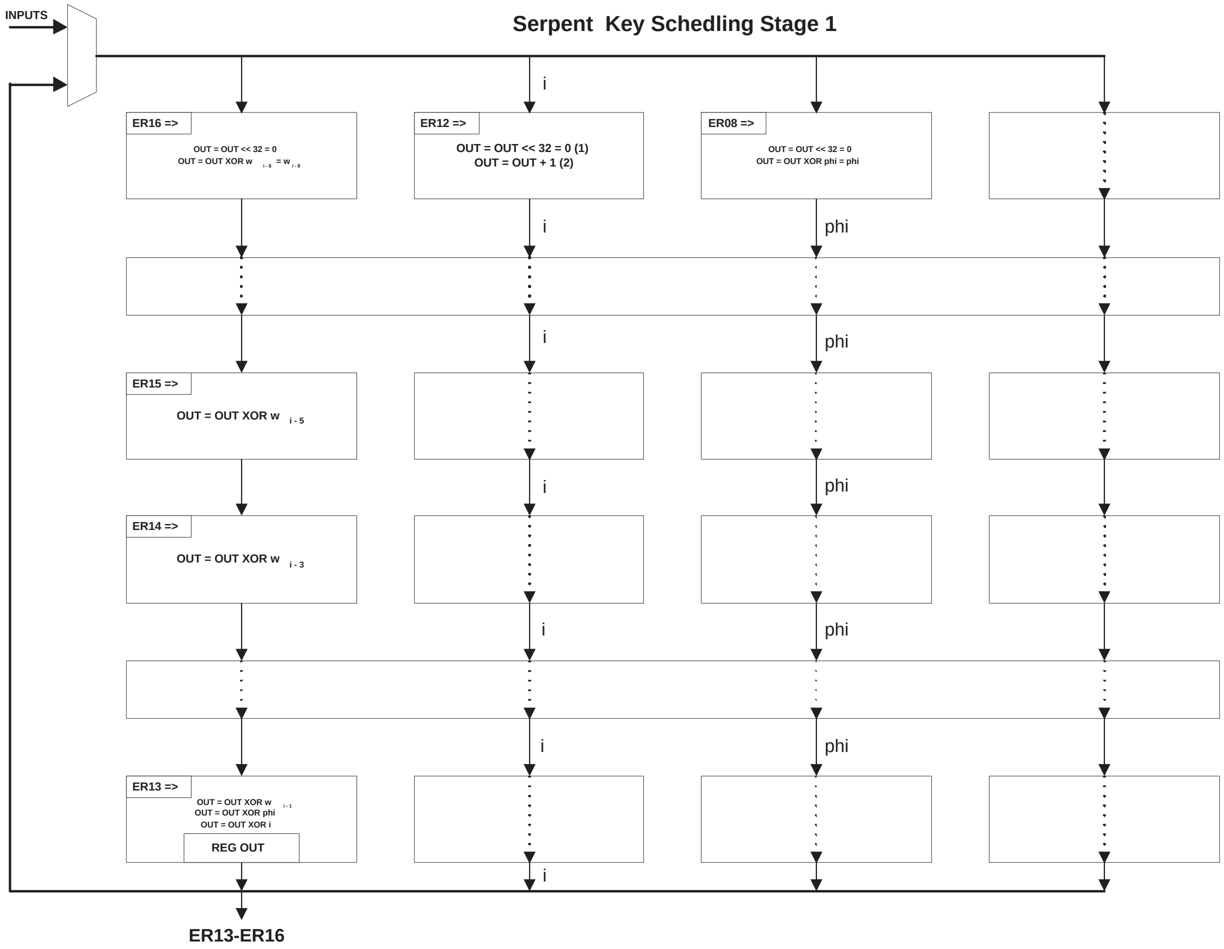


Figure 8.8: Serpent stage one key scheduling — COBRA mapping

Serpent stage two key scheduling passes the w_i array through the S-Boxes as per the Serpent specification [13]. RCE 00 is configured to output w_i which is passed through the appropriate S-Box in RCE 10. Note that the C element in RCE 10 is configured to operate in 4-LUT paging mode to support the eight different Serpent S-Boxes. RCE 20 is configured as a pass-through element and RCE 30 is configured as a registered pass-through element, outputting w_i for storage in eRAMs 13, 14, 15, and 16. RCE 32 is also configured to output the registered w_i, in this case for storage in eRAMs 05, 06, 07, and 08. The choice of eRAMs for the storage of subkeys was made to facilitate the encryption process. Figure 8.9 details the COBRA architecture mapping of Serpent stage two key scheduling.

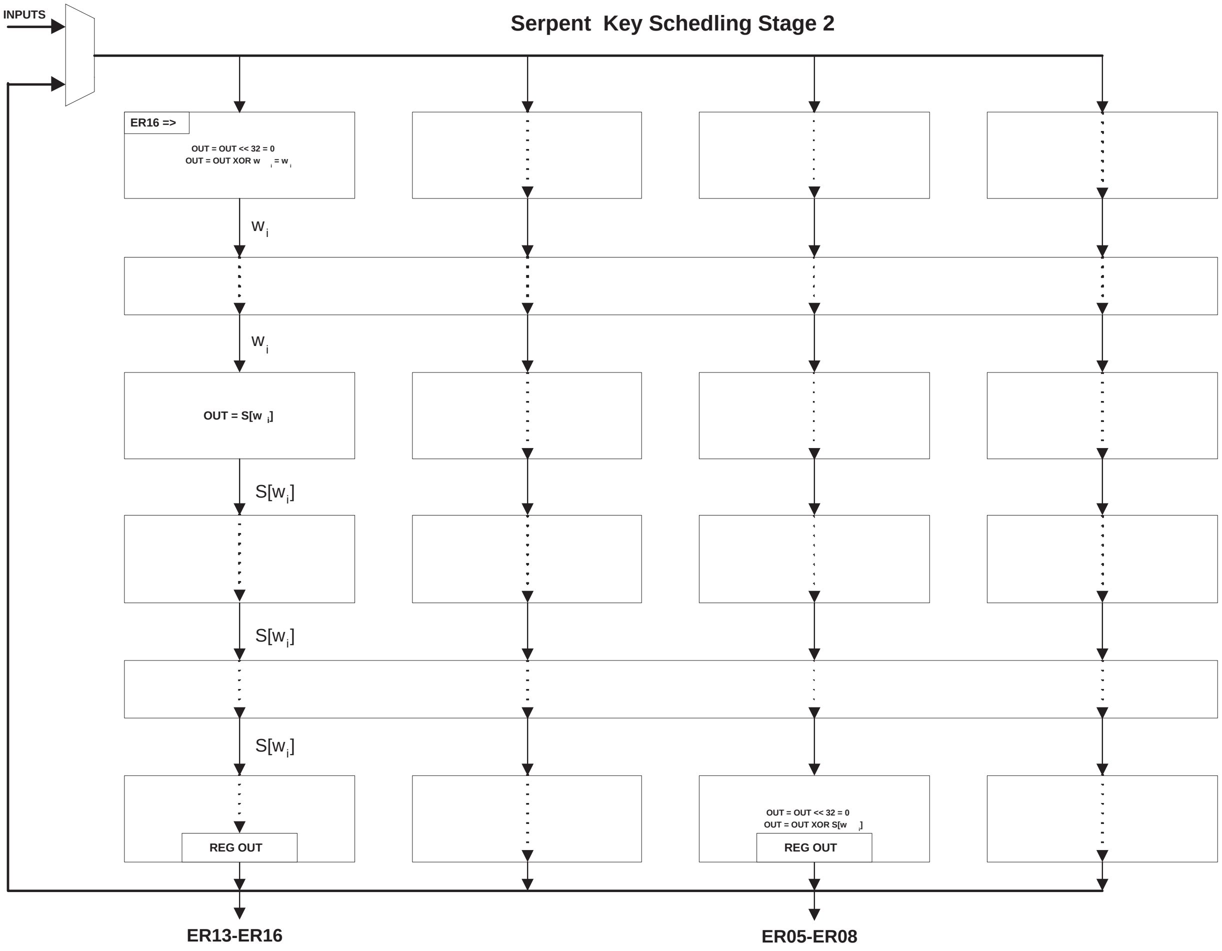


Figure 8.9: Serpent stage two key scheduling — COBRA mapping

During encryption, RCE 00, RCE MUL 01, RCE 02, and RCE MUL 03 are configured to XOR the plaintext with the subkeys associated with the current round before passing the value through the appropriate S-Boxes. Note that the *C* element in the RCEs and the *C1* element in the RCE MULs are configured to operate in 4-LUT paging mode to support the eight different Serpent S-Boxes. RCE MULs 01 and 03 are also configured to begin the Serpent Linear Transformation (see Section 4.2.3 Figure 4.7 for a detailed description of the Serpent Linear Transformation). The Linear Transformation requires the use of all RCEs and RCE MULs in rows one, two, and three. For the final round of the cipher, the Linear Transformation is not performed. As a result, RCE MULs 01 and 03 are reconfigured to remove the start of the Linear Transformation. The RCEs and RCE MULs in row one are reconfigured to perform the final key XORing with the data and the RCEs and RCE MULs in row two are reconfigured as pass-through elements. Finally, the RCEs and RCE MULs in row three are reconfigured as registered pass-through elements. Figure 8.10 details the implementation of a single round of Serpent.

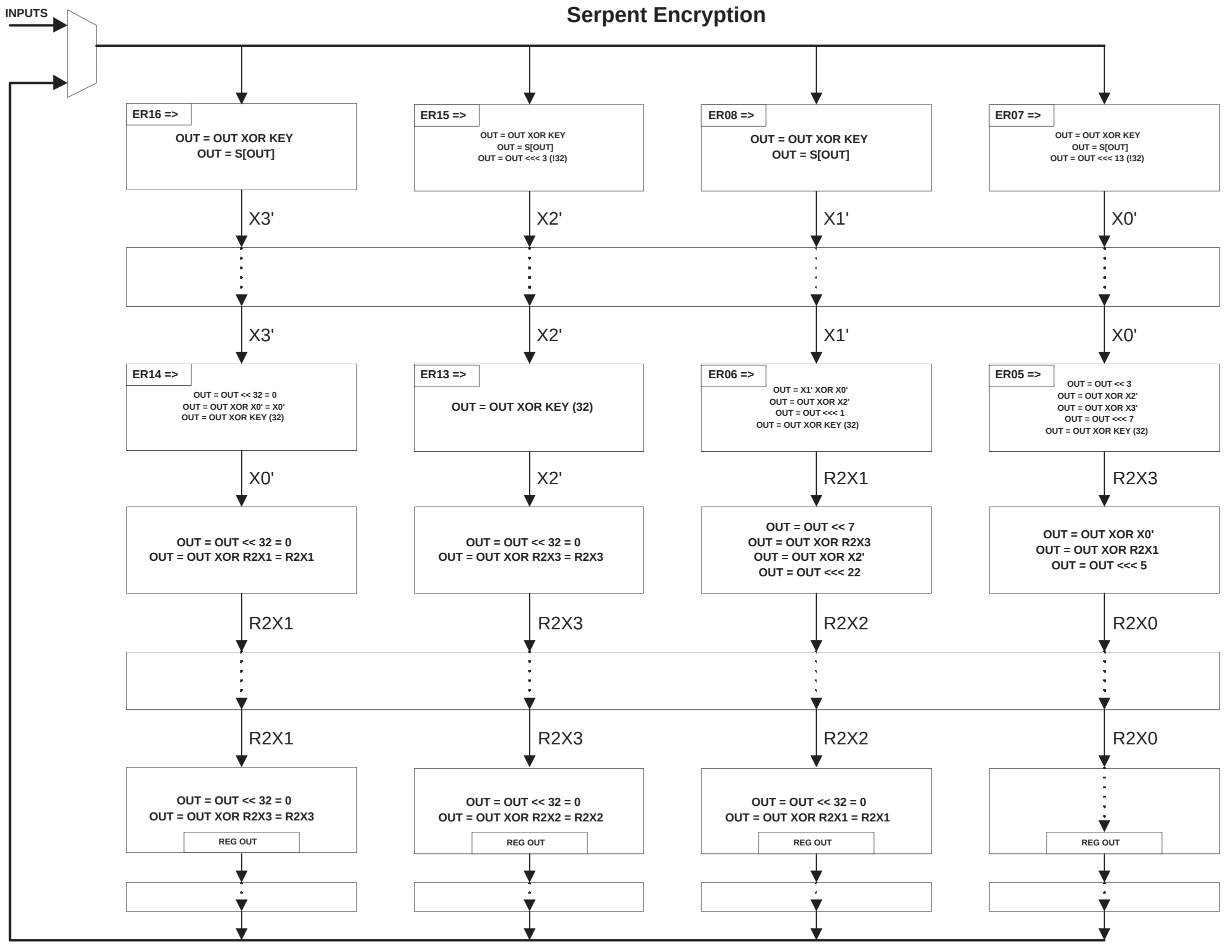


Figure 8.10: Serpent encryption — COBRA mapping

Table 8.7 and Figure 8.11 detail the performance comparison of commercial hardware, software, and COBRA implementations of Serpent. As shown in Figure 8.10, a single round of Serpent may be mapped to the COBRA architecture in its current form. Data for implementations with more than one round assumes implementation within an architecture expanded by increasing both the iRAM address space and the number of rows, crossbar switches, and eRAMs with corresponding additions to the instruction set to support configuration of the new hardware elements. Table 8.8 details the additional hardware required for Serpent implementations assuming that for each two rows added to the hardware a crossbar switch will be added as well, regardless of whether or not it is required by the algorithm.

The COBRA implementation of Serpent significantly outperforms software implementations as the number of rounds implemented increases with pipelined implementations matching or outperforming loop unrolled implementations. The encryption code size is dominated by the instructions required for reconfiguration of the RCEs and RCE MULs to enable and disable the Linear Transformation during the final round of the cipher. This accounts for the small performance increases seen when increasing the number of rounds implemented from eight to sixteen. Moreover, in the sixteen round pipelined implementation, the cycles required to output the sixteen results overshadows the performance gain achieved through operating on multiple blocks of data simultaneously. When all thirty two rounds of the algorithm are implemented within the COBRA architecture, the instructions required for the aforementioned reconfiguration are eliminated, resulting in a significant decrease in the code size which translates to a greatly reduced cycle count for the implementation, as shown in Figure 8.11.

