IMPLEMENTATION OF SELECTED CRYPTOGRAPHIC ALGORITHMS ON A RECONFIGURABLE MICROPROCESSOR PLATFORM

ANDREW L. COOK

IMPLEMENTATION OF SELECTED CRYPTOGRAPHIC ALGORITHMS ON A RECONFIGURABLE MICROPROCESSOR PLATFORM

BY

© ANDREW L. COOK

A Thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering

FACULTY OF ENGINEERING AND APPLIED SCIENCE

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

September 2003

St. John's Newfoundland

Abstract

This research was performed to evaluate the cryptographic capabilities of the Chameleon CS2112 Reconfigurable Communications Processor. The CS2112 is a processor architecture which closely couples a general purpose microprocessor with a specialized reconfigurable core.

To evaluate the architecture, five cryptographic algorithms were chosen for implementation. The first algorithm, the Data Encryption Standard (DES), was the United States National Cryptographic Standard from 1977 until 2001 and has been the most widely used cryptographic algorithm in computing and communications environments. DES's successor, the Advanced Encryption Standard (AES or Rijndael) which was chosen in the fall of 2000, was also implemented. Since the CS2112 is targeted toward wireless communications applications the other three algorithms - E0, KASUMI, RC4 - were chosen as they are currently used to provide security in common wireless protocols. The Bluetooth protocol, developed to provide a cheap and easy method for users to create wireless connections between devices, uses E0 to secure connections. The RC4 algorithm is part of the 802.11b wireless data communications standard and KASUMI forms an integral part of the authentication and privacy portions of the 3rd Generation GSM cell phone standard.

DES and AES were fully implemented on the CS2112 and a working executable application was developed. Our efforts to exploit the parallelism and pipelining capability of the CS2112 and multiple implementations are described for these two algorithms. The maximum throughputs for the DES and AES implementations were found to be 322.5 Mbits/sec and 1.1 Gbits/sec respectively. Also, although complete implementations were not finalized, preliminary implementations for E0, KASUMI, and RC4 were developed with a view to allow performance estimates to be made and provide a basis for future work.

The Chameleon CS2112 implementations of the above algorithms performed respectably and the architecture could be useful in cryptographic applications. However, the architecture does constrain design size considerably. Unfortunately, the CS2112 is no longer commercially available since Chameleon Systems Inc. has ceased operations. However, this architecture, with some modifications, could be used as the basis for a new general cryptographic accelerator.

Acknowledgments

This thesis owes its existence to the encouragement, support and inspiration of many people. Firstly, I would like to thank my supervisors, Dr. Howard Heys and Dr. R. Venkatesan, for their guidance and support through the course of my studies and research. Secondly, I would like to thank Chameleon Systems Inc. and Mark Rollins for their financial and technical support. As well, I would like to thank all of my friends and colleagues, most notably Darrell and Jason, for keeping me focused on my work. Last, but not least, I would like to thank my family for their encouragement and support throughout my university years; without them none of this would have been possible.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

3GPP 3rd Generation Partnership Project

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

ARC Argonaut RISC Core

ASIC Application Specific Integrated Circuit

ATM Automated Teller Machine

CBC Cipher Block Chaining

CLB Configurable Logic Block

CLU Control Logic Unit

CSM Control State Memory

DES Data Encryption Standard

DPU Data Path Unit

ECB Electronic Code Book

FPGA Field Programmable Gate Array

FSM Finite State Machine

LFSR Linear Feedback Shift Register

LSM Local Store Memory

LUT	Lookup Table
MUL	Multiplier Unit
MSB	Most Significant Bit
NIST	United States National Institute of Standards and Technology
NSA	United States National Security Agency
OFB	Output Feedback
PIO	Programmable I/O
PLA	Programmable Logic Array
RFU	Reconfigurable Functional Unit
RISC	Reduced Instruction Set Computer
SAGE	Security Algorithms Experts Group
SoC	System On Chip
SRK	Single Round Kernel
SIMD	Single Instruction Multiple Data

Chapter 1

Introduction

I can add colors to the chameleon,

Change shapes with Proteus for advantages,

And set the murderous Machiavel to school.

