An Instruction-Level Distributed Processor for Symmetric-Key Cryptography

Adam J. Elbirt, *Member*, *IEEE*, and Christof Paar, *Member*, *IEEE*

Abstract—Efficient implementation of block ciphers is critical toward achieving both high security and high-speed processing. Numerous block ciphers have been proposed and implemented, using a wide and varied range of functional operations. Existing architectures such as microcontrollers do not provide this broad range of support. Therefore, we will present a hardware architecture that achieves efficient block cipher implementation while maintaining flexibility through reconfiguration. In an effort to achieve such a hardware architecture, a study of a wide range of block ciphers was undertaken to develop an understanding of the functional requirements of each algorithm. This study led to the development of COBRA, a reconfigurable architecture for the efficient implementation of block ciphers. A detailed discussion of the top-level architecture, interconnection scheme, and underlying elements of the architecture will be provided. System configuration and on-the-fly reconfiguration will be analyzed, and from this analysis, it will be demonstrated that the COBRA architecture satisfies the requirements for achieving efficient implementation of a wide range of block ciphers that meet the 622 Mbps ATM network encryption throughput requirement.

Index Terms—Cryptography, algorithm-agility, FPGA, block cipher, VHDL.

1 Introduction

With more than 100 million Americans connected to the Internet [1], information security has become a top priority. Many applications—electronic mail, electronic banking, medical databases, and electronic commerce—require the exchange of private information. For example, when engaging in electronic commerce, customers provide credit card numbers when purchasing products. If the connection is not secure, an attacker can easily obtain this sensitive data.

In order to implement a comprehensive security plan for a given network and, thus, guarantee the security of a connection, the following services must be provided [2], [3], [4]:

- *Confidentiality*: Information cannot be observed by an unauthorized party. This is accomplished via public-key and private-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 hash functions and Message Authentication Codes (MACs).
- *Authentication*: Parties within a given communication session must provide certifiable proof of their identity. This is accomplished via the use of digital signatures.
- *Nonrepudiation*: Neither the sender nor the receiver of a message may deny transmission. This is accomplished via digital signatures and third party notary services.

Cryptographic algorithms used to insure confidentiality fall 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 [3]. 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 directly follows the 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 of 622 million bits per second (Mbps). 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 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 [3], [5]. Conversely, cryptographic algorithms that are implemented in hardware are by nature more physi-

- *A.J. Elbirt is with the Electrical and Computer Engineering Department, University of Massachusetts Lowell, Lowell, MA 01854. E-mail: Adam_Elbirt@umc.edu.*
- *C. Paar is with the Department of Electrical Engineering and Information Sciences, Ruhr-University Bochum, Universitaetsstr. 150, 44780 Bochum, Germany. E-mail: cpaar@crypto.rub.de.*

Manuscript received 20 Mar. 2003; revised 31 May 2004; accepted 31 Aug. 2004; published online 22 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0039-0303.

cally secure as they cannot easily be read or modified by an outside attacker when the key is stored in special memory internal to the device [5]. As a result, the attacker does not have easy access to the key storage area and cannot discover or alter its value in a straightforward manner [3].

When using a general-purpose processor, even the fastest software implementations of block ciphers cannot satisfy the required data rates for bulk data encryption for high-end applications [6], [7], [8], [9], [10]. As a result, hardware implementations are necessary for block ciphers to achieve this required performance level. Although traditional hardware implementations lack flexibility with respect to algorithm and parameter switching, reconfigurable hardware devices offer a promising alternative for the implementation of block ciphers. One of the potential advantages of block ciphers implemented in reconfigurable hardware is algorithm agility, 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 whose use is negotiated on a per-session basis. Whereas algorithm agility can be very costly with traditional hardware, algorithm agility through reconfigurable hardware appears to be an attractive possibility [11], [12]. It is also conceivable that fielded devices will be upgraded with a new encryption algorithm that 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 is complete [13]. Moreover, applications exist which require modification of a standardized algorithm, e.g., by using proprietary S-Boxes or permutations. Such modifications are easily made with reconfigurable hardware. Although typically slower than Application Specific Integrated Circuit (ASIC) implementations, reconfigurable implementations have the potential of running substantially faster then software implementations. 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).

What follows is a brief overview of previous block cipher implementations in reconfigurable hardware. An examination of a wide range of block ciphers will lead to a set of requirements for a reconfigurable architecture designed to achieve efficient block cipher implementations. We will then present our proposed solution, the COBRA reconfigurable architecture.

2 Block Cipher Background

Many block ciphers may be characterized as Feistel networks [3]. Feistel networks were invented by Feistel [14] and are a general method of transforming a function into a permutation. The basic Feistel network divides the data into two halves where one half operates upon the other [15]. The *f*-function uses one of the halves of the data block and a key to create a pseudorandom bit stream that is used to encrypt or decrypt the other half of the data block.

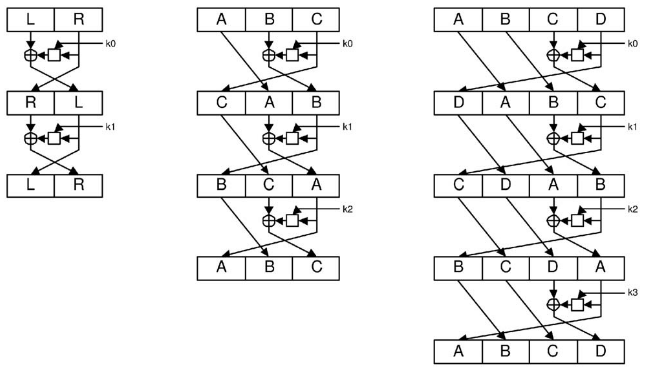

Fig. 1. Block diagram for standard block ciphers.

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 data swap as a circular right shift. This allows for the use of the same *f*-function, but requires multiple rounds to input all of the subblocks to the *f*-function [16]. Fig. 1 from [16] details the block diagram for block ciphers employing both the basic Feistel network and generalized Feistel networks of three and four blocks. The *f*-function is represented by the 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 [17]. *Confusion* obscures the relationship between the plaintext, the ciphertext, 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 [3]. 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 Fig. 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:

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

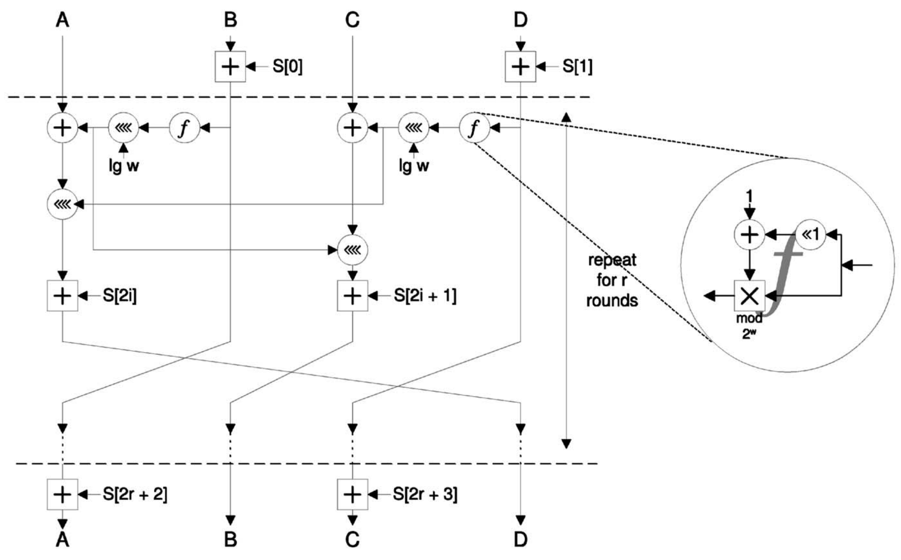

Fig. 2. Block diagram of an RC6 encryption round.