Rindael Rounds	Software Implementations Cycles Per Encrypted Block	COBRA Loop Unrolled Implementations Cycles Per Encrypted Block	COBRA Pipelined Implementations Cycles Per Encrypted Block	Commercial Hardware Implementations Cycles Per Encrypted Block
1	565	546	546	32
8	565	140	70	4
16	565	112	112	2
32	565	7	7	1

Table 8.7: Serpent encryption cycle counts — COBRA implementations

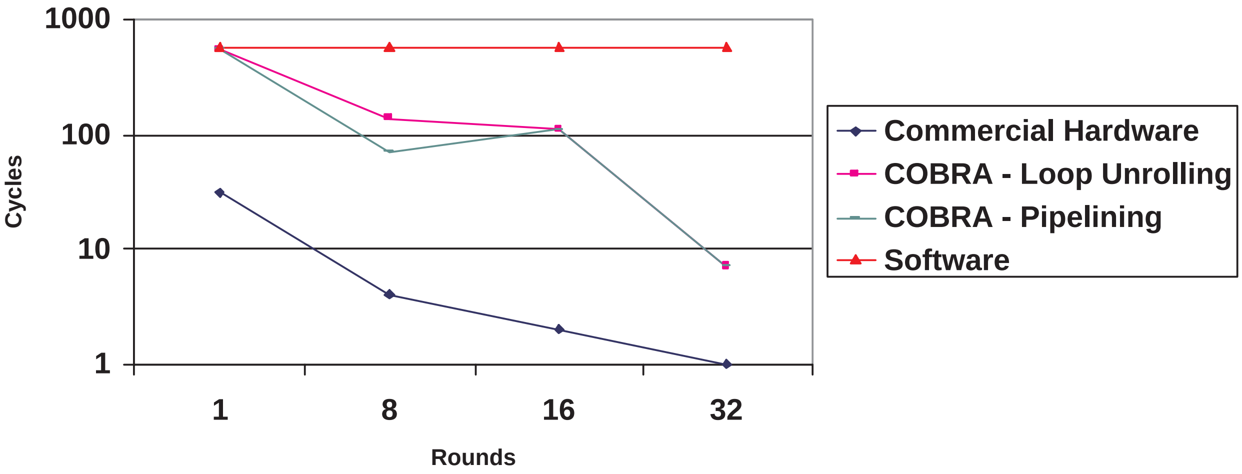


Figure 8.11: Serpent encryption — performance comparison

Serpent Rounds	Rows	Crossbar Switches	eRAMs	iRAM Address Space
1	0	0	0	12 bits
8	28	14	18	13 bits
16	60	30	50	14 bits
32	124	62	114	15 bits

Table 8.8: Additional COBRA hardware required for Serpent implementations

8.3 Hardware Resource Requirements

The VHDL used to implement the COBRA architecture was synthesized using Exemplar Logic's LeonardoSpectrum Level 3 version v2001_1d.46 targeting the ADK TSMC 0.35 micron library. Table 8.9 summarizes the gate counts for each configurable element within a COBRA RCE or RCE MUL. From Table 8.9 it is clear that the dominant element within an RCE or RCE MUL in terms of area is the *C* element. Four 128 × 4 look-up-tables and four 256 × 8 look-up-tables are implemented within the *C* element, resulting in a total of 10,240 storage bits.

Configurable Element	**Gates**
A	172
B	1,012
C	98,624
D	5,243
E	887
F	10,606
4-to-1 Multiplexor, Grouping of 32	160
4-to-1 Multiplexor, Grouping of 5	26
2-to-1 Multiplexor, Grouping of 32	83
32-Bit Register with Enable	267

Table 8.9: Reconfigurable element gate counts

Table 8.10 summarizes the gate counts for an RCE and an RCE MUL broken down by reconfigurable element grouping while Table 8.11 summarizes the gate counts for the COBRA architecture primary sub-blocks. In the case of Table 8.10, a reconfigurable element grouping represents the total gate count for all reconfigurable elements of that type. Once again, the *C* element clearly dominates the gate count requirements for both an RCE and an RCE MUL. The gate count listed for the crossbar switches, input multiplexors, whitening blocks, and embedded RAMs in Table 8.11 is the total gate count for all elements of that type. Table 8.11 indicates a total gate count of just over 6.6 Million gates for the entire

COBRA architecture.

Configurable Element	**RCE Gates**	**RCE MUL Gates**
A	516	516
B	2,024	2,024
C	98,624	197,248
D	0	5,243
E	2,661	2,662
F	10,606	10,606
4-to-1 Multiplexor, Grouping of 32	800	960
4-to-1 Multiplexor, Grouping of 5	26	26
2-to-1 Multiplexor, Grouping of 32	166	249
32-Bit Register	267	267
Overhead	510	605
Total	**116,200**	**220,405**

Table 8.10: RCE and RCE MUL gate counts

Element	**Gates**
RCE/RCE MUL Array	2,692,840
Crossbar Switches	8,556
Input Multiplexors	332
Whitening Blocks	3,128
Embedded RAMs	1,210,640
Instruction RAM	2,773,184
Datapath Overhead	2,464
Chip Overhead	370
Total	**6,691,514**

Table 8.11: COBRA architecture gate counts

Based on the data presented in Sections 8.2, hardware resource requirements can be extrapolated for each of the targeted block ciphers based on the number of rounds implemented. Tables 8.12, 8.13, and 8.14 together with Figures 8.12, 8.13, and 8.14 detail the extrapolated gate counts based on the number of rounds implemented for RC6, Rijndael, and Serpent, respectively. Extrapolated data is based on the additional COBRA hardware required to implement the given algorithms detailed in Tables 8.4, 8.6, and 8.8.

RC6 Rounds	Gates Per Implementation
1	6,691,514
2	6,691,514
4	9,544,240
5	11,197,598
10	19,464,388
20	35,997,968

Table 8.12: RC6 gate count extrapolation

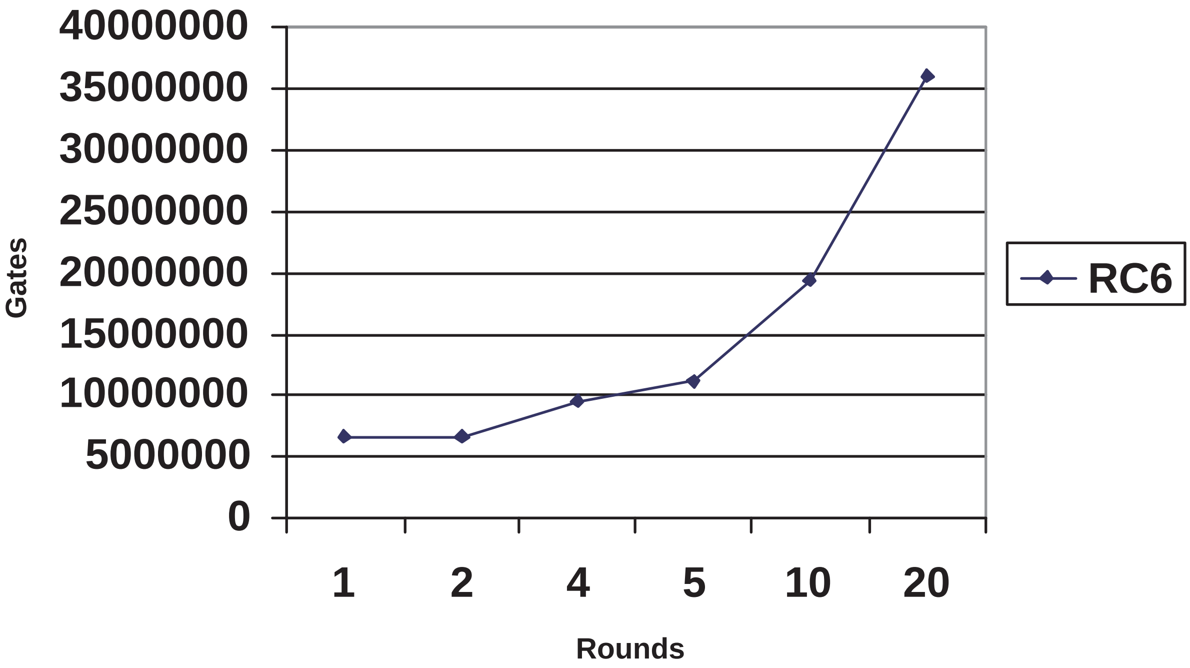


Figure 8.12: RC6 gate count extrapolation

Rijndael Rounds	Gates Per Implementation
1	6,691,514
2	6,691,514
5	13,970,782
10	27,783,940

Table 8.13: Rijndael gate count extrapolation

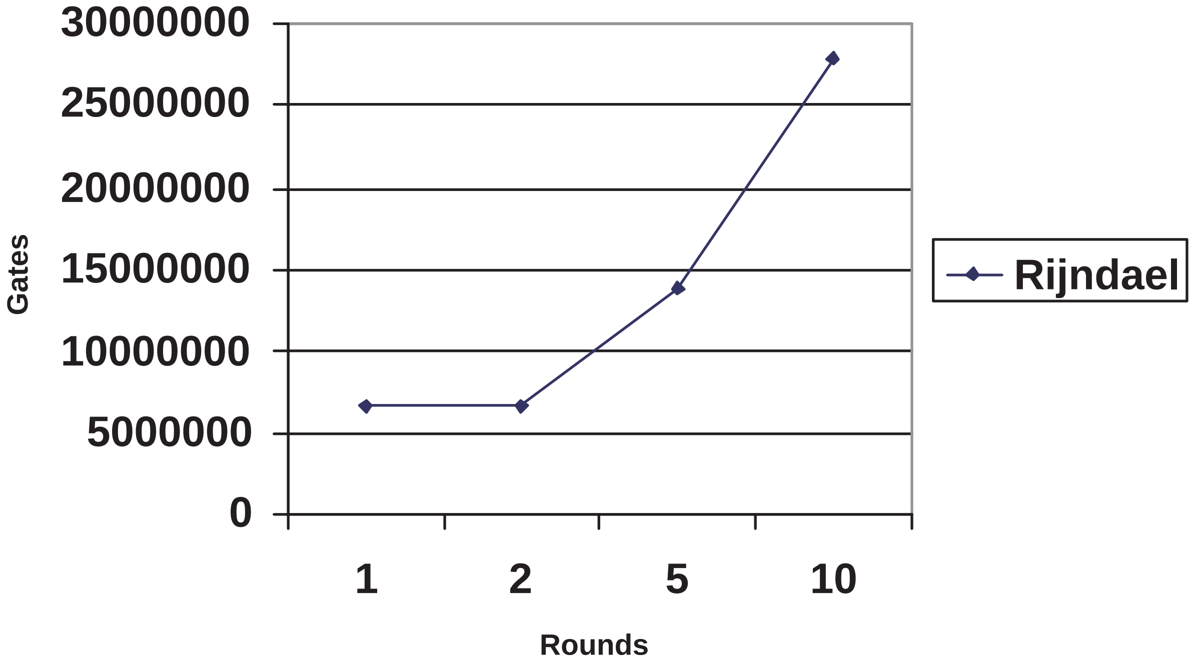


Figure 8.13: Rijndael gate count extrapolation

Serpent Rounds	Gates Per Implementation
1	6,691,514
8	29,736,440
16	59,315,256
32	118,472,88

Table 8.14: Serpent gate count extrapolation

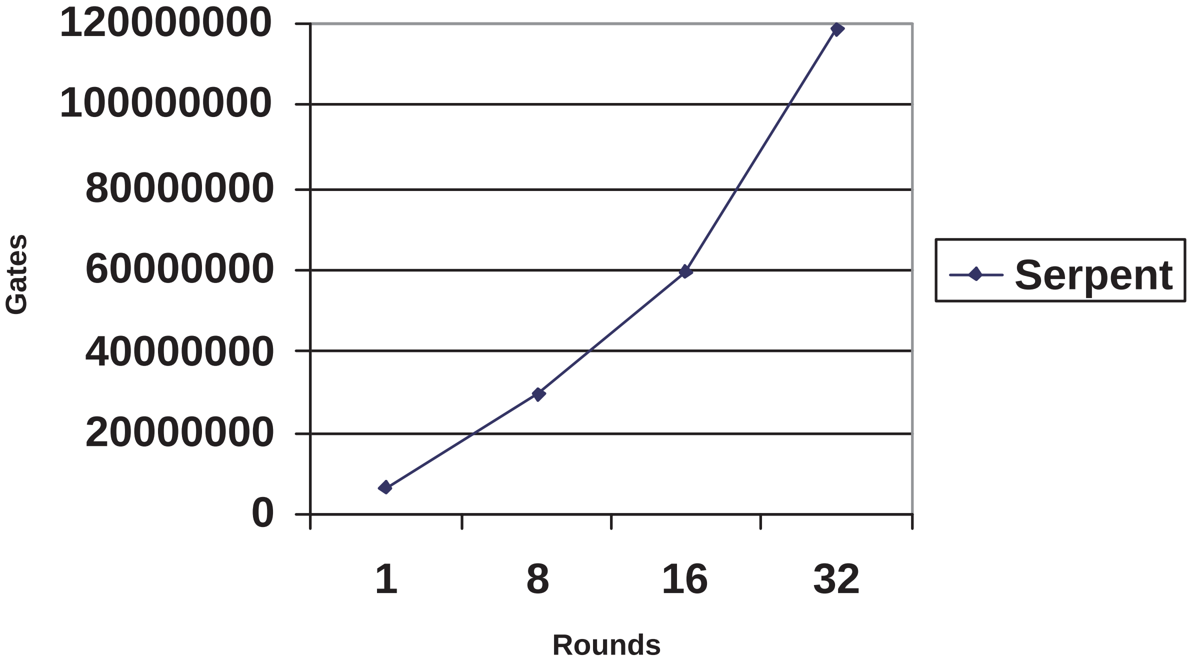


Figure 8.14: Serpent gate count extrapolation

From Figures 8.12, 8.13, and 8.14, a cycle-gates (CG) product similar to the classical time-area (TA) product may be calculated. The normalized CG products for the RC6, Rijndael, and Serpent implementations are shown in Tables 8.15, 8.16, and 8.17 together with Figures 8.15, 8.16, and 8.17. Note that normalization was performed in respect to the largest CG product for each algorithm, resulting in a CG product of zero being the ideal and a CG product of one being the worst case.

Figures 8.12, 8.13, and 8.14 consistently demonstrate that pipelined implementations outperform loop unrolled implementations, although this performance gain only applies to non-feedback modes of operation. The Figures also indicate that the best CG product occurs when all rounds of a cipher can be implemented in the COBRA architecture. However, intermediate degrees of unrolling do not always result in an improved CG product, as evidenced by the five round implementations of Rijndael and the sixteen round implementations of Serpent. In each of these cases, the decrease in encryption cycles realized through loop unrolling is overshadowed by the increase in required hardware resources. In the case of RC6, a steady decrease of the CG product is evidenced due to the pipelined nature of the implementation. As discussed in Section 8.2, RC6 requires a pipelined implementation due to its effective 192-bit datapath and an unrolled implementation cannot be implemented for greater than one round.