– William Shakespeare, King Henry VI Pt. III

At the beginning of 2003 there were more than 170 million hosts connected to the Internet and this number is expected to surpass the 200 million mark by the end of the year [1]. However, demand is also growing from a user's perspective, not just for simple Internet connectivity, but also for high bandwidth, permanent connections. Gone are the days of dialing into a pool of modems at your local Internet service provider. Now, high speed data connection technologies offered by telecommunications and cable companies can give people a permanent link to the Internet in their homes making it a part of daily life for many people. This growth has, in turn, fuelled higher bandwidth demands from business. Online shopping and banking have become as commonplace as going to the market to buy bread or going to a bank to pay bills.

Alongside the growing use of the Internet has come another trend - the desire for mobility. The wireless market worldwide is over a 120 billion dollar industry and it is estimated that by 2006 there will be 64 million mobile Internet subscribers [2]. This estimate does not include the users of devices employing technologies such as WiFi (or 802.11b) or Bluetooth that allow easy setup of wireless local area networks in homes and allow printers to be connected to a PC without a cable. The speed and range of wireless devices is also constantly improving as new standards are developed. For example, the 802.11b standard supports data rates of up to 11Mbps whereas the 802.11g standard, which is currently under development, will support data rates of up to 54Mbps at a similar range. Such developments have the potential to make wired connectivity a thing of the past.

A result of these two trends is the need for improved security. Internet users, for instance, worry about hackers gaining access to their credit card information from an online store's database or wonder if somebody connected to their network was observing the purchase they just made. If they are using a wireless connection, a malicious user could be observing their transactions from a great distance using a sensitive antenna. In 1990 CERT [3] recorded only 252 security related incidents on the Internet. But by 2002 this number had swollen to over 80,000 with another 40,000 in only the first quarter of 2003 [4]. Of course, many more incidents were not recorded. Although most new communications protocols now include some mechanism for providing authentication, data integrity and privacy, the demand for more bandwidth requires the development of new cryptographic algorithms and devices that can keep up at higher speeds. As well, wireless connectivity adds the extra complication of low power usage on system designers.

To meet the needs of the market, the communications industry has been changing rapidly over the last two decades. For example, Ethernet connection speeds have moved from 2.94Mbps to now 1Gbps and soon 10Gbps [5]. System designers are now turning to hardware devices more than ever to meet the demanding requirements of the communications industry. To achieve these changes, designers have also been frequently turning to configurable devices, such as Field Programmable Gate Arrays (FPGAs), to improve their time to market and lower their development costs. As well, such devices can be reconfigured as standards and requirements evolve. However, custom Application Specific Integrated Circuits (ASICs) are still used for the highest speed applications where large volume production is expected.

In recent years, researchers have begun coupling a general purpose microprocessor with reconfigurable logic in order to gain the benefits of a hardware implementation while still having the flexibility of software. In a rapidly changing market this is a definite advantage. Initial research into reconfigurable microprocessors began at a university level. Such systems began at the board level with a microprocessor externally linked to an FPGA [6][7]. Over time, however, architectures began to evolve in which the processor and reconfigurable logic resided on the same chip [8][9][10]. As well, such architectures recently began to appear commercially when companies such as Triscend, BOPS and Chameleon Systems released reconfigurable microprocessors to the market. Unfortunately, many of the companies offering reconfigurable microprocessor devices, including BOPS and Chameleon Systems, are no longer operating. A probable reason for their demise was their inability to provide the necessary development tools to easily take advantage of the technology. Although they were able to "add colours to the chameleon, and change shapes with Proteus", it was not enough to become a king.