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 Section 4. Having established a basic background in the structure of block ciphers, Section 3 presents existing implementations of block ciphers in hardware.

3 Previous Reconfigurable Hardware Implementations

The most frequently implemented block cipher algorithms are DES and the AES candidates. The following summarizes block cipher implementations in FPGAs and other reconfigurable logic. There are also a number of reconfigurable hardware-based implementations of public-key algorithms, many of which target elliptic curve cryptosystems due to the significantly reduced operand size as compared to other public-key algorithms, such as RSA. However, a discussion of these implementations is beyond the scope of this investigation.

3.1 FPGA Implementations

The first published implementation of DES in an FPGA achieved a throughput of 26.4 Mbps [18]. However, this implementation required generation of key-specific circuitry for the target Xilinx XC4013-14, requiring recompilation of the implementation for each key. Until recently, the best throughput an FPGA implementation of DES has achieved is 402.7 Mbps when operating in Electronic Code Book (ECB) mode using a Xilinx XC4028EX-3 FPGA [19]. This implementation employed pipeline design techniques to maximize the system clock frequency at the cost of pipeline latency cycles.

With the advent of runtime reconfiguration and more technologically advanced FPGAs, a recent DES implementation achieved a throughput of 10.752 Gbps when operating in ECB mode using a Xilinx Virtex XCV150-6 FPGA [12]. The use of the Xilinx runtime reconfiguration software application JBits™ allowed for real-time key-specific compilation of the bitstream used to program the FPGA, resulting in a smaller and faster design (which operated at 168 MHz) as compared to the design in [19] (which operated at 25.19 MHz). However, this implementation requires an interface between the FPGA and a host computer. Most recently, a DES implementation achieved a throughput of 12 Gbps when operating in ECB mode using a Xilinx Virtex-E XCV300E-8 FPGA [20], but did not make use of runtime reconfiguration.

Multiple FPGA implementation studies have been presented for the AES candidate algorithm finalists [21], [22], [23], [24], [25], the results of which are found in Table 1. Note that feedback modes and nonfeedback modes of operation are denoted as *FB* and *NFB*, respectively, in Table 1 and that no throughput results are presented in [25]. The studies performed in [22] and [24] used a Xilinx Virtex XCV1000 as the target FPGA. The study performed in [21] used the Xilinx Virtex family of FPGAs but did not specify the target device. Finally, the study performed in [23] used the Altera Flex EPF10K130EQ-1 as the target FPGA.

FPGA implementations of individual candidate algorithm finalists have also been performed. An RC6 implementation achieved a throughput of 37 Mbps using a Xilinx XC4020XV-9 [26]. A Serpent implementation using a Xilinx Virtex

TABLE 1
AES Finalists FPGA Implementation Studies

Alg	NFB Mode Throughput (Mbps) [24]	NFB Mode Throughput (Mbps) [22]	FB Mode Throughput (Mbps) [22]
MARS	●	●	●
RC6	13100	2400	126.5
Rijndael	12200	1940	300.1
Serpent	16800	5040	444.2
Twofish	15200	2400	127.7

Alg	FB Mode Throughput (Mbps) [21]	FB Mode Throughput (Mbps) [24]	FB Mode Throughput (Mbps) [23]
MARS	101.88	61.0	●
RC6	112.87	142.7	●
Rijndael	353.00	414.2	232.7
Serpent	148.95	431.4	125.5
Twofish	173.06	177.3	81.5

XCV1000-4 achieved a throughput of 4.86 Gbps [27]. 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 runtime reconfiguration software application JBits™ which allowed for real-time key-specific compilation of the bit-stream used to program the FPGA [11]. This runtime reconfiguration resulted in a smaller and faster design (which operated at 137.15 MHz) as compared to the design in [27] (which operated at 37.97 MHz), although a host computer interface is required. When implemented using an FPGA from the Altera Flex 10KA family, the Serpent algorithm achieved a maximum throughput of 301 Mbps [28]. However, it is important to note that the implementation in [28] implements eight of the Serpent algorithm's thirty-two rounds while the implementations in [11] and [27] implement all of the rounds of the Serpent algorithm. Note that all of the presented throughput values are for nonfeedback modes of operation.

Multiple implementations of Rijndael, chosen by NIST as the AES algorithm, have been presented using both Xilinx and Altera FPGAs. The implementation in [29] achieves a throughput of 6.956 Gbps using Xilinx Virtex-E XCV3200E-8. Utilizing ROM to implement the ByteSub operation resulted in a significant increase in throughput and decrease in area as compared to implementations in [21], [22], [24]. Rijndael implementations have achieved throughputs of 268 Mbps when targeting the Altera Flex EPF10K250 [30] and throughputs ranging from 570 Mbps to 964 Mbps when targeting the more advanced Altera APEX 20KE200-1, depending on the implementation methodology [31].

3.2 Other Reconfigurable Implementations

A number of reconfigurable architectures have been proposed to accelerate the performance of symmetric-key cryptography. Hybrid architectures composed of a microprocessor core combined with reconfigurable function blocks are typically used to accelerate the performance of a general-purpose processor. 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 [32], but often causes the delay associated with communication between the microprocessor and the reconfigurable logic block to become the bottleneck within the system [33]. This overhead can be reduced by caching multiple configurations within the reconfigurable function blocks at the cost of more expensive and less flexible hardware [34], [35], [36]. Hybrid architectures targeted at accelerating symmetric-key cryptography include ConCISe, Garp, and MorphoSys [35], [37], [38].

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. The ConCISe, CryptoManiac, Garp, MorphoSys, and PipeRench platforms are examples of generalized reconfigurable architectures targeted at accelerating symmetric-key cryptography [35], [37], [38], [39], [40]. Block cipher implementations on these platforms all achieve a significant performance improvement as compared to general-purpose processor implementations. However, none of these implementations match the throughput of equivalent FPGA implementations, falling short of the the 622 Mbps ATM network encryption throughput requirement.