RC6 Rounds	**CG Product**	**Normalized CG Product**
1	1,940,539,060	1.0000
2	970,269,530	0.5000
4	715,818,000	0.3689
5	671,855,880	0.3462
10	583,931,640	0.3009
20	143,991,872	0.0742

Table 8.15: RC6 CG product — pipelined implementations

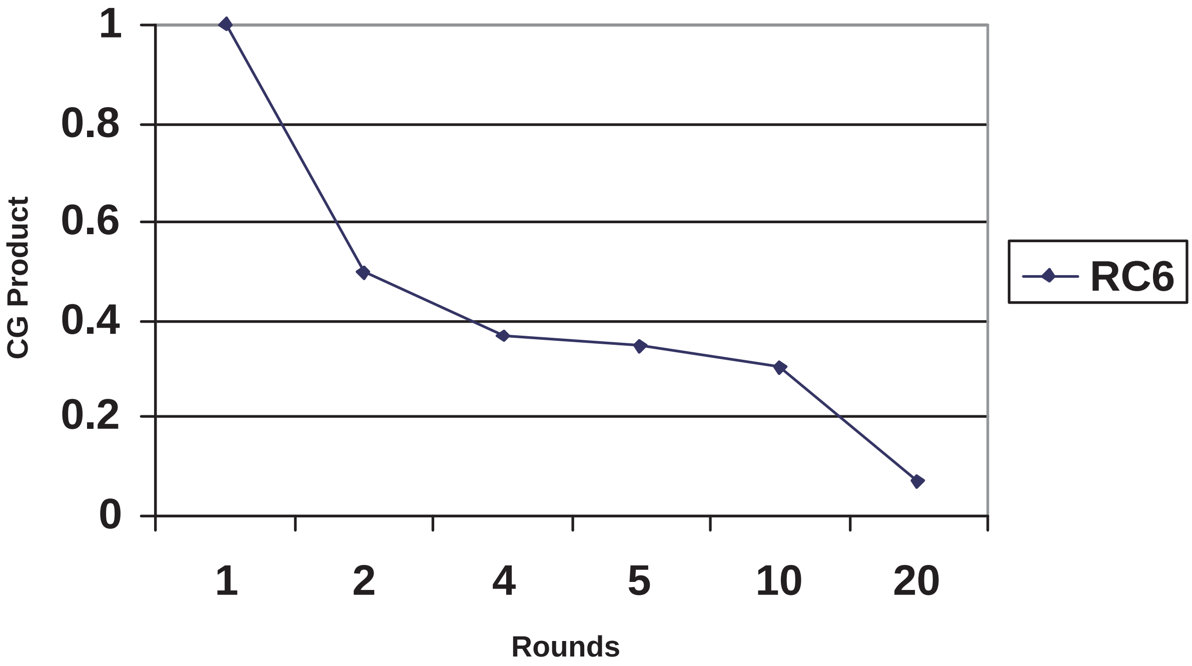


Figure 8.15: RC6 CG product — pipelined implementations

Rijndael Rounds	**Loop Unrolling CG Product**	**Normalized CG Product**	**Pipelining CG Product**	**Normalized CG Product**
1	762,832,596	0.7584	762,832,596	1.0000
2	562,087,176	0.5588	294,426,616	0.3860
5	1,005,896,304	1.0000	614,714,408	0.8058
10	166,703,640	0.1657	166,703,640	0.2185

Table 8.16: Rijndael CG product — loop unrolled and pipelined implementations

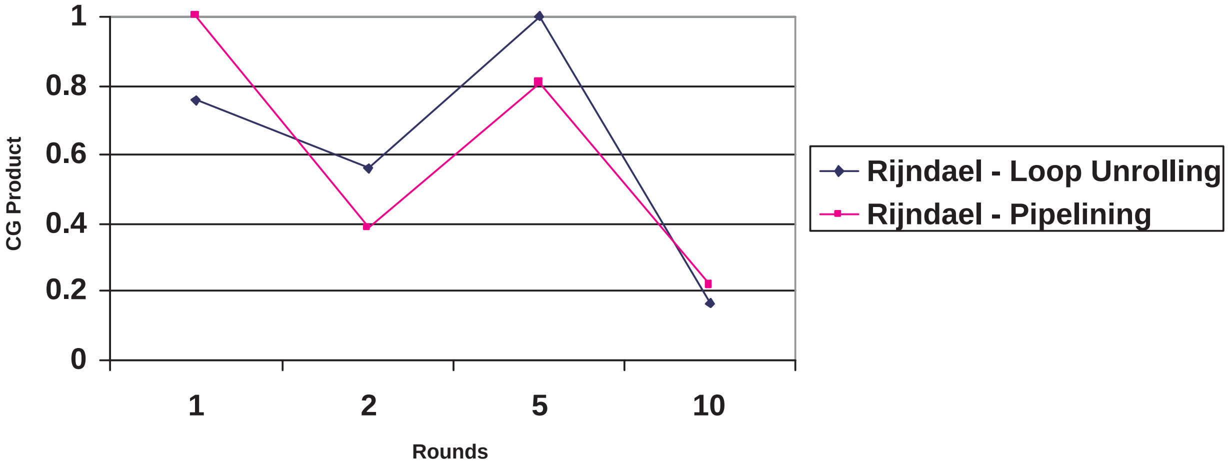


Figure 8.16: Rijndael CG product — loop unrolled and pipelined implementations

Serpent Rounds	**Loop Unrolling CG Product**	**Normalized CG Product**	**Pipelining CG Product**	**Normalized CG Product**
1	3,653,566,644	0.5500	3,653,566,644	0.5500
8	4,163,101,600	0.6267	2,081,550,800	0.3133
16	6,643,308,672	1.0000	6,643,308,672	1.0000
32	829,310,216	0.1248	829,310,216	0.1248

Table 8.17: Serpent CG product — loop unrolled and pipelined implementations

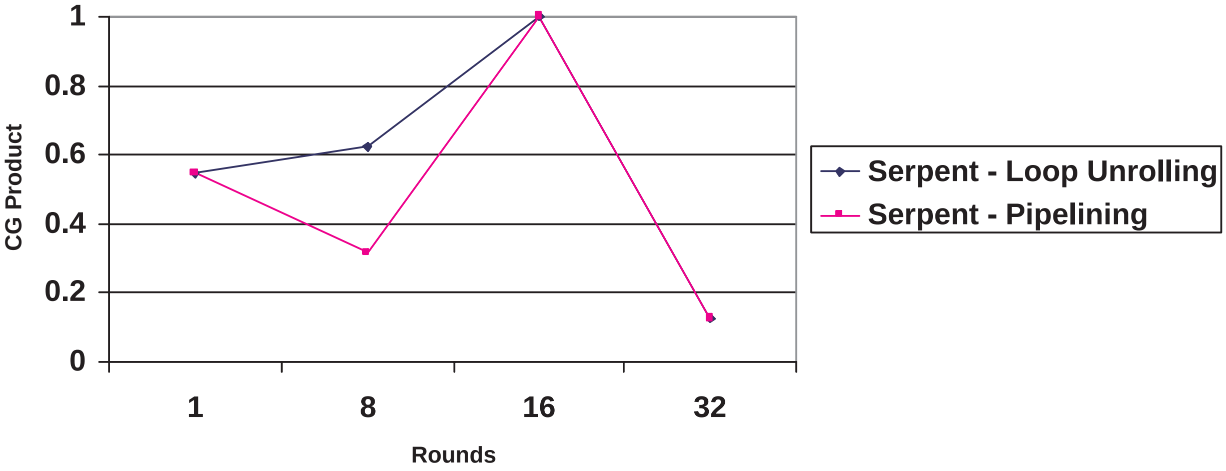


Figure 8.17: Serpent CG product — loop unrolled and pipelined implementations

Chapter 9

Conclusions and Future Work

9.1 Summary

The COBRA architecture has been presented as an innovative implementation option for block ciphers. While other implementation options such as software, commercial FPGAs, custom ASICs, reconfigurable processors, and specialized reconfigurable architectures exist, none of these choices were found to satisfy the requirements that the implementation be both algorithm agile and high speed. The advantages and disadvantages of each of these implementation options were discussed in Section 3.2 and specific examples of these implementation options were examined in Chapter 2. To facilitate the development of an architecture that would satisfy the identified system requirements, a study of a wide range of block ciphers was undertaken to develop an understanding of the functional requirements of these algorithms. This included both an in-depth implementation study of the AES candidate algorithm finalists as well as an examination of other well known block ciphers. This study identified the core operations of each block cipher investigated and resulted in a set of architectural requirements for use in designing the COBRA specialized reconfigurable architecture.

The COBRA architecture was shown to provide flexibility, fast reconfiguration, minimized overhead, and minimized off-chip communication. Because block ciphers are dataflow oriented, a coarse-grained, generalized, run-time reconfigurable datapath was implemented to ensure full support of algorithm-specific operations. The use of a VLIW architecture resulted in a system that is capable of quickly adapting to different operational needs and interface bottlenecks were eliminated via tight coupling of the datapath with the VLIW processor core. A user-controlled dual clocking scheme was implemented for controlling the COBRA architecture, allowing the programmer to optimize system operation for desired mappings. A custom assembly language was developed for use in programming the architecture and software tools were provided to both compile the assembly language into a usable microcode format and to perform a system timing analysis so as to provide the user with system timing requirements for a given application.

Once the COBRA architecture was developed, three block ciphers — RC6, Rijndael, and Serpent — were implemented to measure performance characteristics. These ciphers were chosen based on the orthogonality of their core functions in an effort to exercise as many of the architecture's features as possible. Algorithms were mapped to the COBRA architecture and implemented using the COBRA assembly language and microcode format. Performance data was gathered in terms of both cycle counts and gate counts so as to evaluate the implementations of the targeted block ciphers.

9.2 Conclusions

The following describes the key results of this research:

- *Block cipher characterization*: An in-depth analysis of a wide range of block ciphers was performed in order to develop a hardware architecture optimized for block cipher implementation. This analysis led to a set of requirements for the COBRA architecture that would allow for coverage of a wide variety of algorithms via a generalized datapath and instruction set.

- *Specialized reconfigurable architecture*: The COBRA architecture was developed based on the hybrid model of specialized reconfigurable architectures to provide an environment targeted towards accelerating block cipher implementations. The COBRA architecture provides a coarse-grained, run-time reconfigurable solution that meets the established goals of algorithm agility and significantly increased performance versus software implementations.

- *Software tools*: An assembler and timing analyzer were developed to facilitate ease of use of the COBRA architecture. The assembler compiles a custom assembly language program into the required COBRA microcode format. The timing analyzer evaluates every mapping of the COBRA architecture for a given program and provides the user with timing information for use in specifying the dual system clocks.

- *Algorithm agility*: Multiple block ciphers were successfully implemented targeting the COBRA architecture, employing on-the-fly reconfiguration to perform both key scheduling and encryption. This on-the-fly reconfiguration incurs an extremely low overhead cost, providing a significant improvement over the on-the-fly reconfiguration capability available in current FPGAs. The ability to switch between cryptographic algorithms was provided through reprogramming of the COBRA iRAM with the new algorithm's microcoded implementation.

- *Performance*: Implementations in the COBRA architecture yielded cycle counts that significantly outperformed software implementations as the number of rounds implemented was increased. Performance was measured in cycles per encryption. The performance levels of the COBRA implementations were also shown to asymptotically approached the performance levels of commercial hardware as the number of rounds implemented was increased. However, the cost associated with this performance is increased hardware resource requirements.

- *Performance versus area tradeoff*: A variant of the classical time-area (TA) product, termed the cycles-gates (CG) product, was introduced and this metric indicated that pipelined implementations generally outperformed loop unrolled implementations when operating in non-feedback modes. The CG metric also indicated that the best area versus performance tradeoff occurs when all of the rounds of a cipher are implemented within the COBRA architecture.

9.3 Future Work

The following areas for future research using the COBRA architecture have been identified:

- *Analysis of power consumption*: COBRA has the potential of being well suited for low-power implementations as only some of its resources are required for a given cipher configuration. A study of how unused elements can be “switched off” would allow for further minimization of power consumption.

- *Full-custom implementation*: A VLSI implementation of the COBRA architecture would provide a great deal of data regarding system timing, power consumption, gate count, and production cost. Detailed timing data would allow the timing analyzer to provide more accurate clock frequency estimates and allow for the evaluation of block cipher implementations based on throughput data. Moreover, a VLSI implementation would provide far more accurate gate count data as compared to the estimates provided by the Xilinx FPGA tools.

- *Extending ease of use*: To provide complete ease of use, support of a high-level language for describing a COBRA mapping is a necessity. This would require either a compiler if a software language such as C is to be used, or a synthesizer if a hardware description language such as VHDL or Verilog is to be used. Given the nature of the COBRA architecture, any compiler or synthesizer would be targeted at supporting a subset of the standard language constructs as well as a newly created set of constructs specific to the COBRA architecture.

The COBRA architecture has been demonstrated to be a promising option for the implementation of block ciphers. The continuation of this research may result in the development of a more refined architecture that improves upon the current architecture in the areas of power consumption, performance, gate count, and cost.

Appendix A

AES Finalists Implementation Data

Tables A.1, A.2, A.3, A.4, A.5, A.6, A.7, and A.8 detail the throughput measurements for the FPGA implementations of the four architecture types for each of the AES finalists for both non-feedback and feedback mode from Chapter 4. The architecture types — loop unrolling (LU), full or partial pipelining (PP), and partial pipelining with sub-pipelining (SP) — are listed along with the number of stages and (if necessary) sub-pipeline stages in the associated implementation; e.g., LU-4 implies a loop unrolling architecture with four rounds, while SP-2-1 implies a partially pipelined architecture with two stages and one sub-pipeline stage per pipeline stage. As a result, the SP-2-1 architecture implements two rounds of the given cipher with a total of two stages per round.

It is interesting to note that the slice counts do not increase linearly based on the number of rounds in the implementation of a given architecture. The round function for each algorithm is comprised of combinatorial logic which is implemented via the look-up-tables of the CLB slice while the round keys occupy the register bits of the CLB slice. While the total number of register bits required for round key storage is fixed, the number of slices required for storage of the round keys varies with the number of rounds implemented. Because the round functions do not make use of the associated register bits within a CLB slice, these bits may be used for round key storage, effectively packing the round keys with the round functions to minimize CLB slice use. Little packing is possible when only one round is implemented, resulting in most of the round keys being stored in CLB slices whose look-up-tables are unused. However, as more rounds are unrolled or pipelined, more round keys may occupy the unused register bits of the CLB slices used to implement the round functions, resulting in greater packing. As a result, while the size of the implementation increases when the number of rounds unrolled or pipelined is increased, this increase is partially offset by the packing of the round keys within the round structure.

			Clock	**Cycles**		
			Frequency	**per**	**Throughput**	
Algorithm	**Architecture**	**Slices**	**(MHz)**	**Block**	**(Mbit/s)**	**TPS**
RC6	LU-1	2638	13.8	20	88.5	33558
RC6	LU-2	3069	7.3	10	94.0	30638
RC6	LU-4	4070	3.7	5	94.8	23304
RC6	LU-5	4476	2.9	4	92.2	20604
RC6	LU-10	6406	1.5	2	97.4	15206
RC6	PP-2	3189	19.8	10	253.0	79325
RC6	PP-4	4411	12.3	5	315.5	71524
RC6	PP-5	4848	12.1	4	386.7	79762
RC6	PP-10	7412	13.3	2	848.1	114426
RC6	SP-1-1	2967	26.2	20	167.6	56498
RC6	SP-2-1	3709	26.4	10	337.8	91087
RC6	SP-4-1	5229	24.6	5	629.8	120436
RC6	SP-5-1	5842	25.8	4	825.2	141250
RC6	SP-10-1	8999	26.6	2	1704.6	189425
RC6	SP-1-2	3134	39.1	20	250.0	79769
RC6	SP-2-2	4062	38.9	10	497.4	122448
RC6	SP-4-2	5908	31.3	5	802.3	135795
RC6	SP-5-2	6415	33.3	4	1067.0	166330
RC6	**SP-10-2**	**10856**	**37.5**	**2**	**2397.9**	**220881**
Rijndael	LU-1	3528	25.3	11	294.2	83387
Rijndael	LU-2	5302	14.1	6	300.1	56605
Rijndael	LU-5	10286	5.6	3	237.4	23084
Rijndael	PP-2	5281	23.5	5.5	545.9	103372
Rijndael	PP-5	10533	20.0	2.2	1165.8	110680
Rijndael	SP-1-1	3061	40.4	10.5	491.9	160702
Rijndael	**SP-2-1**	**4871**	38.9	5.25	949.1	**194837**
Rijndael	**SP-5-1**	10992	**31.8**	**2.1**	**1937.9**	176297

Table A.1: RC6 and Rijndael performance evaluation — speed optimized non-feedback mode of operation