Given that PipeRench is a specialized reconfigurable architecture, an examination of its architecture provides useful insights into designing a specialized reconfigurable architecture targeting symmetric-key algorithms. PipeRench 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 its 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 [39], [41]. Systems that require feedback, as is the case with most block ciphers (operating in CBC, CFB, or OFB modes) [15], are difficult to realize within the PipeRench architecture.

4 The COBRA Architecture

When designing the ***Cryptographic*** (***Optimized*** for ***Block*** Ciphers) ***Reconfigurable Architecture*** (COBRA) for efficient block cipher implementation, a number of architectural requirements were established to facilitate achieving an optimized solution. 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. In particular, runtime reconfiguration of FPGAs requires either communication with a host computer or large overhead costs that significantly impact performance in stand-alone embedded systems. Therefore, the architecture must be capable of high-speed, on-the-fly reconfiguration. Overhead and off-chip communication are also 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 of the hybrid architecture [34], [38], [42], [43]. Ensuring full support of algorithm-specific operations requires maximizing the functional density of the datapath. This results in the need for a generalized and runtime reconfigurable datapath with functional blocks optimized to efficiently implement the core operations of the target block cipher space [36], [38], [40], [44], [45], [46], [47], [48]. 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

TABLE 2
Occurrence of Block Cipher Atomic Operations

Operation	Occurrences
Boolean	40 of 41
Modular Addition and Subtraction	20 of 41
Fixed Shift	25 of 41
Variable Rotation	10 of 41
Modular Multiplication	7 of 41
Galois Field Multiplication	7 of 41
Modular Inversion	1 of 41
Look-Up Table Substitution	30 of 41

[35], [37], [38], [45], [49], [50]. Finally, an interconnect matrix must also be designed to support both Feistel networks as well as other networks, such as Substitution-Permutation (SP) networks.

An analysis of block ciphers was performed to develop a hardware architecture optimized for block cipher implementation. The analysis was restricted to block ciphers that operate on block sizes of 64 and 128 bits as they are representative of algorithms in use that meet current and expected future security requirements. The block ciphers examined were Blowfish, CAST, CAST-128, CAST-256, CRYPTON, CS-Cipher, DEAL, DES, DFC, E2, FEAL, FROG, GOST, Hasty Pudding, ICE, IDEA, Khafre, Khufu, LOKI91, LOKI97, Lucifer, MacGuffin, MAGENTA, MARS, MISTY1, MISTY2, MMB, RC2, RC5, RC6, Rijndael, SAFER K, SAFER+, Serpent, SQUARE, SHARK, SKIPJACK, TEA, Twofish, WAKE, and WiderWake. The block cipher analysis resulted in a list of operations that a specialized reconfigurable hybrid architecture must support to guarantee efficient implementation over the range of block ciphers studied (a summary of the analysis results is shown in Table 2). It was determined that a reconfigurable cryptographic processor core should support a 32-bit datapath replicated at least four times to allow for the implementation of algorithms requiring either 64-bit or 128-bit block lengths. The system must also support communication between the 32-bit datapaths. Based on the most commonly occurring atomic operations, the following operations should be implemented as part of any reconfigurable cryptographic processor core:

- Bitwise XOR, AND, or OR.
- Addition/subtraction modulo 2^8, 2^{16}, 2^{32}.
- Fix shift/rotation.
- Variable data-dependent rotation.
- 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.

While modular inversion does not occur often enough to merit dedicated hardware resources, such an operation may be implemented using modular exponentiation in combination with Euler's Theorem to calculate the desired inverse.

4.1 Block Diagram and Interconnect

Fig. 3 details the architecture and interconnect of the COBRA architecture. The COBRA architecture allows for distributed processing across a 128-bit datapath via four interconnected 32-bit datapaths (denoted as columns). Each 32-bit datapath interconnects four Reconfigurable Cryptographic Elements (RCEs) which are the primary processing elements within the COBRA architecture. Note that each reconfigurable element is identified by its (row, column) identification number. COBRA hardware resources are managed by the user via microcode to optimize a given block cipher implementation for both speed and efficiency.

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

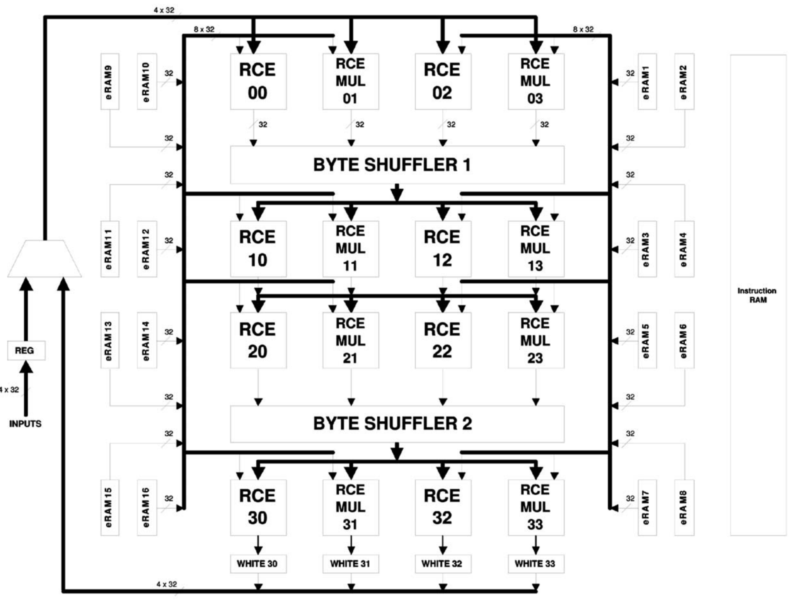

Fig. 3. COBRA architecture and interconnect.

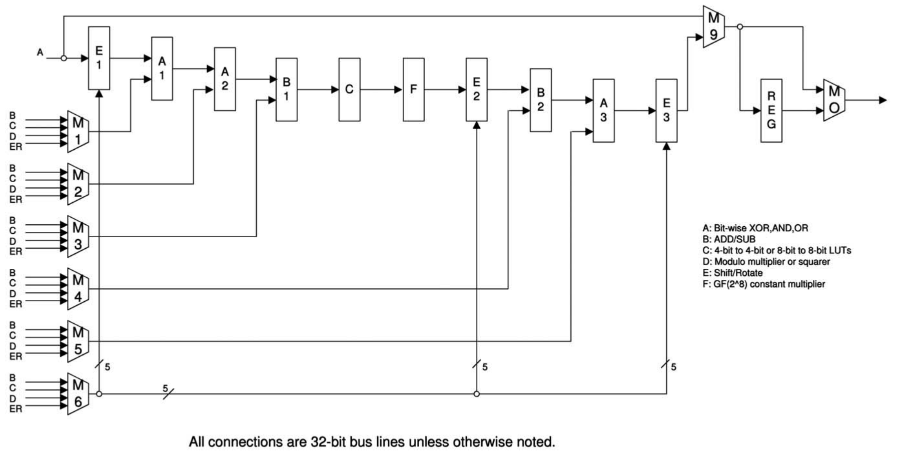
A
B C D ER
M1 M2 M3 M4 M5 M6 M9 MO
E1 A1 A2 B1 C F E2 B2 A3 E3 REG
5
A: Bit-wise XOR,AND,OR
B: ADD/SUB
C: 4-bit to 4-bit or 8-bit to 8-bit LUTs
D: Modulo multiplier or squarer
E: Shift/Rotate
F: GF(2^8) constant multiplier
All connections are 32-bit bus lines unless otherwise noted.