			Clock	**Cycles**		
			Frequency	**per**	**Throughput**	
Algorithm	**Architecture**	**Slices**	**(MHz)**	**Block**	**(Mbit/s)**	**TPS**
Serpent	LU-1	5511	15.5	32	61.9	11236
Serpent	LU-8	7964	13.9	4	444.2	55771
Serpent	LU-32	8103	2.4	1	312.3	38544
Serpent	PP-8	6849	30.4	4	971.8	141886
Serpent	**PP-32**	**9004**	**38.0**	**1**	**4860.2**	**539778**
Twofish	LU-1	2666	13.0	16	104.2	39079
Twofish	LU-2	3392	7.1	8	113.6	33495
Twofish	LU-4	4665	3.3	4	106.8	22890
Twofish	LU-8	6990	1.7	2	108.1	15464
Twofish	PP-2	3519	11.9	8	190.4	54115
Twofish	PP-4	5044	11.5	4	369.3	73218
Twofish	PP-8	7817	10.8	2	689.5	88210
Twofish	SP-1-1	3053	29.9	16	239.2	78339
Twofish	SP-2-1	3869	28.6	8	457.1	118137
Twofish	SP-4-1	5870	27.3	4	872.3	148606
Twofish	SP-8-1	9345	24.8	2	1585.3	169639
Twofish	SP-1-2	2954	37.9	16	303.4	102719
Twofish	SP-2-2	4012	45.8	8	732.5	182580
Twofish	SP-4-2	6101	38.8	4	1242.1	203597
Twofish	**SP-8-2**	**10008**	**37.4**	**2**	**2395.7**	**239380**

Table A.2: Serpent and Twofish performance evaluation — speed optimized non-feedback mode of operation

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)	TPS
RC6	LU-1	2784	14.6	20	93.6	33630
RC6	LU-2	3165	6.9	10	88.4	27934
RC6	LU-4	4058	3.7	5	94.7	23335
RC6	LU-5	4465	3.0	4	97.1	21744
RC6	LU-10	6398	1.5	2	97.3	15205
RC6	PP-2	3215	19.1	10	244.6	76091
RC6	PP-4	4322	14.1	5	362.0	83760
RC6	PP-5	4881	12.8	4	411.1	84225
RC6	PP-10	7393	11.7	2	751.3	101623
RC6	SP-1-1	3012	28.1	20	180.1	59778
RC6	SP-2-1	3609	24.3	10	311.3	86245
RC6	SP-4-1	4969	25.3	5	648.0	130411
RC6	SP-5-1	5700	24.8	4	792.7	139065
RC6	SP-10-1	8687	23.6	2	1509.1	173714
RC6	SP-1-2	3136	37.4	20	239.1	76233
RC6	SP-2-2	3942	37.7	10	483.1	122558
RC6	SP-4-2	5637	34.8	5	891.3	158123
RC6	SP-5-2	6471	26.2	4	839.7.0	129760
RC6	**SP-10-2**	**10308**	**31.2**	**2**	**1996.9**	**193720**
Rijndael	LU-1	3488	24.9	11	290.1	83159
Rijndael	LU-2	5276	13.9	6	296.4	56184
Rijndael	LU-5	10276	5.7	3	243.1	23654
Rijndael	PP-2	5275	24.7	5.5	575.3	109066
Rijndael	PP-5	10550	16.4	2.2	956.8	90692
Rijndael	SP-1-1	3540	40.0	10.5	487.7	137763
Rijndael	**SP-2-1**	**5449**	40.0	5.25	975.6	**179034**
Rijndael	**SP-5-1**	10923	**30.8**	**2.1**	**1878.5**	**171976**

Table A.3: RC6 and Rijndael performance evaluation — area optimized non-feedback mode of operation

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)	TPS
Serpent	LU-1	5890	19.2	32	77.0	13065
Serpent	LU-8	6467	10.4	4	333.3	51535
Serpent	LU-32	10936	2.8	1	355.8	32538
Serpent	PP-8	6849	38.8	4	1241.6	181277
Serpent	**PP-32**	**11988**	**39.3**	**1**	**5035.0**	**420004**
Twofish	LU-1	2695	16.0	16	127.7	47380
Twofish	LU-2	3372	6.6	8	106.1	31469
Twofish	LU-4	4642	3.5	4	112.2	24162
Twofish	LU-8	7011	1.6	2	103.4	14752
Twofish	PP-2	3467	12.8	8	204.7	59030
Twofish	PP-4	5033	11.5	4	366.8	72882
Twofish	PP-8	7814	10.6	2	675.9	86499
Twofish	SP-1-1	2865	28.2	16	225.7	78771
Twofish	SP-2-1	3619	28.7	8	459.6	126988
Twofish	SP-4-1	5439	26.5	4	849.2	156129
Twofish	SP-8-1	8745	26.7	2	1709.0	195425
Twofish	SP-1-2	2976	41.4	16	331.2	111306
Twofish	SP-2-2	3840	38.6	8	617.6	160842
Twofish	SP-4-2	5720	35.0	4	1120.4	195883
Twofish	**SP-8-2**	**9486**	**36.9**	**2**	**2358.8**	**248666**

Table A.4: Serpent and Twofish performance evaluation — area optimized non-feedback mode of operation

			Clock	**Cycles**		
			Frequency	**per**	**Throughput**	
Algorithm	**Architecture**	**Slices**	**(MHz)**	**Block**	**(Mbit/s)**	**TPS**
RC6	LU-1	2638	13.8	20	88.5	33558
RC6	LU-2	3069	7.3	10	94.0	30638
RC6	LU-4	4070	3.7	5	94.8	23304
RC6	LU-5	4476	2.9	4	92.2	20604
RC6	LU-10	6406	1.5	2	97.4	15206
RC6	**PP-2**	**3189**	**19.8**	**20**	**126.5**	**39662**
RC6	PP-4	4411	12.3	20	78.9	17881
RC6	PP-5	4848	12.1	20	77.3	15952
RC6	PP-10	7412	13.3	20	84.8	11443
RC6	SP-1-1	2967	26.2	40	83.8	28249
RC6	SP-2-1	3709	26.4	40	84.5	22772
RC6	SP-4-1	5229	24.6	40	78.7	15055
RC6	SP-5-1	5842	25.8	40	82.5	14125
RC6	SP-10-1	8999	26.6	40	85.2	9471
RC6	SP-1-2	3134	39.1	60	83.3	26590
RC6	SP-2-2	4062	38.9	60	82.9	20408
RC6	SP-4-2	5908	31.3	60	66.9	11316
RC6	SP-5-2	6415	33.3	60	71.1	11089
RC6	SP-10-2	10856	37.5	60	79.9	7363
Rijndael	**LU-1**	**3528**	25.3	11	294.2	**83387**
Rijndael	**LU-2**	5302	**14.1**	**6**	**300.1**	56605
Rijndael	LU-5	10286	5.6	3	237.4	23084
Rijndael	PP-2	5281	23.5	11	273.0	51686
Rijndael	PP-5	10533	20.0	11	233.2	22136
Rijndael	SP-1-1	3061	40.4	21	246.0	80351
Rijndael	SP-2-1	4871	38.9	21	237.3	48709
Rijndael	SP-5-1	10992	31.8	21	193.8	17630

Table A.5: RC6 and Rijndael performance evaluation — speed optimized feedback mode of operation

Algorithm	**Architecture**	**Slices**	**Clock Frequency (MHz)**	**Cycles per Block**	**Throughput (Mbit/s)**	**TPS**
Serpent	LU-1	5511	15.5	32	61.9	11236
Serpent	**LU-8**	**7964**	**13.9**	**4**	**444.2**	**55771**
Serpent	LU-32	8103	2.4	1	312.3	38544
Serpent	PP-8	6849	30.4	32	121.5	17736
Serpent	PP-32	9004	38.0	32	151.9	16868
Twofish	LU-1	2666	13.0	16	104.2	39079
Twofish	LU-2	3392	7.1	8	113.6	33495
Twofish	LU-4	4665	3.3	4	106.8	22890
Twofish	LU-8	6990	1.7	2	108.1	15464
Twofish	PP-2	3519	11.9	16	95.2	27058
Twofish	PP-4	5044	11.5	16	92.3	18305
Twofish	PP-8	7817	10.8	16	86.2	11026
Twofish	**SP-1-1**	**3053**	29.9	32	119.6	**39169**
Twofish	SP-2-1	3869	28.6	32	114.3	29534
Twofish	SP-4-1	5870	27.3	32	109.0	18576
Twofish	SP-8-1	9345	24.8	32	99.1	10602
Twofish	SP-1-2	2954	37.9	48	101.1	34240
Twofish	**SP-2-2**	4012	**45.8**	**48**	**122.1**	30430
Twofish	SP-4-2	6101	38.8	48	103.5	16966
Twofish	SP-8-2	10008	37.4	48	99.8	9974

Table A.6: Serpent and Twofish performance evaluation — speed optimized feedback mode of operation

Algorithm	**Architecture**	**Slices**	**Clock Frequency (MHz)**	**Cycles per Block**	**Throughput (Mbit/s)**	**TPS**
RC6	LU-1	2784	14.6	20	93.6	33630
RC6	LU-2	3165	6.9	10	88.4	27934
RC6	LU-4	4058	3.7	5	94.7	23335
RC6	LU-5	4465	3.0	4	97.1	21744
RC6	LU-10	6398	1.5	2	97.3	15205
RC6	**PP-2**	**3215**	**19.1**	**20**	**122.3**	**38046**
RC6	PP-4	4322	14.1	20	90.5	20940
RC6	PP-5	4881	12.8	20	82.2	16845
RC6	PP-10	7393	11.7	20	75.1	10162
RC6	SP-1-1	3012	28.1	40	90.0	29889
RC6	SP-2-1	3609	24.3	40	77.8	21561
RC6	SP-4-1	4969	25.3	40	81.0	16301
RC6	SP-5-1	5700	24.8	40	79.3	13907
RC6	SP-10-1	8687	23.6	40	75.5	8686
RC6	SP-1-2	3136	37.4	60	79.7	25411
RC6	SP-2-2	3942	37.7	60	80.5	20426
RC6	SP-4-2	5637	34.8	60	74.3	13177
RC6	SP-5-2	6471	26.2	60	56.0	8651
RC6	SP-10-2	10308	31.2	60	66.6	6457
Rijndael	**LU-1**	**3488**	24.9	11	290.1	**83159**
Rijndael	**LU-2**	5276	**13.9**	**6**	**296.4**	56184
Rijndael	LU-5	10276	5.7	3	243.1	23654
Rijndael	PP-2	5274	24.7	11	287.7	54544
Rijndael	PP-5	10550	16.4	11	191.4	18138
Rijndael	SP-1-1	3540	40.0	21	243.8	68881
Rijndael	SP-2-1	5449	40.0	21	243.9	44758
Rijndael	SP-5-1	10923	30.8	21	187.8	17198

Table A.7: RC6 and Rijndael performance evaluation — area optimized feedback mode of operation

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)	TPS
Serpent	LU-1	5890	19.2	32	77.0	13065
Serpent	**LU-8**	**6467**	**10.4**	**4**	**333.3**	**51535**
Serpent	LU-32	10936	2.8	1	355.8	32538
Serpent	PP-8	6849	38.8	32	155.2	22660
Serpent	PP-32	11988	39.3	32	157.3	13125
Twofish	**LU-1**	**2695**	**16.0**	**16**	**127.7**	**47380**
Twofish	LU-2	3372	6.6	8	106.1	31469
Twofish	LU-4	4642	3.5	4	112.2	24162
Twofish	LU-8	7011	1.6	2	103.4	14752
Twofish	PP-2	3467	12.8	16	102.3	29515
Twofish	PP-4	5033	11.5	16	91.7	18221
Twofish	PP-8	7814	10.6	16	84.5	10812
Twofish	SP-1-1	2865	28.2	32	112.8	39386
Twofish	SP-2-1	3619	28.7	32	114.9	31747
Twofish	SP-4-1	5439	26.5	32	106.1	19516
Twofish	SP-8-1	8745	26.7	32	106.8	12214
Twofish	SP-1-2	2976	41.4	48	110.4	37102
Twofish	SP-2-2	3840	38.6	48	102.9	26807
Twofish	SP-4-2	5720	35.0	48	93.4	16324
Twofish	SP-8-2	9486	36.9	48	98.3	10361

Table A.8: Serpent and Twofish performance evaluation — area optimized feedback mode of operation

Appendix B

COBRA Assembly Language

B.1 Command Line Execution

The COBRA software tool suite is command-line driven and has been coded in C to guarantee portability across all possible software platforms. The command line required to execute the software tool suite takes the form:

$$ubmasm \ \ < infile >$$

The input file is specified without an extension — the file extension is required to be *.asm* and the file name cannot be longer than forty characters. The assembler automatically generates the following output files:

- <infile>.tv — the vector file used to program the COBRA architecture.
- <infile>.dat — the vector file with comments — not used by the COBRA architecture.
- <infile>.twr — a timing report file for the given program.

B.2 Language

The COBRA assembly language is subdivided based on operation type. What follows is a description of each language construct as well as examples of construct usage. Execution of the assembler will also be discussed along with the format of the input and output files.

LB00E Instruction

The *LB00E* instruction is used to configure the basic elements within RCE 00. Table B.1 details the elements within RCE 00 that may be configured using this instruction. Note that when *M9* is assigned *IN* the RCE is in bypass mode. Multiplexor elements that determine the secondary input to an element that is in pass-through mode need not be specified. Note that the secondary input terminology is as specified in Section 6.2 and that functionality is as defined in Chapter 6.

The example shown in Table B.2 illustrates the usage of the *LB00E* instruction. Elements may be specified for configuration in any order and the *END* statement indicates that configuration of RCE 00 is complete. In the example, because *A1* is configured to be in pass-through mode, multiplexor *M1* need not be specified as the secondary input to *A1* is irrelevant. The 5-bit values specified for *E1* and *E2* are the shift or rotation value used due to *E1* and *E2* being configured to operate in normal mode as indicated by Operand 1 being set to *NRM*. Note that the shift or rotation values are specified in decimal notation. However, *E3* is configured to operate in data-dependent mode as Operand 1 is set to *OVRD*. In this case, the 5-bit shift or rotation value is ignored as the shift or rotation value is derived from the five LSBs of the secondary input routed to each of the *E* elements by *M6*.

When configuring *B[2:1]*, an Operand 1 specification of *IN1* refers to a pass-through of the primary input while an Operand 1 specification of *IN2* refers to a pass-through of the secondary input specified by the associated multiplexors (*M3* and *M4*). When configuring *C1*, an Operand 1 setting of *NRM* indicates that the LUTs will operate in normal mode with the entire 32-bit input being used to address the LUTs. An Operand 1 setting of *SH4* indicates that the four LSBs of the 32-bit input will be used to address all of the LUTs while an Operand 1 setting of *SH8* indicates that the eight LSBs of the 32-bit input will be used to address all of the LUTs. Operand 2 for element *C1* indicates operation in either 4-LUT mode, where eight 4-input LUTs are used, or 8-LUT mode, where four 8-input LUTs are used. Operand 3 indicates that the *C1* element will operate in either pass-through mode, specified as *IN*, or LUT mode, specified as *LUT*.

Element	**Operand 1**	**Operand 2**	**Operand 3**
M9	E3,IN		
MO	M9,REG		
M6	INB,INC,IND,INER		
M5	INB,INC,IND,INER		
M4	INB,INC,IND,INER		
M3	INB,INC,IND,INER		
M2	INB,INC,IND,INER		
M1	INB,INC,IND,INER		
A3	XOR,AND,OR,IN		
A2	XOR,AND,OR,IN		
A1	XOR,AND,OR,IN		
B2	ADD32,ADD16,ADD8,SUB32,SUB16,SUB8,IN1,IN2		
B1	ADD32,ADD16,ADD8,SUB32,SUB16,SUB8,IN1,IN2		
F	IN,MULGF		
E3	NRM,OVRD	ROL,SHL,SHR,IN	5-bit Value
E2	NRM,OVRD	ROL,SHL,SHR,IN	5-bit Value
E1	NRM,OVRD	ROL,SHL,SHR,IN	5-bit Value
C1	SH4,SH8,NRM	4,8	IN,LUT

Table B.1: LB00E assembly instruction — usage

LB00E				
E1	<	NRM	SHL	31
A1	<	IN		
A2	<	IN		
B1	<	IN1		
C1	<	NRM	8	IN
F	<	IN		
E2	<	NRM	SHL	1
B2	<	IN1		
A3	<	XOR		
E3	<	NRM	OVRD	0
M5	<	INER		
M6	<	INC		
M9	<	E3		
MO	<	M9		
END				

Table B.2: LB00E assembly instruction — example

LB01E Instruction

The *LB01E* instruction is used to configure the basic elements within RCE MUL 01. Table B.3 details the elements within RCE MUL 01 that may be configured using this instruction. Note that when *M9* is assigned *IN* the RCE MUL is in bypass mode. Multiplexor elements that determine the secondary input to an element that is in pass-through mode need not be specified. Note that the secondary input terminology is as specified in Section 6.2 and that functionality is as defined in Chapter 6.

The example shown in Table B.4 illustrates the usage of the *LB01E* instruction. Elements may be specified for configuration in any order and the *END* statement indicates that configuration of RCE MUL 01 is complete. In the example, because *A1* is configured to be in pass-through mode, multiplexor *M1* need not be specified as the secondary input to *A1* is irrelevant. Similarly, the *M7* multiplexor need not be specified as *D* is configured to be in pass-through mode. Should *D* be configured otherwise, *M7* must be specified. The 5-bit values specified for *E1* and *E2* are the shift or rotation value used due to *E1* and *E2* being configured to operate in normal mode as indicated by Operand 1 being set to *NRM*. Note that the shift or rotation values are specified in decimal notation. However, *E3* is configured to operate in data-dependent mode as Operand 1 is set to *OVRD*. In this case, the 5-bit shift or rotation value is ignored as the shift or rotation value is derived from the five LSBs of the secondary input routed to each of the *E* elements by *M6*.

When configuring *B[2:1]*, an Operand 1 specification of *IN1* refers to a pass-through of the primary input while an Operand 1 specification of *IN2* refers to a pass-through of the secondary input specified by the associated multiplexors (*M3* and *M4*). When configuring *C[2:1]*, an Operand 1 setting of emphNRM indicates that the LUTs will operate in normal mode with the entire 32-bit input being used to address the LUTs. An Operand 1 setting of *SH4* indicates that the four LSBs of the 32-bit input will be used to address all of the LUTs while an Operand 1 setting of *SH8* indicates that the eight LSBs of the 32-bit input will be used to address all of the LUTs. Operand 2 for elements *C[2:1]* indicates operation in either 4-LUT mode, where eight 4-input LUTs are used, or 8-LUT mode, where four 8-input LUTs are used. Operand 3 indicates that the *C* element will operate in either pass-through mode, specified as *IN*, or LUT mode, specified as *LUT*.

Element	Operand 1	Operand 2	Operand 3
M9	E3,IN		
MO	M9,REG		
M7	M5,GF		
M8	INB,INC,IND,INER		
M6	INB,INC,IND,INER		
M5	INB,INC,IND,INER		
M4	INB,INC,IND,INER		
M3	INB,INC,IND,INER		
M2	INB,INC,IND,INER		
M1	INB,INC,IND,INER		
A3	XOR,AND,OR,IN		
A2	XOR,AND,OR,IN		
A1	XOR,AND,OR,IN		
B2	ADD32,ADD16,ADD8,SUB32,SUB16,SUB8,IN1,IN2		
B1	ADD32,ADD16,ADD8,SUB32,SUB16,SUB8,IN1,IN2		
F	IN,MULGF		
E3	NRM,OVRD	ROL,SHL,SHR,IN	5-bit Value
E2	NRM,OVRD	ROL,SHL,SHR,IN	5-bit Value
E1	NRM,OVRD	ROL,SHL,SHR,IN	5-bit Value
C1	SH4,SH8,NRM	4,8	IN,LUT
C2	SH4,SH8,NRM	4,8	IN,LUT
D	MUL32,MUL16,SQ,IN		

Table B.3: LB01E assembly instruction — usage

LB01E				
E1	<	NRM	SHL	31
A1	<	IN		
A2	<	IN		
B1	<	IN1		
C1	<	NRM	8	IN
F	<	IN		
D	<	IN		
E2	<	NRM	SHL	1
C2	<	NRM	8	IN
B2	<	IN1		
A3	<	XOR		
E3	<	NRM	OVRD	0
M5	<	INER		
M6	<	INC		
M9	<	E3		
MO	<	M9		
END				

Table B.4: LB01E assembly instruction — example

LB02E Instruction

The *LB02E* instruction is used to configure the basic elements within RCE 02. The elements within RCE 02 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB02E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB03E Instruction

The *LB03E* instruction is used to configure the basic elements within RCE MUL 03. The elements within RCE MUL 03 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB03E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB10E Instruction

The *LB10E* instruction is used to configure the basic elements within RCE 10. The elements within RCE 10 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB10E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB11E Instruction

The *LB11E* instruction is used to configure the basic elements within RCE MUL 11. The elements within RCE MUL 11 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB11E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB12E Instruction

The *LB12E* instruction is used to configure the basic elements within RCE 12. The elements within RCE 12 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB12E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB13E Instruction

The *LB13E* instruction is used to configure the basic elements within RCE MUL 13. The elements within RCE MUL 13 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB13E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB20E Instruction

The *LB20E* instruction is used to configure the basic elements within RCE 20. The elements within RCE 20 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB20E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB21E Instruction

The *LB21E* instruction is used to configure the basic elements within RCE MUL 21. The elements within RCE MUL 21 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB21E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB22E Instruction

The *LB22E* instruction is used to configure the basic elements within RCE 22. The elements within RCE 22 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB22E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB23E Instruction

The *LB23E* instruction is used to configure the basic elements within RCE MUL 23. The elements within RCE MUL 23 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB23E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB30E Instruction

The *LB30E* instruction is used to configure the basic elements within RCE 30. The elements within RCE 30 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB30E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB31E Instruction

The *LB31E* instruction is used to configure the basic elements within RCE MUL 31. The elements within RCE MUL 31 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB31E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB32E Instruction

The *LB32E* instruction is used to configure the basic elements within RCE 32. The elements within RCE 32 are the same as those within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB32E* instead of *LB00E* at the beginning of the assembly code block — refer to Tables B.1 and B.2 for usage information and a configuration example.

LB33E Instruction

The *LB33E* instruction is used to configure the basic elements within RCE MUL 33. The elements within RCE MUL 33 are the same as those within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB33E* instead of *LB01E* at the beginning of the assembly code block — refer to Tables B.3 and B.4 for usage information and a configuration example.

LB00B Instruction

The *LB00B* instruction is used to configure the RCE 00 bank address of element *C1* when operating in 4-LUT mode. Table B.5 details the configuration of the bank address of *C1* when using this instruction. The example shown in Table B.6 illustrates the usage of the *LB00B* instruction when configuring *C1* to bank address two. The value specified for *C1* ranges from zero to seven.

Element	**Operand 1**
C1	0 to 7

Table B.5: LB00B assembly instruction — usage

```
LB00B
C1        <   2
END
```

Table B.6: LB00B assembly instruction — example

LB01B Instruction

The *LB01B* instruction is used to configure the RCE MUL 00 bank address of elements *C1* and *C2* when operating in 4-LUT mode. Table B.7 details the configuration of the *C1* and *C2* bank addresses when using this instruction. The example shown in Table B.8 illustrates the usage of the *LB01B* instruction when configuring *C1* to bank address two and *C1* to bank address three. The values specified for *C1* and *C2* range from zero to seven.

Element	**Operand 1**
C1	0 to 7
C2	0 to 7

Table B.7: LB01B assembly instruction — usage

```
LB01B
C1        <   2
C2        <   3
END
```

Table B.8: LB01B assembly instruction — example

LB02B Instruction

The *LB02B* instruction is used to configure the *C1* element within RCE 02. The *C1* element within RCE 02 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB02B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB03B Instruction

The *LB03B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 03. The *C1* and *C2* elements within RCE MUL 03 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB03B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB10B Instruction

The *LB10B* instruction is used to configure the *C1* element within RCE 10. The *C1* element within RCE 10 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB10B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB11B Instruction

The *LB11B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 11. The *C1* and *C2* elements within RCE MUL 11 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB11B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB12B Instruction

The *LB12B* instruction is used to configure the *C1* element within RCE 12. The *C1* element within RCE 12 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB12B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB13B Instruction

The *LB13B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 13. The *C1* and *C2* elements within RCE MUL 13 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB13B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB20B Instruction

The *LB20B* instruction is used to configure the *C1* element within RCE 20. The *C1* element within RCE 20 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB20B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB21B Instruction

The *LB21B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 21. The *C1* and *C2* elements within RCE MUL 21 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB21B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB22B Instruction

The *LB22B* instruction is used to configure the *C1* element within RCE 22. The *C1* element within RCE 22 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB22B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB23B Instruction

The *LB23B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 23. The *C1* and *C2* elements within RCE MUL 23 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB23B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB30B Instruction

The *LB30B* instruction is used to configure the *C1* element within RCE 30. The *C1* element within RCE 30 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB30B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB31B Instruction

The *LB31B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 31. The *C1* and *C2* elements within RCE MUL 31 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB31B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB32B Instruction

The *LB32B* instruction is used to configure the *C1* element within RCE 32. The *C1* element within RCE 32 are the same as the one within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB32B* instead of *LB00B* at the beginning of the assembly code block — refer to Tables B.5 and B.6 for usage information and a configuration example.

LB33B Instruction

The *LB33B* instruction is used to configure the *C1* and *C2* elements within RCE MUL 33. The *C1* and *C2* elements within RCE MUL 33 are the same as the one within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB33B* instead of *LB01B* at the beginning of the assembly code block — refer to Tables B.7 and B.8 for usage information and a configuration example.

LB00C Instruction

The *LB00C* instruction is used to configure the data in the *C1* element within RCE 00 as detailed in Table B.9. The example shown in Table B.10 illustrates the usage of the *LB00C* instruction for both 4-LUT and 8-LUT modes of operation.

When operating in 4-LUT mode, the three MSBs of the address indicate the bank address while the four LSBs of the address indicate the address within the LUTs to store the data. In the example, if it is assumed that the LUT is configured to operate in 4-LUT mode, the address $E0_{16}$ corresponds to 11100000_2. Therefore, the data will be stored in 4-LUT bank address 111_2 (seven) at local address 0000_2 (zero) for each of the 4-input LUTs. The data is listed MSBs to LSBs where 4-LUT zero will receive the value 1_{16}, 4-LUT one will receive the value 2_{16}, etc. Note that the upper thirty two bits of data are ignored as RCE 00 does not have a LUT element *C2*.

When operating in 8-LUT mode, the data will be stored in local address $E0_{16}$ for each of the 8-input LUTs. In the example, if it is assumed that the LUT is configured to operate in 8-LUT mode, the data will be stored in local address $E0_{16}$ for each of the 8-input LUTs. The data is listed MSBs to LSBs where 8-LUT zero will receive the value 21_{16}, 8-LUT one will receive the value 43_{16}, etc. Note that the upper thirty two bits of data are ignored as RCE 00 does not have a LUT element *C2*.

Element	**Operand 1**
8-Bit Address	64-Bit Hexadecimal Data

Table B.9: LB00C assembly instruction — usage

```
LB00C
E0      <   0000000087654321
END
```

Table B.10: LB00C assembly instruction — example

LB01C Instruction

The *LB01C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 01 as detailed in Table B.11. The example shown in Table B.12 illustrates the usage of the *LB01C* instruction for both 4-LUT and 8-LUT modes of operation.

When operating in 4-LUT mode, the three MSBs of the address indicate the bank address while the four LSBs of the address indicate the address within the LUTs to store the data. In the example, if it is assumed that the LUT is configured to operate in 4-LUT mode, the address $E0_{16}$ corresponds to 11100000_2. Therefore, the data will be stored in

4-LUT bank address 111_2 (seven) at local address 0000_2 (zero) for each of the 4-input LUTs of both *C1* and *C2*. The data is listed MSBs to LSBs where *C1* 4-LUT zero will receive the value 0_{16}, *C1* 4-LUT one will receive the value 1_{16}, etc. Note that *C2* 4-LUT zero will receive the value 8_{16}, *C2* 4-LUT one will receive the value 9_{16}, etc.

When operating in 8-LUT mode, the data will be stored in local address $E0_{16}$ for each of the 8-input LUTs in both *C1* and *C2*. In the example, if it is assumed that the LUT is configured to operate in 8-LUT mode, the data will be stored in local address $E0_{16}$ for each of the 8-input LUTs. The data is listed MSBs to LSBs where *C1* 8-LUT zero will receive the value 10_{16}, *C1* 8-LUT one will receive the value 32_{16}, etc. Note that *C2* 8-LUT zero will receive the value 98_{16}, *C2* 8-LUT one will receive the value BA_{16}, etc.

Element	**Operand 1**
8-Bit Address	32-Bit Hexadecimal Data

Table B.11: LB01C assembly instruction — usage

```
LB00C
E0      <   FEDCBA9876543210
END
```

Table B.12: LB01C assembly instruction — example

LB02C Instruction

The *LB02C* instruction is used to configure the data in the *C1* element within RCE 02. The configuration of the data in the *C1* element within RCE 02 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB02C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB03C Instruction

The *LB03C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 03. The configuration of the data in the *C1* and *C2* elements within RCE MUL 03 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB03C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB10C Instruction

The *LB10C* instruction is used to configure the data in the *C1* element within RCE 10. The configuration of the data in the *C1* element within RCE 10 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB10C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB11C Instruction

The *LB11C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 11. The configuration of the data in the *C1* and *C2* elements within RCE MUL 11 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB11C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB12C Instruction

The *LB12C* instruction is used to configure the data in the *C1* element within RCE 12. The configuration of the data in the *C1* element within RCE 12 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB12C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB13C Instruction

The *LB13C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 13. The configuration of the data in the *C1* and *C2* elements within RCE MUL 13 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB13C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB20C Instruction

The *LB20C* instruction is used to configure the data in the *C1* element within RCE 20. The configuration of the data in the *C1* element within RCE 20 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB20C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB21C Instruction

The *LB21C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 21. The configuration of the data in the *C1* and *C2* elements within RCE MUL

21 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB21C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB22C Instruction

The *LB22C* instruction is used to configure the data in the *C1* element within RCE 22. The configuration of the data in the *C1* element within RCE 22 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB22C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB23C Instruction

The *LB23C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 23. The configuration of the data in the *C1* and *C2* elements within RCE MUL 23 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB23C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB30C Instruction

The *LB30C* instruction is used to configure the data in the *C1* element within RCE 30. The configuration of the data in the *C1* element within RCE 30 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB30C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB31C Instruction

The *LB31C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 31. The configuration of the data in the *C1* and *C2* elements within RCE MUL 31 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB31C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB32C Instruction

The *LB32C* instruction is used to configure the data in the *C1* element within RCE 32. The configuration of the data in the *C1* element within RCE 32 is the same as the configuration of the data in the *C1* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB32C* instead of *LB00C* at the beginning of the assembly code block — refer to Tables B.9 and B.10 for usage information and a configuration example.