Fig. 4. COBRA RCE.

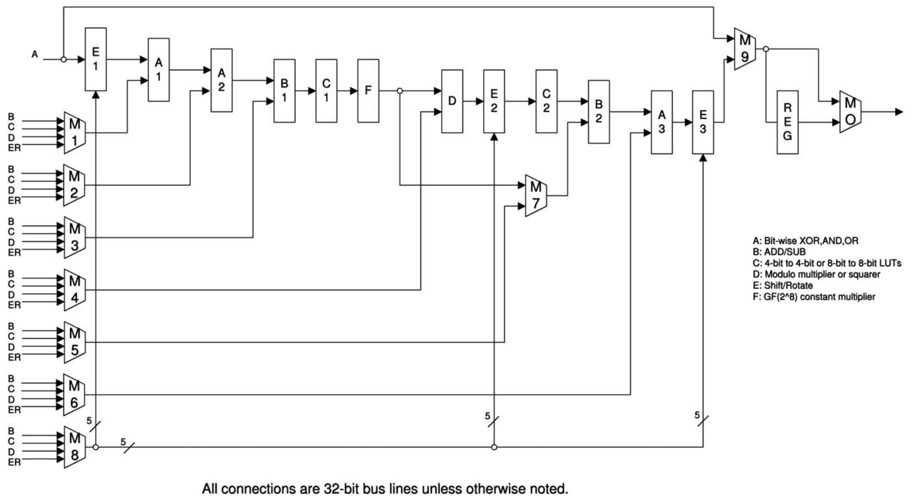
A
B C D ER
M1 M2 M3 M4 M5 M6 M7 M8 M9 MO
E1 A1 A2 B1 C1 F D E2 C2 B2 A3 E3 REG
5
A: Bit-wise XOR,AND,OR
B: ADD/SUB
C: 4-bit to 4-bit or 8-bit to 8-bit LUTs
D: Modulo multiplier or squarer
E: Shift/Rotate
F: GF(2^8) constant multiplier
All connections are 32-bit bus lines unless otherwise noted.

Fig. 5. COBRA RCE with multiplier.

RCE MULs. Each RCE operates upon a 32-bit data stream within a 128-bit block. Two byte shufflers 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 byte shufflers. 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 mod 2^{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. Half of the eRAMs are allocated to columns 0 and 1 while 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, resulting in a corresponding reduction of fabrication costs. Also, note that each RCE or RCE MUL receives the full 128-bit data stream. The 128-bit data stream is partitioned into four 32-bit blocks, with bits 31 to 0 mapping to block 0, bits 63 to 32 mapping to block 1, etc. Block 0 is then denoted as the primary input for all RCE elements in column 0, termed *INA*. Similarly, blocks 1 through 3 map to the primary inputs for RCE or RCE MUL elements in columns 1 through 3, respectively. The other three blocks that make up the remainder of the 128-bit data stream are used as secondary input blocks to the RCE or RCE MUL elements for use in operating upon the primary input block. The secondary input blocks are grouped in ascending order, where the lowest numbered block is termed *INB*, the middle block is termed *INC*, and the highest numbered block is termed *IND*. The eRAM input is termed *INER*.

4.2 RCEs

Figs. 4 and 5 detail the functionality of the RCE and RCE MUL. The elements of the RCE structure are:

- *A*: Bit-wise XOR, AND, or OR.
- *B*: Add or subtract mod $2^8, 2^{16}$, or 2^{32}.
- *C*: Look-Up-Tables (LUTs)—four 8-bit to 8-bit or eight pages of eight 4-bit to 4-bit mappings.
- *D*: Multiply mod 2^{16} or 2^{32} or Square mod 2^{32}.

- *E*: Shift left, shift right, or rotate left. Shift and rotate values may be data dependent.
- *F*: GF (2^8) fixed field constant Galois field multiplier.
- *M*: Multiplexor.
- *REG*: Register.

While the data flow through the elements within the RCE structures is fixed, each element may be selectively disabled via the microcode. To allow for multiple useful and efficient configurations, the location and ordering of the elements within the RCE structures was carefully engineered based on the results of the block cipher study. For example, *E* elements are placed at the front, rear, and middle of the structures to allow for flexibility in manipulation of the data.

When not operating in short mode, the entire 32-bit input to a *C* element 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. To provide secondary inputs to the RCE structure datapath, 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 *REG* element provides a registered version of the input and may be enabled or disabled as needed. Note that the RCE structures may be pipelined by enabling the *REG* element in multiple rows for a particular 32-bit datapath.

The *C* element requires the largest number of instructions for reconfiguration as data must be loaded into each address of the look-up tables. Note that all of the look-up tables may load the data in parallel for a single look-up table address. Therefore, loading the look-up tables in the *C* element requires 16 instructions when operating on 4-bit blocks and 256 instructions when operating on 8-bit blocks. The look-up tables contained in the *F* element are loaded in a similar manner. All other RCE and RCE MUL elements are reconfigured via a single instruction.

4.3 Instruction Mapping

COBRA operates via a Very Long Instruction Word (VLIW) format. Each instruction is 80 bits and the instruction bus is connected to the control mechanisms for the RCEs, RCE MULs, byte shufflers, 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 $\times$ 80-bit memory which supports programs of up to 4,096 total instructions. Instructions are classified as follows:

- *Configure RCE or RCE MUL:*
- *Enable/disable RCE or RCE MUL outputs.*
- *Configure byte shuffler.*
- *Configure input multiplexor.*
- *Configure eRAM:*
- *Control whitening elements:*
- *Read/write flag register.*
- *Jump to specified address.*
- *No operation (NOP).*

The instruction word comprises the *operation code*, *slice address*, *element address*, *LUT address*, and *configuration data* fields. The *operation code* type indicates the operation to be performed. The *operation code* groups similar instructions and use some or all of the *slice address* to indicate the specific COBRA element to be configured. In the case of an RCE or RCE MUL, the *element address* is used to indicate which specific components within the RCE or RCE MUL are to be configured. The *LUT address* is used when configuring either an LUT block or a Galois field fixed field constant multiplier within an RCE or RCE MUL. COBRA elements are configured by instructions that write control words to the associated elements' control registers, allowing for on-the-fly reconfiguration. As an example, to reconfigure the elements within an RCE, an instruction must be executed that contains the control information for each element within the RCE with the individual RCE components' control words forming the *configuration data* field of the instruction.

As an example, consider the AES candidate algorithm finalist RC6 presented in Section 2. We may represent the equation

$$D_{i+1} = [(([(2B_i^2 + B_i) \lll 5] \oplus A_i) \lll ([2D_i^2 + D_i] \lll 5)] \\ + S[2i]$$

as

$$O_1 = [(2B_i^2 + B_i) \lll 5]$$
$$O_2 = [(2D_i^2 + D_i) \lll 5]$$
$$D_{i+1} = [(O_1 \oplus A_i) \lll O_2] + S[2i].$$

If we then assume that the equation for D_{i+1} is to be implemented in RCE MUL 11, the value for A_i is available on the *A* input bus, the value for O_1 is available on the *B* input bus, the value for O_2 is available on the *D* input bus, the value for S[2i] is available on the *ER* input bus, and the final result is to be stored in a 32-bit register, the corresponding COBRA assembly language code would be:

$$\begin{array}{l}
LB11E \\
\quad E1 < \text{NRM IN } 0 \\
\quad A1 < XOR \\
\quad A2 < IN \\
\quad B1 < IN1 \\
\quad C1 < \text{NRM } 8 \text{ IN} \\
\quad F < IN \\
\quad D < IN \\
\quad E2 < \text{OVRD ROL } 0 \\
\quad C2 < \text{NRM } 8 \text{ IN} \\
\quad B2 < ADD32 \\
\quad A3 < IN \\
\quad E3 < \text{NRM IN } 0 \\
\quad M1 < INB \\
\quad M5 < INER \\
\quad M7 < M5 \\
\quad M8 < IND \\
\quad M9 < E3 \\
\quad MO < REG \\
END.
\end{array}$$

In this example, all elements of the RCE MUL are programmed as pass-through elements except *A1*, *E2*, and

B2, which are configured as XOR, left rotation, and 32-bit modular addition elements, respectively. Multiplexor *M1* is configured to pass O_1 to element *A1*, multiplexor *M8* is configured to pass O_2 as the data-dependant rotation value to element *E2*, and multiplexors *M5* and *M7* are configured to pass S[2i] to element *B2*. Multiplexors *M9* and *MO* are configured to store the result of the entire computation in the RCE MUL's 32-bit register.

4.4 Configuration and Control

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, automatically initiating the loading and executing of instructions within the iRAM. Generic flags may be used in the microcode 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 *busy* 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 or decryption operation, 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. COBRA will idle while the *go* signal is inactive.

Reconfiguration is achieved via a dual clocking scheme. The maximum operating frequency of the datapath is 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 iRAM clock is then set to twice the maximum operating frequency based on a worst-case delay analysis performed across these functions. This ensures that the datapath will have settled after loading and executing an instruction given that this process requires two iRAM clock cycles to complete.

To determine the datapath clock frequency, the programmer must determine the optimal number of instructions that to be executed within a datapath clock cycle (the *instruction window*). The programmer must examine the number of *overfull* and *underfull* instruction cycles for a given *instruction window* size. An *overfull* instruction cycle occurs when the number of instructions that are necessary to properly reconfigure the COBRA architecture exceeds the *instruction window* size. An *overfull* instruction cycle is completed by disabling the RCE outputs before the end of the datapath cycle and enabling them when reconfiguration is completed. An *underfull* instruction cycle occurs when the number of instructions that are necessary to properly reconfigure the COBRA architecture is less than the *instruction window* size, requiring the insertion of NOP instructions.

In most applications, the time associated bulk data encryption or decryption far outweighs the time associated with key scheduling. Therefore, to achieve optimal system performance, the programmer must carefully analyze the encryption/decryption segment of the program when choosing the optimal *instruction window* size. This selection is made to minimize the number of *overfull* and *underfull* instruction cycles and usually corresponds to the number of instructions required to reconfigure the datapath to compute the output of the current round of the cipher. It is important to note that the first and last round of a cipher typically require additional overhead instructions to perform operations such as the enabling/disabling of key whitening or the addition/removal of particular functions. Therefore, the programmer must focus on the intermediate rounds when choosing the *instruction window* size, typically selecting the *instruction window* size such that none of the intermediate rounds require an *overfull* instruction cycle. Once the optimal *instruction window* size has been determined, the datapath clock frequency is calculated as $F_{DP} = F_{iRAM}/(2 \times \mathit{instruction\ window\ size})$.

4.5 Software Tools

A number of software tools were developed to support hardware implementations targeting the COBRA architecture. An assembler was created to ease coding while a timing analyzer was created to provide datapath operating frequency information. The COBRA software tool suite is command-line driven and has been coded in C to guarantee portability across all possible software platforms.

The COBRA assembler eliminates the need for the programmer to manually enter bit patterns. Instead, a custom assembly language that supports the full COBRA functionality is interpreted by the assembler to generate a vector file used to configure the COBRA architecture. While not currently available, future versions of COBRA software are expected to support compilation of high-level languages, such as C, into COBRA configuration vector files. Currently, however, users must manually map the block cipher of interest onto the COBRA architecture via the COBRA assembly language.

In conjunction with the COBRA assembler, the COBRA timing analyzer determines maximum operating frequency of the COBRA architecture for a given assembly language program. The timing analyzer evaluates the various configurations of the COBRA architecture as specified by a given program and generates a maximum operating frequency for each configuration as well as for the set of configurations. Given that two iRAM clock cycles are required to load and execute an instruction, the iRAM clock frequency is set by the programmer to be twice that of the maximum operating frequency determined by the timing analyzer.

The timing analyzer creates COBRA mapping structures that contain the delay values for the RCEs, RCE MULs, byte shufflers, and whitening registers. Within the mapping structure, substructures are created for the RCEs and RCE MULs that contain delay values for their subelements. 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.

The timing analyzer maintains a *unique* flag for each mapping structure. A mapping structure is considered unique if a reconfiguration instruction that creates a mapping structure is followed by a runtime 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 runtime 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 unique before performing the system timing analysis.

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 subelements in the substructure associated with the given RCE or RCE MUL. For each of the unique mapping structures, the timing analyzer calculates the worst-case delay for each row of reconfigurable elements within the COBRA architecture 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 register row is calculated for each mapping structure by selecting the delay value of the whitening register that is the longest of the four possible values. The delay values for rows 1 and 3 of each mapping structure are then updated by adding the delay values for the associated byte shuffler. Finally, the delay value for row 3 of each mapping structure is updated by adding the delay value for the whitening register 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 is then calculated. However, the maximum delay value is not simply the sum of the row values, as each reconfigurable element 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 1 and 3 have registered outputs, a delay value must be calculated for the block between row 3 and row 1 (encompassing the whitening registers, row 0, byte shuffler 1, and row 1) and for the block between row 1 and row 3 (encompassing row 2, byte shuffler 2, and row 3). 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 zero or one row of registered outputs, 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 shufflers 1 and 2 and the whitening register row are also listed so that the programmer can determine how much of the delay in rows 1 and 3 are associated with these elements versus the RCEs and RCE MULs. Additionally, the timing analyzer calculates the worst-case system clock period by comparing the maximum delay of each mapping structure. The iRAM clock period is then calculated as half the worst-case system clock period 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.