LB33C Instruction

The *LB33C* instruction is used to configure the data in the *C1* and *C2* elements within RCE MUL 33. The configuration of the data in the *C1* and *C2* elements within RCE MUL 33 is the same as the configuration of the data in the *C1* and *C2* elements within RCE MUL 01 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB33C* instead of *LB01C* at the beginning of the assembly code block — refer to Tables B.11 and B.12 for usage information and a configuration example.

LB00GF Instruction

The *LB00GF* instruction is used to configure the data in the F element within RCE 00 as detailed in Table B.13. The example shown in Table B.14 illustrates the usage of the *LB00GF* instruction. When programming the F element, the four LSBs of the address are used to indicate which constant element within the fixed field is being specified while the four MSBs are unused. As detailed in Table 6.25, the address $0C_{16}$ used in the example corresponds to K_{30}. As detailed in Section 6.8.5, The sixty four bits of data are listed MSBs to LSBs and correspond to the mapping matrix used to implement the given constant. In the example, row seven will receive the value 01_{16}, row six will receive the value 23_{16}, etc.

Element	**Operand 1**
8-Bit Address	64-Bit Hexadecimal Data

Table B.13: LB00GF assembly instruction — usage

```
LB00GF
0C        <   0123456789ABCDEF
END
```

Table B.14: LB00GF assembly instruction — example

LB01GF Instruction

The *LB01GF* instruction is used to configure the data in the F element within RCE MUL 01. The configuration of the data in the F element within RCE MUL 01 is the same as the configuration of the data in the F element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB01GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB02GF Instruction

The *LB02GF* instruction is used to configure the data in the *F* element within RCE 02. The configuration of the data in the *F* element within RCE MUL 02 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB02GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB03GF Instruction

The *LB03GF* instruction is used to configure the data in the *F* element within RCE MUL 03. The configuration of the data in the *F* element within RCE MUL 03 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB03GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB10GF Instruction

The *LB10GF* instruction is used to configure the data in the *F* element within RCE 10. The configuration of the data in the *F* element within RCE MUL 10 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB10GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB11GF Instruction

The *LB11GF* instruction is used to configure the data in the *F* element within RCE MUL 11. The configuration of the data in the *F* element within RCE MUL 11 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB11GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB12GF Instruction

The *LB12GF* instruction is used to configure the data in the *F* element within RCE 12. The configuration of the data in the *F* element within RCE MUL 12 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB12GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB13GF Instruction

The *LB13GF* instruction is used to configure the data in the *F* element within RCE MUL 13. The configuration of the data in the *F* element within RCE MUL 13 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB13GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB20GF Instruction

The *LB20GF* instruction is used to configure the data in the *F* element within RCE 20. The configuration of the data in the *F* element within RCE MUL 20 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB20GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB21GF Instruction

The *LB21GF* instruction is used to configure the data in the *F* element within RCE MUL 21. The configuration of the data in the *F* element within RCE MUL 21 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB21GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB22GF Instruction

The *LB22GF* instruction is used to configure the data in the *F* element within RCE 22. The configuration of the data in the *F* element within RCE MUL 22 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB22GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB23GF Instruction

The *LB23GF* instruction is used to configure the data in the *F* element within RCE MUL 23. The configuration of the data in the *F* element within RCE MUL 23 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB23GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB30GF Instruction

The *LB30GF* instruction is used to configure the data in the *F* element within RCE 30. The configuration of the data in the *F* element within RCE MUL 30 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB30GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB31GF Instruction

The *LB31GF* instruction is used to configure the data in the *F* element within RCE MUL 31. The configuration of the data in the *F* element within RCE MUL 31 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB31GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB32GF Instruction

The *LB32GF* instruction is used to configure the data in the *F* element within RCE 32. The configuration of the data in the *F* element within RCE MUL 32 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB32GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

LB33GF Instruction

The *LB33GF* instruction is used to configure the data in the *F* element within RCE MUL 33. The configuration of the data in the *F* element within RCE MUL 33 is the same as the configuration of the data in the *F* element within RCE 00 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *LB33GF* instead of *LB00GF* at the beginning of the assembly code block — refer to Tables B.13 and B.14 for usage information and a configuration example.

XBAR1 Instruction

The *XBAR1* instruction is used to configure Byte-Wise Crossbar 1 as detailed in Table B.15. The example shown in Table B.16 illustrates the usage of the *XBAR1* instruction. When programming Byte-Wise Crossbar 1, each output byte is a mapping from an input byte. The 128-bit block is divided from left to right, with the left-most byte denoted as *O15* down to the right-most byte denoted as *O0.* An input byte (one of sixteen, denoted as *I15* down to *I0*) may be mapped to multiple output bytes. In the example, each input byte is mapped one-to-one to an output byte, resulting in Byte-Wise Crossbar 1 being configured as a pass-through element.

Element	**Operand 1**
O15	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O14	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O13	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O12	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O11	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O10	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O9	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O8	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O7	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O6	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O5	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O4	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O3	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O2	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O1	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0
O0	I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0

Table B.15: XBAR1 assembly instruction — usage

```
XBAR1
O15     <   I15
O14     <   I14
O13     <   I13
O12     <   I12
O11     <   I11
O10     <   I10
O9      <   I9
O8      <   I8
O7      <   I7
O6      <   I6
O5      <   I5
O4      <   I4
O3      <   I3
O2      <   I2
O1      <   I1
O0      <   I0
END
```

Table B.16: XBAR1 assembly instruction — example

XBAR2 Instruction

The *XBAR2* instruction is used to configure Byte-Wise Crossbar 2. The configuration of Byte-Wise Crossbar 2 is the same as the configuration of Byte-Wise Crossbar 1 and configuration via the assembly language is the same for both instructions with the only difference being the specification of *XBAR2* instead of *XBAR1* at the beginning of the assembly code block — refer to Tables B.15 and B.16 for usage information and a configuration example.

IMUX Instruction

The *IMUX* instruction is used to configure the Input Multiplexors as detailed in Table B.17. The example shown in Table B.18 illustrates the usage of the *IMUX* instruction. The Input

Multiplexors control the flow of data through the COBRA architecture. When a multiplexor is set to *IN* it passes the registered incoming data from the external inputs into the first row of the datapath. When a multiplexor is set to *REG* it passes data from the Whitening Blocks into the first row of the datapath. Note that Table B.19 details the data flow from each Input Multiplexor to its corresponding RCE or RCE MUL.

Element	**Operand 1**
SW1	IN,REG
SW2	IN,REG
SW3	IN,REG
SW4	IN,REG

Table B.17: IMUX assembly instruction — usage

```
IMUX
SW1    <   IN
SW2    <   IN
SW3    <   IN
SW4    <   IN
END
```

Table B.18: IMUX assembly instruction — example

Input Multiplexor	Element Receiving Data from the Input Multiplexor
SW1	RCE 00
SW2	RCE MUL 01
SW3	RCE 02
SW4	RCE MUL 03

Table B.19: IMUX data flow

ERADR Instruction

The *ERADR* instruction is used to configure the address registers for the eRAMs located on the right side of the COBRA architecture (eRAMs one through eight) as detailed in Table B.20. The example shown in Table B.21 illustrates the usage of the *ERADR* instruction.

Element	Operand 1
RM8	8-Bit Hexadecimal Value
RM7	8-Bit Hexadecimal Value
RM6	8-Bit Hexadecimal Value
RM5	8-Bit Hexadecimal Value
RM4	8-Bit Hexadecimal Value
RM3	8-Bit Hexadecimal Value
RM2	8-Bit Hexadecimal Value
RM1	8-Bit Hexadecimal Value

Table B.20: ERADR assembly instruction — usage

```
ERADR
RM8    <  16
RM7    <  13
RM4    <  1C
RM1    <  0F
END
```

Table B.21: ERADR assembly instruction — example

ERADL Instruction

The *ERADL* instruction is used to configure the address registers for the eRAMs located on the right side of the COBRA architecture (eRAMs nine through sixteen) as detailed in Table B.22. The example shown in Table B.23 illustrates the usage of the *ERADL* instruction.

Element	**Operand 1**
RM16	8-Bit Hexadecimal Value
RM15	8-Bit Hexadecimal Value
RM14	8-Bit Hexadecimal Value
RM13	8-Bit Hexadecimal Value
RM12	8-Bit Hexadecimal Value
RM11	8-Bit Hexadecimal Value
RM10	8-Bit Hexadecimal Value
RM9	8-Bit Hexadecimal Value

Table B.22: ERADL assembly instruction — usage

ERADL		
RM16	<	16
RM15	<	13
RM12	<	1C
RM9	<	0F
END		

Table B.23: ERADL assembly instruction — example

ERDIS Instruction

The *ERDIS* instruction is used to configure the input data source for each of the eRAMs as detailed in Table B.24. Note that the values *IN[4:1]* refer to the unregistered external inputs to the COBRA architecture. The example shown in Table B.25 illustrates the usage of the *ERDIS* instruction.

Element	**Operand 1**
RM16	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN1
RM15	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN1
RM14	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN1
RM13	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN1
RM12	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN2
RM11	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN2
RM10	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN2
RM9	LB00,LB01,LB10,LB11,LB20,LB21,LB30,LB31,IN2
RM8	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN4
RM7	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN4
RM6	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN4
RM5	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN4
RM4	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN3
RM3	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN3
RM2	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN3
RM1	LB02,LB03,LB12,LB13,LB22,LB23,LB32,LB33,IN3

Table B.24: ERDIS assembly instruction — usage

```
ERDIS
RM16   <   LB00
RM15   <   LB10
RM14   <   LB21
RM13   <   IN1
RM12   <   IN2
RM8    <   IN4
END
```

Table B.25: ERDIS assembly instruction — example

ERLIS Instruction

The *ERLIS* instruction is used to configure the source of the eRAM input for each of the RCEs and RCE MULs as detailed in Table B.26. The example shown in Table B.27 illustrates the usage of the *ERLIS* instruction.

Element	**Operand 1**
LB00	RM[16:9]
LB01	RM[16:9]
LB02	RM[8:1]
LB03	RM[8:1]
LB10	RM[16:9]
LB11	RM[16:9]
LB12	RM[8:1]
LB13	RM[8:1]
LB20	RM[16:9]
LB21	RM[16:9]
LB22	RM[8:1]
LB23	RM[8:1]
LB30	RM[16:9]
LB31	RM[16:9]
LB32	RM[8:1]
LB33	RM[8:1]

Table B.26: ERLIS assembly instruction — usage

ERLIS		
LB00	<	RM16
LB01	<	RM15
LB02	<	RM8
LB03	<	RM7
LB10	<	RM14
LB11	<	RM13
LB12	<	RM6
LB13	<	RM5
END		

Table B.27: ERLIS assembly instruction — example

ERSTB Instruction

The *ERSTB* instruction is used to strobe data into the eRAMs as detailed in Table B.28. Note that to properly store data into an eRAM, an *ERSTB* instruction must be used to set the eRAM's write strobe to *ON* and then another *ERSTB* instruction must be used to set the eRAM's write strobe to *OFF*. This sequence of instructions should be executed with no other instructions between them. The example shown in Table B.29 illustrates the usage of the *ERSTB* instruction.

Element	**Operand 1**
RM[16:1]	ON,OFF

Table B.28: ERSTB assembly instruction — usage

```
ERSTB
RM16    <   ON
RM15    <   ON
RM14    <   ON
RM13    <   ON
RM12    <   ON
RM8     <   ON
END
ERSTB
RM16    <   OFF
RM15    <   OFF
RM14    <   OFF
RM13    <   OFF
RM12    <   OFF
RM8     <   OFF
END
```

Table B.29: ERSTB assembly instruction — example

LBEN Instruction

The *LBEN* instruction is used to enable or disable the output registers for the RCEs, RCE MULs, and Whitening Blocks as detailed in Table B.30. This instruction is required when operating in an *overfull* instruction cycle as described in Section 6.8.14. The example shown in Table B.31 illustrates the usage of the *LBEN* instruction.

Element	**Operand 1**
LB00	TRUE,FALSE
LB01	TRUE,FALSE
LB02	TRUE,FALSE
LB03	TRUE,FALSE
LB10	TRUE,FALSE
LB11	TRUE,FALSE
LB12	TRUE,FALSE
LB13	TRUE,FALSE
LB20	TRUE,FALSE
LB21	TRUE,FALSE
LB22	TRUE,FALSE
LB23	TRUE,FALSE
LB30	TRUE,FALSE
LB31	TRUE,FALSE
LB32	TRUE,FALSE
LB33	TRUE,FALSE
W30	TRUE,FALSE
W31	TRUE,FALSE
W32	TRUE,FALSE
W33	TRUE,FALSE

Table B.30: LBEN assembly instruction — usage

```
LBEN
LB00  <  FALSE
LB01  <  FALSE
LB02  <  FALSE
LB03  <  FALSE
LB10  <  FALSE
LB11  <  FALSE
LB12  <  FALSE
LB13  <  FALSE
LB20  <  FALSE
LB21  <  FALSE
LB22  <  FALSE
LB23  <  FALSE
LB30  <  FALSE
LB31  <  FALSE
LB32  <  FALSE
LB33  <  FALSE
W30   <  FALSE
W31   <  FALSE
W32   <  FALSE
W33   <  FALSE
END
```

Table B.31: LBEN assembly instruction — example

WPARAM Instruction

The *WPARAM* instruction is used to configure the Whitening Blocks as detailed in Table B.32. The example shown in Table B.33 illustrates the usage of the *WPARAM* instruction. Elements may be specified for configuration in any order and the *END* statement indicates that configuration is complete.

Element	Operand 1	Operand 2
W30	REG,COMB	XOR,ADD,IN
W31	REG,COMB	XOR,ADD,IN
W32	REG,COMB	XOR,ADD,IN
W33	REG,COMB	XOR,ADD,IN

Table B.32: WPARAM assembly instruction — usage

```
WPARAM
W30        <   COMB   IN
W31        <   REG    XOR
W32        <   COMB   ADD
W33        <   COMB   IN
END
```

Table B.33: WPARAM assembly instruction — example

W30 Instruction

The *W30* instruction is used to load Whitening Block 30 with the data output from RCE 30. The example shown in Table B.34 illustrates the usage of the *W30* instruction.

```
W30
```

Table B.34: W30 assembly instruction — example

W31 Instruction

The *W31* instruction is used to load Whitening Block 31 with the data output from RCE MUL 31. The example shown in Table B.35 illustrates the usage of the *W31* instruction.

W31

Table B.35: W31 assembly instruction — example

W32 Instruction

The *W32* instruction is used to load Whitening Block 32 with the data output from RCE 32. The example shown in Table B.36 illustrates the usage of the *W32* instruction.

W32

Table B.36: W32 assembly instruction — example

W33 Instruction

The *W33* instruction is used to load Whitening Block 33 with the data output from RCE MUL 33. The example shown in Table B.37 illustrates the usage of the *W33* instruction.

W33

Table B.37: W33 assembly instruction — example

FLAG Instruction

The *FLAG* instruction is used to configure the Flag register as detailed in Table B.38. The example shown in Table B.39 illustrates the usage of the *FLAG* instruction. Elements may be specified for configuration in any order and the *END* statement indicates that configuration is complete. Note that the *DONE* element refers to the *idle* flag detailed in Section 6.8.15 and that a flag will default to *FALSE* if not specified.