5 Results

The COBRA architecture was implemented in VHDL using a bottom-up design and test methodology. Key scheduling and encryption were either coded in COBRA assembly language and assembled into microcode or written directly as microcode. System operation was controlled via a VHDL test bench that served to load the iRAM with the microcode, apply the 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.

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. However, it is important to note that all of the block ciphers examined in Section 4 may be implemented targeting the COBRA architecture with varying degrees of efficiency and performance.

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 and their use of both Feistel and SP networks. Given the large number of implementations of DES, this block cipher was also considered for implementation but proved to be poorly suited to the COBRA architecture in its current form. DES requires both an initial and a final permutation, each of which is a bit-wise reordering of the 64-bit data. While bit-wise shifts and rotations are possible, bit-wise permutations are extremely difficult to implement. A 32-bit bit-level shuffler 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

TABLE 3
COBRA Encryption Performance Comparison—Nonfeedback Mode

Algorithm	Rounds	Clock Cycles	Clock Frequency (MHz)	COBRA Throughput (Mbps)	Equivalent FPGA Throughput (Mbps) [22]	Best Software Throughput (Mbps)
RC6	1	145	60.975	53.83	250.0	258.3 [6]
RC6	2	73	60.975	106.92	497.4	258.3 [6]
RC6	4	38	60.975	205.39	891.3	258.3 [6]
RC6	5	30	60.975	260.16	1067.0	258.3 [6]
RC6	10	15	60.975	520.32	2397.9	258.3 [6]
RC6	20	2	60.975	**3902.40**	•	258.3 [6]
Rijndael	1	57	102.041	229.14	294.2	516.1 [51]
Rijndael	2	22	102.041	593.69	575.3	516.1 [51]
Rijndael	5	22	102.041	593.69	1165.8	516.1 [51]
Rijndael	10	9	102.041	**1451.25**	•	516.1 [51]
Serpent	1	273	54.054	25.34	77.0	113.3 [51]
Serpent	8	35	54.054	197.68	1241.6	113.3 [51]
Serpent	16	56	54.054	123.55	•	113.3 [51]
Serpent	32	3	54.054	**2306.30**	5035.0	113.3 [51]

ciphers design and to maintain the coarse-grained nature of the COBRA architecture for the initial design. However, given the large number of currently deployed implementations of DES, addition of the special purpose hardware required for improved performance is an enhancement that may become necessary in future generations of the COBRA architecture.

5.1 Performance Analysis

Table 3 details the performance of the COBRA implementations of RC6, Rijndael, and Serpent when operating in nonfeedback mode. Up to two rounds of RC6, two rounds of Rijndael, or one round of Serpent may be mapped to the COBRA architecture in its current form. Data for implementations with more rounds assumes implementation within an architecture expanded by increasing both the iRAM address space and the number of rows, byte shufflers, and eRAMs with corresponding additions to the instruction set to support configuration of the new hardware elements. Timing estimates for the RCEs and RCE MULs were obtained from Synopsys Design Compiler targeting a 0.35 micron library.

The cycle counts in Table 3 are for pipelined implementations operating on multiple blocks of data in nonfeedback mode where the round function is used as the atomic unit of the pipeline and the pipeline latency is assumed to be met. Note that when all rounds of an algorithm are implemented within the COBRA architecture, the instructions required for on-the-fly reconfiguration are eliminated, resulting in a greatly reduced cycle count for the implementation. Moreover, as evidenced by the 16 round Serpent implementation, it is possible that the cycles required to output the blocks in the pipeline overshadows the performance gain achieved through operating on multiple blocks of data simultaneously. Note that these implementations do not achieve similar performance numbers for modes of operation that require feedback, such as CBC, CFB, or OFB modes. For these modes of operation, the performance of a COBRA implementation will be constant, resulting in throughput results that are two to three times smaller than the best software results. However, it is significant to note that for ATM applications, Counter Mode is typically used and this is a nonfeedback mode of operation.

COBRA performance nears that of equivalent implementations targeting a Xilinx Virtex XCV1000 FPGA [22], with the performance gap decreasing as the number of rounds implemented for a given block cipher increases. Note that the data provided in [24] only details peak performance results and does not include performance data for intermediate degrees of partial pipelining and was therefore not used for this comparison. Also, note that the clock frequencies for COBRA implementations remain constant for each block cipher as the number of rounds increases. These implementations use the block cipher round function as the atomic unit of the pipeline and the implementations do not increase the pipeline depth within the round functions.

The COBRA implementations significantly outperform software implementations as the number of rounds implemented increases. It is important to note that these throughput values are representative of high-end software implementations of the targeted algorithms. While faster general-purpose processors continue to be produced, it is expected that the COBRA architecture will exhibit a similar improvement in performance when implemented targeting a more leading-edge process, such as 0.1 micron or smaller. The throughput data for the RC6 implementation was achieved in [6] when targeting a 450 MHz Pentium II processor while the throughput data for both the Rijndael and Serpent implementations was achieved in [51] when targeting a 500 MHz Itanium processor.

5.2 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 4 summarizes the gate counts for each configurable element within a COBRA RCE or RCE MUL. From Table 4, it is clear that the dominant element in terms of area is the *C* element. Four 128 $\times$ 4 look-up-tables and four 256 $\times$ 8 look-up-tables are implemented within the *C* element, resulting in 10,240 storage bits. The gate count listed for the elements in Table 5 is the total gate count for

TABLE 4
Reconfigurable Element Gate Counts

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	267

TABLE 5
COBRA Architecture Gate Counts

Element	Gates
RCE/RCE MUL Array	2,692,840
Byte Shufflers	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**

all elements of that type, resulting in a total gate count of nearly 6.7 million gates for the COBRA architecture. Note, however, that the gate counts for memory elements are significantly inflated due to the synthesis tool using D flip-flops to implement memory elements as opposed to SRAM blocks. It is estimated that memory element gate counts will decrease by a factor of three should SRAM blocks be used (using an estimate of four gates per SRAM bit [52]), resulting in a total of approximately 2.5 million gates.

The value added by the COBRA architecture versus the FPGA implementations in [22] occurs in the areas of scalability, cost, and design cycle time. The COBRA architecture easily scales up or down by adding rows or columns to the base architecture, as tiling is readily performed at the RCE level. From a cost perspective, while the COBRA architecture requires 2.5 times the logic resources as the FPGA used in [22], the COBRA layout is significantly less complex due to the use of a fixed interconnect. We believe that the use of a fixed interconnect will more than offset the cost of the additional logic resources, resulting in the design and manufacturing costs of the COBRA architecture being far smaller than those associated with the FPGA used in [22]. Finally, the time required to implement and verify algorithms mapped to the COBRA architecture is expected to require significantly less time as compared to the schematic and language-based design methodologies used to target FPGAs due to the assembly language design methodology and the specialized cryptographic elements that COBRA employs.

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 Table 6. As previously noted, up to two rounds of RC6, two round of Rijndael, or one round of Serpent may be mapped to the COBRA architecture in its current form. Gate counts for implementations with more rounds assumes implementation within an architecture expanded by increasing both the iRAM address space and the number of rows, byte shufflers, and eRAMs with corresponding additions to the instruction set to support configuration of the new elements.

In the case of both RC6 and Serpent, 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 implementation of Rijndael and the 16 round implementation 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. However, in the case of Rijndael, the two round implementation exhibits a slightly better CG product versus the 10 round implementation. In this case, the increase in performance gained using a full-length pipeline does not justify the required increase in hardware resources. However, a significant argument in favor of full-length pipeline implementations is that these implementations meet the ATM network encryption throughput requirement of 622 Mbps for all three algorithms.

TABLE 6
COBRA Encryption CG Product

Algorithm	Rounds	Clock Cycles	Gates	CG Product	Normalized CG Product
RC6	1	145	6,691,514	970,269,530	13.477
RC6	2	73	6,691,514	488,480,522	6.785
RC6	4	38	9,544,240	362,681,120	5.038
RC6	5	30	11,197,598	335,927,940	4.666
RC6	10	15	19,464,388	291,965,820	4.055
RC6	20	2	35,997,968	71,995,936	1.000
Rijndael	1	57	6,691,514	381,416,298	2.591
Rijndael	2	22	6,691,514	147,213,308	1.000
Rijndael	5	22	13,970,782	307,357,204	2.088
Rijndael	10	9	27,783,940	250,055,460	1.699
Serpent	1	273	6,691,514	1,826,783,322	5.140
Serpent	8	35	29,736,440	1,040,775,400	2.928
Serpent	16	56	59,315,256	3,321,654,336	9.346
Serpent	32	3	118,472,888	355,418,664	1.000

6 Conclusions

An investigation of block cipher implementations in reconfigurable hardware has been presented and a wide range of block ciphers were examined. This examination led to an understanding of the functionality required to implement these algorithms through the characterization of their key components. This characterization led to a set of requirements used to develop COBRA, an innovative reconfigurable architecture designed to achieve efficient block cipher implementations. A detailed discussion of the top-level architecture, interconnection scheme, and underlying elements of the architecture was provided along with an examination of system configuration and on-the-fly reconfiguration. Algorithms were mapped to the COBRA architecture and implemented using the COBRA assembly language and microcode format. Performance data was gathered in terms of cycle counts to evaluate the implementations of the targeted block ciphers. This evaluation demonstrated that the COBRA architecture achieved efficient implementation of a wide range of block ciphers that meet the 622 Mbps ATM network encryption throughput requirement and approach the performance levels of custom hardware implementations while significantly outperforming software implementations.

Acknowledgments

The authors would like to thank Yusuf Leblebici from the Swiss Federal Institute of Technology for the useful discussions they had with him at the beginning of the COBRA project. This research was supported in part through US National Science Foundation CAREER award #CCR-9733246.

References

[1] "The Nielsen NetRatings Reporter," June 1999, http://www.nielsen-netratings.com/weekly.html.

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

[3] B. Schneier, *Applied Cryptography.* New York: John Wiley & Sons Inc., second ed. 1996.

[4] W. Stallings, *Network and Internetwork Security—Principles and Practice.* Englewood Cliffs, N.J.: Prentice Hall, 1995.

[5] R. Doud, "Hardware Crypto Solutions Boost VPN," *Electronic Eng. Times,* no. 1056, pp. 57-64, Apr. 1999.

[6] K. Aoki and H. Lipmaa, "Fast Implementations of AES Candidates," *Proc. Third Advanced Encryption Standard Candidate Conf.,* pp. 106-122, Apr. 2000.

[7] L. Bassham III, "Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard," *Proc. Third Advanced Encryption Standard Candidate Conf.,* pp. 136-148, Apr. 2000.

[8] J. Dray, "NIST Performance Analysis of the Final Round Java™ AES Candidates," *Proc. Third Advanced Encryption Standard Candidate Conf.,* pp. 149-160, Apr. 2000.

[9] A. Sterbenz and P. Lipp, "Performance of the AES Candidate Algorithms in Java™," *Proc. Third Advanced Encryption Standard Candidate Conf.,* pp. 161-168, Apr. 2000.

[10] T. Wollinger, M. Wang, J. Guajardo, and C. Paar, "How Well Are High-End DSPs Suited for the AES Algorithms?" *Proc. Third Advanced Encryption Standard Candidate Conf.,* pp. 94-105, Apr. 2000.

[11] C. Patterson, "A Dynamic Implementation of the Serpent Block Cipher," *Proc. Workshop Cryptographic Hardware and Embedded Systems—CHES 2000,* pp. 142-155, Aug. 2000.

[12] C. Patterson, "High Performance DES Encryption in Virtex™ FPGAs Using JBits™," *Proc. Eighth Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '00,* pp. 113-121, Apr. 2000.

[13] "Advanced Encryption Standard," http://www.nist.gov/aes, 2004.

[14] H. Feistel, "Cryptography and Computer Privacy," *Scientific Am.,* vol. 228, no. 5, pp. 15-23, May 1973.

[15] B. Schneier and J. Kelsey, "Unbalanced Feistel Networks and Block Cipher Design," *Proc. Third Int'l Workshop Fast Software Encryption,* 1996.

[16] C. Adams, "The CAST-256 Encryption Algorithm," *Proc. First Advanced Encryption Standard (AES) Conf.,* 1998.

[17] C.E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical J.,* vol. 27, no. 4, pp. 656-715, 1949.

[18] J. Leonard and W.H. Mangione-Smith, "A Case Study of Partially Evaluated Hardware Circuits: Keyspecific DES," *Proc. Seventh Int'l Workshop Field-Programmable Logic and Applications, FPL '97,* Sept. 1997.

[19] J.-P. Kaps and C. Paar, "DES auf FPGAs (DES on FPGAs, in German)," *Datenschutz und Datensicherheit,* vol. 23, no. 10, pp. 565-569, 1999.

[20] S. Trimberger, R. Pang, and A. Singh, "A 12 Gbps DES Encryptor/Decryptor Core in an FPGA," *Proc. Workshop Cryptographic Hardware and Embedded Systems—CHES 2000,* pp. 156-163, Aug. 2000.