Element	Operand 1
RDY	TRUE,FALSE
VLD	TRUE,FALSE
FL0	TRUE,FALSE
FL1	TRUE,FALSE
DONE	TRUE,FALSE
FL3	TRUE,FALSE
FL4	TRUE,FALSE
FL5	TRUE,FALSE

Table B.38: FLAG assembly instruction — usage

```
FLAG
RDY    <  TRUE
FL0    <  FALSE
DONE   <  TRUE
END
```

Table B.39: FLAG assembly instruction — example

JMP Instruction

The *JMP* instruction is used to specify the address in the iRAM to set the program counter to so as to begin another encryption or decryption operation as detailed in Table B.40. The address is one less than the location at which the system waits for the *go* input signal to become active as the program counter will be automatically incremented by one following the jump operation (see Section 6.8.16 for more details on configuring the jump address). The example shown in Table B.41 illustrates the usage of the *JMP* instruction.

Element	Operand 1
JMP	12-Bit Hexadecimal Value

Table B.40: JMP assembly instruction — usage

JMP	<	A30

Table B.41: JMP assembly instruction — example

NOP Instruction

The *NOP* instruction is used to fill empty instruction cycles when operating in an *underfull* instruction cycle and requires no other operands.

Comments

Comments may be inserted into the assembly code using the | character. Comments must be enclosed by the | character and have at least one character (white space is acceptable) between the delineators. The following example illustrates the usage of comments.

```
| Load eRAMs 16-13 with w_-8 |
```

B.3 Input File Format

The assembly language input file must conform to the language constructs detailed in Section B.2. Carriage returns may not be inserted as white space. White space in the form of carriage returns must be implemented as comments with no information between the | characters. Finally, the *JMP* instruction must be the final instruction in the assembly code as there is no halt instruction. The *JMP* instruction allows for looping and the system is halted by deactivating the *go* input and allowing the current encryption/decryption to complete, halting the COBRA architecture at the jump address.

Bibliography

[1] Advanced Encryption Standard. At http://www.nist.gov/aes. The Advanced Encryption Standard (AES) ongoing development effort should finish sometime in the year 2001.

[2] DES Modes of Operation, FIPS, Federal Information Processing Standard, Pub No. 81. Available at http://csrc.nist.gov/fips/change81.ps, December 1980.

[3] TURBOchannel Hardware Specification. Technical Report EK-369AA-OD-007B, Digital Equipment Corporation, 1991.

[4] Federal Register: Vol. 62, Num. 177. Available at http://www.nist.gov/aes, September 12 1997.

[5] The Nielsen NetRatings Reporter. World Wide Web, June 20 1999.
`http://www.nielsen-netratings.com/weekly.html`

[6] Recommendation for Block Cipher Modes of Operation, NIST Special Publication SP 800-38A. Available at http://csrc.nist.gov/publications/nistpubs/index.html, December 2001.

[7] C. Adams. The CAST-256 Encryption Algorithm. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.

[8] P. Alfke. Xilinx M1 Timing Parameters. Electronic Mail Personal Correspondance, December 1999.

[9] C. Alippi, W. Fornaciari, L. Pozzi, M. Sami, and P. L. Da Vinci. Determining the Optimum Extended Instruction-Set Architecture for Application Specific Reconfigurable VLIW CPUs. In *Proceedings of the 12th International Workshop on Rapid System Prototyping, RSP 2001*, pages 50–56, Monterey, California, USA, June 25–27 2001.

[10] Altera Corporation, San Jose, California, USA. *Data Book*, 1999.

[11] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac — Configurable Custom Computing. In *Third Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '95*, pages 32–38, Napa Valley, California, USA, April 19-21 1995. IEEE, Inc.

[12] R. Amerson and P. Kuekes. A Twenty-Seven Chip MCM-C. In *Proceedings of the THird International Conference on Multichip Modules*, pages 578–582, Denver, Colorado, USA, April 13-15 1994.

[13] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.

[14] K. Aoki and H. Lipmaa. Fast Implementations of AES Candidates. In *The Third Advanced Encryption Standard Candidate Conference*, pages 106–122, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[15] K. Aoki and H. Ueda. Optimized Software Implementations of E2. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[16] J. M. Arnold, D. A. Buell, D. T. Hoang, D. V. Pryor, N. Shirazi, and M. R. Thistle. The Splash 2 Processor and Applications. In *Proceedings of the 1993 IEEE International Conference on Computer Design ICCD'93*, pages 482–485, October 3–6 1993.

[17] L. Bassham, III. Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard. In *The Third Advanced*

Encryption Standard Candidate Conference, pages 136–148, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[18] K. E. Batcher. Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, C-29:836–840, September 1980.

[19] O. Baudrom, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Nguyen, F. Noilhan, D. Pointchebval, T. Pornin, G. Poupard, J. Stern, and S. Vaudenay. Report on the AES Candidates. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[20] J. Becker and M. Glesner. A Parallel Dynamically Reconfigurable Architecture Designed for Flexible Application-Tailored Hardware/Software Systems in Future Mobile Communication. *The Journal of Supercomputing*, 19(1):105–127, 2001.

[21] J. Becker, T. Pionteck, C. Habermann, and M. Glesner. Design and Implementation of a Coarse-Grained Dynamically Reconfigurable Hardware Architecture. In *Proceedings IEEE Computer Society Workshop on VLSI 2001. Emerging Technologies for VLSI Systems*, pages 41–46, Orlando, Florida, USA, April 19–20 2001.

[22] E. Biham. A Fast New DES Implementation in Software. In *Fourth International Workshop on Fast Software Encryption*, volume LNCS 1267, pages 260–272, Berlin, Germany, 1997. Springer-Verlag.

[23] E. Biham. A Note on Comparing the AES Candidates. In *Proceedings: The Second AES Conference (AES2)*, pages 85–92, March 22–23 1999.

[24] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology — CRYPTO '90*, volume LNCS 537, pages 2–21, Berlin, Germany, 1991. Springer-Verlag.

[25] M. Blaze. High-Bandwidth Encryption with Low-Bandwidth Smartcards. In D. Gollmann, editor, *Third International Workshop on Fast Software Encryption*, volume

LNCS 1039, pages 33–40, Berlin, Germany, 1996. Springer-Verlag. Conference Location: Cambridge, UK.

[26] K. Bondalapati and V. K. Prasanna. Reconfigurable Computing: Architectures, Models and Algorithms. *Current Science*, 78(7):828–837, 2000.

[27] K. K. Bondalapati. *Modeling and Mapping for Dynamically Reconfigurable Hybrid Architectures*. PhD thesis, University of Southern California, Los Angeles, California, USA, August 2001.

[28] H. Bonnenberg, A. Curiger, N. Felber, H. Kaeslin, R. Zimmermann, and W. Fichtner. VINCI: Secure Test of a VLSI High-Speed Encryption System. In *Proceedings: International Test Conference*, pages 782–790, October 1993. IEEE Cat Num: 93CH3356-3 ISBN: 0-7803-1430-1.

[29] S. Brown and J. Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design & Test of Computers*, 13(2):42–57, 1996.

[30] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, April 2000.

[31] D. C. Chen and J. M. Rabaey. A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

[32] B. Chetwynd. Universal Block Cipher Module: Towards a Generalized Architectures for Block Ciphers. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, November 1999.

[33] O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong. Tradeoffs in Parallel and Serial Implementations of the International Data Encryption Algorithm IDEA. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, Paris, France, May 14–16 2001. Springer-Verlag.

[34] S. Contini, R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin. The Security of the RC6^{TM} Block Cipher, v1.0. Available at http://www.rsa.com/rsalabs/aes/security.pdf, August 20, 1998. RSA Laboratories.

[35] Triscend Corporation. Configurable System-On-Chip Home Page. http://www.triscend.com.

[36] J. Daemen, L. Knudsen, and V. Rijmen. The Block Cipher SQUARE. In *Fourth International Workshop on Fast Software Encryption*, volume LNCS 1267, pages 149–165, Berlin, Germany, 1997. Springer-Verlag.

[37] J. Daemen and V. Rijmen. AES Proposal: Rijndael. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.

[38] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim. A Comparative Study of Performance of AES Final Candidates Using FPGAs. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, Worcester, Massachusetts, USA, August 2000. Springer-Verlag.

[39] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim. An Adaptive Cryptographic Engine for IPSec Architectures. In K. L. Pocek and J. M. Arnold, editors, *Eighth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '00*, pages 122–131, April 2000.

[40] M. Davio, Y. Desmedt, J. Goubert, F. Hoornaert, and J. J. Quisquater. Efficient Hardware and Software Implementations for the DES. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology — CRYPTO '84*, volume LNCS 196, pages 144–146, Berlin, Germany, 1985. Springer-Verlag.

[41] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In D. A. Buell and K. L. Pocek, editors, *Second Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '94*, pages 31–39, Napa Valley, California, USA, April 10-13 1994. IEEE, Inc.

[42] A. DeHon. DPGA Utilization and Application. In *FPGA '96 - ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 115–121, Monterey, CA, USA, February 11–13 2000. ACM.

[43] R. Doud. Hardware Crypto Solutions Boost VPN. *Electronic Engineering Times*, (1056):57–64, April 12 1999.

[44] J. Dray. NIST Performance Analysis of the Final Round Java^{TM} AES Candidates. In *The Third Advanced Encryption Standard Candidate Conference*, pages 149–160, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[45] H. Eberle. A High-speed DES Implementation for Network Applications. In E. F. Brickell, editor, *Advances in Cryptology — CRYPTO '92*, volume LNCS 740, pages 521–539, Berlin, Germany, August 16–20 1993. Springer-Verlag. Conference Location: Santa Barbara, California, USA.

[46] H. Eberle and C. P. Thacker. A 1 Gbit/second GaAs DES Chip. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 19.7.1–19.7.4, Boston, Massachusetts, USA, May 3–6 1992. IEEE, Inc.

[47] A. Elbirt. An FPGA Implementation and Performance Evaluation of the CAST-256 Block Cipher. Technical Report, Cryptography and Information Security Group, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 1999.

[48] A. Elbirt and C. Paar. An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher. In *FPGA '00 - ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–40, Monterey, CA, USA, February 2000. ACM.

[49] A. Elbirt and C. Paar. An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):545–557, August 2001.

[50] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. In *The Third Advanced Encryption Standard Candidate Conference*, pages 13–27, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[51] J. G. Eldredge and B. L. Hutchings. Density Enchancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In D. A. Buell and K. L. Pocek, editors, *Second Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '94*, pages 180–188, Napa Valley, California, USA, April 10-13 1994. IEEE, Inc.

[52] J. P. Hayes et. al. A Microprocessor-Based Hypercube Supercomputer. *IEEE Micro*, 6(5):6–17, October 1986.

[53] H. Feistel. Cryptography and Computer Privacy. *Scientific American*, 228(5):15–23, May 1973.

[54] V. Fischer and M. Drutarovsky. Two Methods of Rijndael Implementation in Reconfigurable Hardware. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, Paris, France, May 14–16 2001. Springer-Verlag.

[55] A. Folmsbee. AES Java^{TM} Technology Comparisons. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[56] A. Freier, P. Karlton, and P. Kocker. *The SSL Protocol, Version 3.0.* Netscape Communications Corporation, Mountain View, California, USA, March 1996.

[57] K. Gaj and P. Chodowiec. Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware. In *The Third Advanced Encryption Standard Candidate Conference*, pages 40–54, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[58] B. Gladman. Implementation Experience with AES Candidate Algorithms. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[59] B. Gladman. AES Algorithm Efficiency. World Wide Web, 2000.
`http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes2/`

[60] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *Computer*, 24(1):81–89, January 1991.

[61] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer*, 33(4):70–77, April 2000.

[62] J. Goodman and A. P. Chandrakasan. An Energy-Efficient Reconfigurable Public-Key Cryptography Processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[63] L. Granboulan. AES Timings of Best Known Implementations. World Wide Web.
`http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html`

[64] F. Gurkaynak. COBRA Atomic Elements — Timing and Area Estimates. Electronic Mail Personal Correspondance, January 2001.

[65] H. Bonnenberg and A. Curiger and N. Felber and H. Kaelsin and X. Lai. VLSI Implementation of a New Block Cipher. In *Proceedings of the IEEE International Conference on Computer Design*, pages 510–513, Los Alamitos, California, USA, 1991. IEEE Computer Society Press.

[66] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky. *Computer Organization*. McGraw-Hill Publishing Company, New York, New York, USA, 3rd edition, 1978.

[67] S. Hauck and G. Borriello. Pin Assignment for Multi-FPGA Systems. In D. A. Buell and K. L. Pocek, editors, *Second Annual IEEE Symposium on Field-Programmable*

Custom Computing Machines, FCCM '94, pages 11–13, Napa Valley, California, USA, April 10-13 1994. IEEE, Inc.

[68] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera Reconfigurable Function Unit. In *Fifth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '97*, pages 87–96, Napa Valley, California, USA, April 16–18 1997.

[69] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor With A Reconfigurable Coprocessor. In *Fifth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '97*, Napa Valley, California, USA, April 16–18 1997.

[70] J. P. Hayes. *Computer Architecture and Organization*. McGraw-Hill Publishing Company, New York, New York, USA, 2nd edition, 1978.

[71] R. W. Hockney and C. R. Jesshope. *Parallel COmputers*. Adam Hilger Ltd., Bristol, England, 1981.

[72] E. Hong, J. H. Chung, and C. H. Lim. Hardware Design and Performance Estimation of the 128-bit Block Cipher CRYPTON. In Ç. Koç and C. Paar, editors, *Proceedings: Workshop on Cryptographic Hardware and Embedded Systems*, volume LNCS 1717, pages 49–60, Worcester, Massachusetts, USA, August 1999. Springer-Verlag.

[73] A. Huang and E. H. Kim. ReRISC: A Reconfigurable Reduced Instruction Set Computer. Technical report, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1998. `http://web.mit.edu/bunnie/www`

[74] K. Huber and S. Wolter. Telekom's MAGENTA Algorithm for En-/Descryption in the Gigabit/sec Range. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3233–3235, New York, New York, USA, 1996. IEEE, Inc.

[75] T. Ichikawa, T. Kasuya, and M. Matsui. Hardware Evaluation of the AES Finalists. In *The Third Advanced Encryption Standard Candidate Conference*, pages 279–285,

New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[76] H. Ito, R. Konishi, H. Nakada, K. Oguri, A. Nagoya, N. Imlig, K. Nagami, T. Shiozawa, and M. Inamori. Dynamically Reconfigurable Logic LSI-PCA-1. In *2001 Symposium on VLSI Circuits*, pages 103–106, Kyoto, Japan, June 14–16 2001.

[77] A. K. Jones and E. F. Gehringer (eds.). The Cm* Multiprocessor Project: A Research Review. Technical Report CMU-CS-80-131, Carnegie-Mellon University, Department of Computer Science, July 1980.

[78] J. Kaps and C. Paar. Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine. In S. Tavares and H. Meijer, editors, *Fifth Annual Workshop on Selected Areas in Cryptography*, volume LNCS 1556, Berlin, Germany, August 1998. Springer-Verlag. Conference Location: Queen's University, Kingston, Ontario, Canada.

[79] J. P. Kaps. High Speed FPGA Architectures for the Data Encryption Standard. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 1998.

[80] J.-P. Kaps and C. Paar. DES auf FPGAs (DES on FPGAs, in German). *Datenschutz und Datensicherheit*, 23(10):565–569, 1999. invited contribution.

[81] B. Kastrup, A. Bink, and J. Hoggerbrugge. ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '99*, pages 92–101, Napa Valley, California, USA, April 21-23 1999. IEEE, Inc.

[82] G. Keating. Analysis of AES Candidates On the 6805 CPU Core. World Wide Web, 1999.
`http://members.ozemail.com.au/~geoffk/aes-6805/`

[83] G. Keating. Performance Analysis of AES candidates on the 6805 CPU Core. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[84] S. Kent and R. Atkinson. *RFC 2401: Security Architecture for the Internet Protocol.* Corporation for National Research Initiatives, Internet Engineering Task Force, Network Working Group, Reston, Virginia, USA, November 1998.

[85] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, K. Imlig, T. Shiozawa, M. Inamori, and K. Nagami. PCA-1: A Fully Asynchronous, Self-Reconfigurable LSI. In *Proceedings of the Seventh International Symposium on Asynchronous Circuits and Systems, ASYNC 2001*, pages 54–61, Salt Lake City, Utah, USA, March 11–14 2001.

[86] H. Kuo and I. Verbauwhede. Architectural Optimization for a 1.82 Gbit/sec VLSI Implementation of the AES Rijndael Algorithm. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, Paris, France, May 14–16 2001. Springer-Verlag.

[87] M. Kwan. The Design of the ICE Encryption Algorithm. In *Fourth International Workshop on Fast Software Encryption*, volume LNCS 1267, pages 69–82, Berlin, Germany, 1997. Springer-Verlag.

[88] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. In Ivan B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume LNCS 473, pages 389–404, Berlin, Germany, May 1990. Springer-Verlag.

[89] X. Lai and Y. Massey. Markov Ciphers and Differential Cryptoanalysis. In D. W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, volume LNCS 547, Berlin, Germany, 1991. Springer-Verlag.

[90] J. Leonard and W.H. Magione-Smith. A Case Study of Partially Evaluated Hardware Circuits: Keyspecific DES. In W. Luk, P.Y.K. Cheung, and M. Glesner, editors, *Seventh International Workshop on Field-Programmable Logic and Applications, FPL '97*, London, UK, September 1–3 1997. Springer-Verlag.

[91] M. P. Leong, O. Y. H. Cheung, K. H. Tsoi, and P. H. W. Leong. A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA. In K. L. Pocek and J. M. Arnold, editors, *Eighth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '00*, pages 122–131, April 2000.

[92] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading, Massachusetts, USA, 1983.

[93] H. Lipmaa. C Implementations of the AES Candidates on the UltraSparc. World Wide Web.
`http://www.tml.hut.fi/~helger/aes/table.html`

[94] H. Lipmaa. AES Candidates: A Survey of Implementations. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[95] Lucent Technologies, Allentown, Pennsylvania, USA. *Field Programmable Gate Arrays Data Book*, 1999.

[96] L. Maliniak. Multiplexing Enhances Hardware Emulation. *Electronic Design*, 40:76–78, November 1992.

[97] M. Matsiu. New Block Encryption Algorithm MISTY. In *Fourth International Workshop on Fast Software Encryption*, volume LNCS 1267, Berlin, Germany, 1997. Springer-Verlag.

[98] Gael Hachëz, François Koeune, and Jean-Jacques Quisquater. cAESar results: Implementation of Four AES Candidates on Two Smart Cards. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[99] M. McLoone and J. McCanny. High Performance Single-Chip FPGA Rijndael Algorithm. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, Paris, France, May 14–16 2001. Springer-Verlag.

[100] O. Mencer, M. Morf, and M. J. Flynn. Hardware Software Tri-Design of Encryption for Mobile Communication Units. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3045–3048, New York, New York, USA, May 1998. IEEE. Conference Location: Seattle, Washington, USA.

[101] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, USA, 1997.

[102] Annapolis Microsystems. Reconfigurable Computing Home Page. http://www.annapmicro.com.

[103] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Fourth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '96*, pages 157–166, Napa Valley, California, USA, April 17–19 1996. IEEE, Inc.

[104] E. Mosanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez. CryptoBooster: A Reconfigurable and Modular Crytographic Coprocessor. In Ç. Koç and C. Paar, editors, *Proceedings: Workshop on Cryptographic Hardware and Embedded Systems*, volume LNCS 1717, pages 246–256, Worcester, Massachusetts, USA, August 1999. Springer-Verlag.

[105] J. Nakajima and M. Matsui. Fast Software Implementations of MISTY1 on Alpha Processors. *IEICE Transactions on Fundamentals of Electronics, Communications and COmputer Sciences*, E82-A(1):107–116, 1999.

[106] National Institute of Standards and Technology (NIST). *Second Advanced Encryption Standard (AES) Conference*, Rome, Italy, March 1999.

[107] J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, and E. Roback. Status Report on the First Round of the Development of The Advanced Encryption Standard. Available at http://csrc.nist.gov/encryption/aes/round1/r1report.pdf, August 9 1999.

[108] P. Bora and T. Czajka. Implementation of the Serpent Algorithm Using Altera FPGA Devices. World Wide Web, 1999.
`http://csrc.nist.gov/encryption/aes/round2/pubcmnts.htm`

[109] P. Mroczkowski. Implementation of the Block Cipher Rijndael Using Altera FPGA. World Wide Web, 1999.
`http://csrc.nist.gov/encryption/aes/round2/pubcmnts.htm`

[110] C. Paar. Optimized Arithmetic for Reed-Solomon Encoders. In *1997 IEEE International Symposium on Information Theory*, page 250, Ulm, Germany, June 29–July 4 1997.

[111] C. Paar, B. Chetwynd, T. Connor, S. Deng, and S. Marchant. An Algorithm-Agile Cryptographic Co-Processor Board on FPGAs. In J. Schewel, P. Athanas, S. Guccione, S. Ludwig, and J. McHenry, editors, *The SPIE's Symposium on Voice, Video, and Data Communications*, volume 3844, Boston, Massachusetts, USA, September 19–22 1999. SPIE - The International Society for Optical Engineering.

[112] C. Patterson. A Dynamic Implementation of the Serpent Block Cipher. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, pages 142–155, Worcester, Massachusetts, USA, August 17–18 2000. Springer-Verlag.

[113] C. Patterson. High Performance DES Encryption in Virtex^{TM} FPGAs Using JBits^{TM}. In K. L. Pocek and J. M. Arnold, editors, *Eighth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '00*, pages 113–121, April 2000.

[114] M. Peyravian and D. Coppersmith. A Structured Symmetric-Key Block Cipher. *Computers & Security*, 18(2):134–147, 1999.

[115] C. Phillips and K. Hodor. Breaking the 10k FPGA Barrier Calls For an ASIC-Like Design Style. *Integrated System Design*, 1996.

[116] R. Razdan and M. Smith. A High-Performance Microarchitecture With Hardware-Programmable Functional Units. In *MICRO-27*, pages 172–180, November 1994.

[117] M. Riaz and H. Heys. The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms. In *Proceedings: IEEE 1999 Canadian Conference on Electrical and Computer Engineering*, Edmonton, Alberta, Canada, March 1999.

[118] V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win. The Cipher SHARK. In D. Gollmann, editor, *Third International Workshop on Fast Software Encryption*, volume LNCS 1039, Berlin, Germany, 1996. Springer-Verlag. Conference Location: Cambridge, UK.

[119] R. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6$^{\text{TM}}$ Block Cipher. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.

[120] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *Sixth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '98*, pages 28–37, Napa Valley, California, USA, April 15-17 1998. IEEE, Inc.

[121] S. L. C. Salomao, V. C. Alves, and E. M. C. Filho. HiPCrypto: A High-Performance VLSI Cryptographic Chip. In *Proceedings of the Eleventh Annual IEEE International ASIC Conference*, pages 7–11, Rochester, New York, USA, September 1998.

[122] F. Sano, M. Koike, S. Kawamura, and M. Shiba. Performance Evaluation of AES Finalists on the High-End Smart Card. In *The Third Advanced Encryption Standard Candidate Conference*, pages 82–93, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[123] G. Sassatelli, G. Cambon, J. Galy, and L. Torres. A Dynamically Reconfigurable Architecture for Embedded Systems. In *Proceedings of the 12th International Workshop on Rapid System Prototyping, RSP 2001*, pages 32–37, Monterey, California, USA, June 25–27 2001.

[124] A. Satoh, N. Ooba, K. Takano, and E. D'Avignon. High-Speed MARS Hardware. In *The Third Advanced Encryption Standard Candidate Conference*, pages 279–285, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[125] B. Schneier. *Applied Cryptography.* John Wiley & Sons Inc., New York, New York, USA, 2nd edition, 1996.

[126] B. Schneier and J. Kelsey. Unbalanced Feistel Networks and Block Cipher Design. In D. Gollmann, editor, *Third International Workshop on Fast Software Encryption*, volume LNCS 1039, Berlin, Germany, 1996. Springer-Verlag. Conference Location: Cambridge, UK.

[127] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: A 128-Bit Block Cipher. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.

[128] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Performance Comparison of the AES Submissions. In *Proceedings: Second AES Candidate Conference (AES2)*, Rome, Italy, March 1999.

[129] A. Schubert and W. Anheier. Efficient VLSI Implementation of Modern Symmetric Block Ciphers. In *Proceedings: IEEE International Conference on Electronics, Circuits and Systems (ICECS'99)*, volume 2, pages 757–760, Pafos, Cyprus, September 5–8 1999. IEEE, Inc.

[130] A. Schubert, V. Meyer, and W. Anheier. Reusable Cryptographic VLSI Core Based on the SAFER K-128 Algorithm with 251.8 Mbit/s Throughput. In *Proceedings: IEEE Workshop on Signal Processing Systems (SiPS)*, pages 437–446, New York, New York, USA, 1998. IEEE, Inc.

[131] C. E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 27(4):656–715, 1949.

[132] T. Shimoyama, S. Amada, and S. Moriai. Improved Fast Software Implementation of Block Ciphers. In *Information and Communications Security, First International Conference, ICIS'97*, pages 269–273, 1997.

[133] H. J. Siegel, J. B. Armstrong, and D. W. Watson. Mapping Computer-Vision-Related Tasks onto Reconfigurable Parallel-Processing Systems. *IEEE Computer Society: Computer*, 25(2):54–63, February 1992.

[134] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[135] W. Stallings. *Network and Internetwork Security – Principles and Practice.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1995.

[136] A. Sterbenz and P. Lipp. Performance of the AES Candidate Algorithms in Java^{TM}. In *The Third Advanced Encryption Standard Candidate Conference*, pages 161–168, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[137] J. Stern and S. Vaudenay. CS-Cipher. In *Fifth International Workshop on Fast Software Encryption*, Berlin, Germany, March 1998. Springer-Verlag.

[138] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995.

[139] R. Taylor and S. Goldstein. A High-Performance Flexible Architecture for Cryptography. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 1999*, volume LNCS 1717, pages 231–245, Worcester, Massachusetts, USA, August 1999. Springer-Verlag.

[140] S. Trimberger, R. Pang, and A. Singh. A 12 Gbps DES Encryptor/Decryptor Core in an FPGA. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, pages 156–163, Worcester, Massachusetts, USA, August 17–18 2000. Springer-Verlag.

[141] V. Fischer. Realization of Round 2 AES Candidates Using Altera FPGA. World Wide Web, 2000.
`http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`

[142] J. Varghese, M. Butts, and J. Batcheller. An Efficient Logic Emulation System. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):171–174, June 1993.

[143] I. Verbauwhede, F. Hoornaert, J. Vandewalle, and H. De Man. Security Considerations in the Design and Implementation of a New DES Chip. In D. Chaum and W. L. Price, editors, *Advances in Cryptology - EUROCRYPT '87*, volume LNCS 304, pages 287–300, Berlin, Germany, 1987. Springer-Verlag.

[144] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, Mar 1996.

[145] N. Weaver and J. Wawrzynek. A Comparison of the AES Candidates Amenability to FPGA Implemenation. In *The Third Advanced Encryption Standard Candidate Conference*, pages 28–39, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[146] B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms. In *The Third Advanced Encryption Standard Candidate Conference*, pages 286–304, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[147] R. Weis and S. Lucks. The Performance of Modern Block Ciphers in Java^{TM}. In *Smart Card Research and Applications Third International Conference, CARDIS'98*,

volume LNCS 1820, pages 125–133, Louvain-la-Neuve, Belgium, September 14–16 1998. Springer-Verlag.

[148] R. Weiss and N. Binkert. A Comparison of AES Candidates on the Alpha 21264. In *The Third Advanced Encryption Standard Candidate Conference*, pages 75–81, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[149] D. Wheeler and R. Needham. TEA, A Tiny Encryption Algorithm. In *Second International Workshop on Fast Software Encryption*, Berlin, Germany, 1995. Springer-Verlag.

[150] D. C. Wilcox, L. Pierson, P. Robertson, E. Witzke, and K. Gass. A DES ASIC Suitable for Network Encryption at 10 Gbps and Beyond. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 1999*, volume LNCS 1717, pages 37–48, Worcester, Massachusetts, USA, August 1999. Springer-Verlag.

[151] M. J. Wirthlin and B. L. Hutchings. A Dynamic Instruction Set Computer. In *Third Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '95*, pages 99–107, Napa Valley, California, USA, April 19-21 1995. IEEE, Inc.

[152] M. J. Wirthlin and B. L. Hutchings. DISC: The Dynamic Instruction Set Computer. In J. Schewel, editor, *Proceedings of Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, volume 2607, pages 92–103, Philadelphia, Pennsylvania, USA, October 25–26 1995. SPIE - The International Society for Optical Engineering.

[153] R. Wittig and P. Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *Fourth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '96*, 1996.

[154] A. Wolfe and J. P. Shen. Flexible Processors: A Promising Application-Specific Processor Design Approach. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture — MICRO '21*, pages 30–39, San Diego, California, USA, November 1988.

[155] T. Wollinger, M. Wang, J. Guajardo, and C. Paar. How Well Are High-End DSPs Suited for the AES Algorithms? In *The Third Advanced Encryption Standard Candidate Conference*, pages 94–105, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[156] S. Wolter, H. Matz, A. Schubert, and R. Laur. On the VLSI Implementation of the International Data Encryption Algorithm IDEA. In *IEEE Symposium on Circuits and Systems*, volume 1, pages 397–400, New York, New York, USA, 1995. IEEE, Inc.

[157] J. Woodfill and B. Von Herzen. Real-Time Stereo Vision on the PARTS Reconfigurable Computer. In *Fifth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '97*, pages 201–210, Napa Valley, California, USA, April 16–18 1997. IEEE, Inc.

[158] J. Worley, B. Worley, T. Christian, and C. Worley. AES Finalists on PA-RISC and IA-64: Implementations & Performance. In *The Third Advanced Encryption Standard Candidate Conference*, pages 57–74, New York, New York, USA, April 13–14 2000. National Institute of Standards and Technology.

[159] Xilinx Inc., San Jose, California, USA. *Virtex^{TM} 2.5V Field Programmable Gate Arrays*, 1998.

[160] Xilinx, Inc., San Jose, California, USA. *The Programmable Logic Data Book*, 1999.

[161] A. K. Yeung and J. M. Rabaey. A 2.4 GOPS Data-Driven Reconfigurable Multiprocessor IC for DSP. In *Proceedings of the 1995 IEEE International Solid-State Circuits Conference*, pages 108–109, 346, San Francisco, California, USA, February 15–17 1995.

[162] A. M. Youssef, S. Mister, and S. E. Tavares. On the Design of Linear Transformations for Substitution Permutation Encryption Networks. In *Fourth Annual Workshop on Selected Areas in Cryptography*, pages 40–48, Berlin, Germany, 1997. Springer-Verlag. Conference Location: School of Computer Science, Carleton University.

[163] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey. A 1V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications. In *Proceedings of the 2000 IEEE International Solid-State Circuits Conference*, pages 68–69, 448, San Francisco, California, USA, February 7–9 2000.

[164] R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner. A 177 Mb/s VLSI Implementation of the International Data Encryption Algorithm. *IEEE Journal of Solid-State Circuits*, 29(3):303–307, March 1994.