[21] A. Dandalis, V.K. Prasanna, and J.D. P. Rolim, "A Comparative Study of Performance of AES Final Candidates Using FPGAs," *Proc. Workshop Cryptographic Hardware and Embedded Systems—CHES 2000,* Aug. 2000.

[22] A.J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 9, no. 4, pp. 545-557, Aug. 2001.

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

[24] K. Gaj and P. Chodowiec, "Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays," *Proc. RSA Security Conf.,* Apr. 2001.

[25] N. Weaver and J. Wawrzynek, "A Comparison of the AES Candidates Amenability to FPGA Implemenation," *Proc. Third Advanced Encryption Standard Candidate Conf.,* pp. 28-39, Apr. 2000.

[26] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," *Proc. IEEE 1999 Canadian Conf. Electrical and Computer Eng.,* Mar. 1999.

[27] A.J. Elbirt and C. Paar, "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher," *Proc. FPGA '00-ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays,* pp. 33-40, Feb. 2000.

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

[29] M. McLoone and J. McCanny, "High Performance Single-Chip FPGA Rijndael Algorithm," *Proc. Workshop Cryptographic Hardware and Embedded Systems—CHES 2001,* May 2001.

[30] P. Mroczkowski, "Implementation of the Block Cipher Rijndael Using Altera FPGA," http://csrc.nist.gov/encryption/aes/round2/pubcmnts.htm, 1999.

[31] V. Fischer and M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware," *Proc. Workshop Cryptographic Hardware and Embedded Systems—CHES 2001,* May 2001.

[32] K. Bondalapati and V.K. Prasanna, "Reconfigurable Computing: Architectures, Models and Algorithms," *Current Science,* vol. 78, no. 7, pp. 828-837, 2000.

[33] K.K. Bondalapati, "Modeling and Mapping for Dynamically Reconfigurable Hybrid Architectures," PhD thesis, Univ. of Southern California, Los Angeles, Calif., Aug. 2001.

[34] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Function Unit," *Proc. Fifth Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '97,* pp. 87-96, Apr. 1997.

[35] B. Kastrup, A. Bink, and J. Hoggerbrugge, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator," *Proc. Seventh Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '99,* pp. 92-101, Apr. 1999.

[36] M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer," *Proc. Third Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '95,* pp. 99-107, Apr. 1995.

[37] J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor With A Reconfigurable Coprocessor," *Proc. Fifth Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '97*, Apr. 1997.

[38] 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 Trans. Computers*, vol. 49, no. 5, pp. 465-481, May 2000.

[39] R. Taylor and S. Goldstein, "A High-Performance Flexible Architecture for Cryptography," *Proc. Workshop Cryptographic Hardware and Embedded Systems—CHES 1999*, Aug. 1999.

[40] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," *Proc. 28th Ann. Int'l Symp. Computer Architecture—ISCA 2001*, pp. 110-119, June 2001.

[41] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, no. 4, pp. 70-77, Apr. 2000.

[42] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," *Proc. Sixth Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '98*, pp. 28-37, Apr. 1998.

[43] R. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," *Proc. Fourth Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '96*, 1996.

[44] 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," *Proc. 12th Int'l Workshop Rapid System Prototyping, RSP 2001*, pp. 50-56, June 2001.

[45] D.C. Chen and J.M. Rabaey, "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths," *IEEE J. Solid-State Circuits*, vol. 27, no. 12, pp. 1895-1904, Dec. 1992.

[46] J.G. Eldredge and B.L. Hutchings, "Density Enchancement of a Neural Network Using FPGAs and Run-Time Reconfiguration," *Proc. Second Ann. IEEE Symp. Field-Programmable Custom Computing Machines, FCCM '94*, pp. 180-188, Apr. 1994.

[47] A.K. Yeung and J.M. Rabaey, "A 2.4 GOPS Data-Driven Reconfigurable Multiprocessor IC for DSP," *Proc. 1995 IEEE Int'l Solid-State Circuits Conf.*, pp. 108-109, Feb. 1995.

[48] 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," *Proc. 2000 IEEE Int'l Solid-State Circuits Conf.*, pp. 68-69, Feb. 2000.

[49] G. Sassatelli, G. Cambon, J. Galy, and L. Torres, "A Dynamically Reconfigurable Architecture for Embedded Systems," *Proc. 12th Int'l Workshop Rapid System Prototyping—RSP 2001*, pp. 32-37, June 2001.

[50] A. Wolfe and J.P. Shen, "Flexible Processors: A Promising Application-Specific Processor Design Approach," *Proc. 21st Ann. Workshop Microprogramming and Microarchitecture—MICRO '21*, pp. 30-39, Nov. 1988.

[51] J. Worley, B. Worley, T. Christian, and C. Worley, "AES Finalists on PA-RISC and IA-64: Implementations & Performance," *Proc. Third Advanced Encryption Standard Candidate Conf.*, pp. 57-74, Apr. 2000.

[52] B. Penner, "What Is Gate Count? What Are Gate Count Metrics for Virtex/Spartan-II/4K Devices?," e-mail personal correspondence, Jan. 2003.

Adam J. Elbirt received the BS degree in electrical engineering from Tufts University in 1991, the MEng degree in electrical engineering from Cornell University in 1993, and the PhD degree in electrical engineering from Worcester Polytechnic Institute in 2002. Dr. Elbirt is currently an assistant professor at the University of Massachusetts Lowell and is an active member of the Center for Computer Man/Machine Intelligence, Networking, and Distributed Systems and the Network & Systems Security Laboratory. He is the director of the Information Security Laboratory and the associate director of the Center for Network and Information Security. His research interests include computer architecture, digital system design, real-time and embedded systems, reconfigurable computing, and information security. Prior to his appointment at the University of Massachusetts Lowell, Dr. Elbirt spent nearly 10 years in industry working as a hardware and software developer, embedded systems designer, programmable logic specialist, and field applications engineer. He is a member of the IEEE and the IEEE Computer Society.

Christof Paar received the PhD degree in electrical engineering from the Institute of Experimental Mathematics at the University of Essen, Germany, in 1994. He is the chair for communication security at the University of Bochum, Germany. From 1995-2001, he was a faculty member in the Electrical and Computer Engineering Department of Worcester Polytechnic Institute (WPI), Massachusetts. At WPI, he founded the Cryptography and Information Security Labs. He is cofounder of the Cryptographic Hardware and Embedded Systems (CHES) Workshop series. In 1997, he received a US National Science Foundation CAREER award for research in cryptography and reconfigurable hardware. His research interest include efficient software and hardware algorithms for cryptography, side channel attacks, security in ad hoc networks, and many other aspects of industrial cryptography. He has more than 60 scientific publications in applied cryptography, and he is the editor of six conference proceedings and special journal issues dealing with cryptographic engineering. He is a member of the IEEE and the IEEE Computer Society.