Motivation, Scope and Organization of Research

In the Fall of 2000, Chameleon Systems Inc. proposed that researchers at Memorial University of Newfoundland, led by Dr. Howard Heys and Dr. R. Venkatesan, study the suitability of their new product for cryptographic algorithms. The Chameleon Systems CS2112 RCP chip was considered the industry's first reconfigurable processor targeted at communications applications. Although designed for protocol processing and signal processing, potential cryptographic capabilities would further its suitability for communications applications. A number of cryptographic algorithms that are part of current communications standards were selected for implementation on the CS2112. In total, five cryptographic algorithms were chosen for implementation, including the Data Encryption Standard (DES), a widely used cryptographic standard released in 1977, and its successor the Advanced Encryption Standard (AES) which was finalized in 2001. As well, the algorithms E0, KASUMI and RC4 which are all utilized in current wireless protocols were chosen since the CS2112 is targeted toward wireless communications applications. The goal was to implement the chosen algorithms in hardware so as to achieve a performance increase over pure software implementations. Chameleon Systems provided Memorial with their proprietary set of design tools as well as a development board for testing the resulting designs. Although the company stopped production of the CS2112 in early 2002, Chameleon Systems still provided technical support as needed. In early 2003, Chameleon Systems Inc. ceased operations but all major research was completed before this occurred.

The following is an outline of the research presented in the following chapters:

- Chapter 2 presents a brief overview of cryptography as well as descriptions of the algorithms selected for implementation on the CS2112.

- Chapter 3 provides some background in the field of reconfigurable computing and some recent results using such hardware for cryptographic purposes.

- Chapter 4 details the Chameleon CS2112 processor architecture and gives the reader insight into the design methodology used when implementing algorithms on the CS2112.

- Chapter 5 describes research efforts related to the implementation of the Data Encryption Standard.

- Chapter 6 presents efforts in the development of Advanced Encryption Standard functions on the CS2112.

- Chapter 7 details some preliminary design work completed with three other ciphers used in current communications standards - the Bluetooth encryption algorithm E0, KASUMI, and RC4.

- Chapter 8 summarizes the results of this research and provides recommendations for future work.

Chapter 2

Cryptography Overview and Selected Algorithm Descriptions

Cryptography, from the Greek *kryptos* meaning hidden and *graphein* meaning to write, is the art and science of making communications unintelligible to all except the intended recipient(s). Cryptographic techniques attempt to protect information by altering its form. The origins of secret writing can be traced back nearly four millennia to the hieroglyphic writing system of the Egyptians [11]. Until recently, the use of cryptographic methods to secure communication has been within the realm of governments and has been directed by their associated national cryptographic services. With the steady growth of the Internet and the ever-increasing private use of communications channels comes the need for public cryptographic standards. Without such standards, users are forced to improvise on their own which can lead to relatively insecure cryptographic methods being employed. The following sections provide a brief overview of some cryptographic principles as well as descriptions of the cryptographic algorithms studied in this research.

2.1 Cryptography

As stated above, cryptography protects information by making it unreadable to all but the authorized parties. Encipherment (E_k) is the process whereby the original text, called the plaintext (P), is replaced by random-looking text called the ciphertext (C). Both texts are composed of a concatenation of symbols from an alphabet. Decipherment (D_k) is the process whereby C is transformed back into the original P. Notationally:

$$P \xrightarrow{E_k} C \xrightarrow{D_k} P$$

A cryptographic system is a family of transformations on plaintexts. The members of the family are indexed by a parameter called the key, k. Typically, the key is a sequence of symbols from an alphabet and the associated transformation E_k (or D_k) is an algorithm determined by k. A key is used since it is much simpler to change a key than to change the entire algorithm used to protect the data.

There are two general classifications for cryptographic algorithms: Public Key and Private (or Symmetric) Key [12]. In public key cryptography, the sender uses a publicly known key generated by the receiver to encrypt a secret message before sending it through an insecure communications channel. The receiver then uses a combination of a private key, known only to them, and the public key to decrypt the message. In general, public key algorithms are much more computationally intensive than private key algorithms since the former rely on the principle that it is infeasible to find the private key given the public key. If otherwise, it would be trivial for an attacker to decode the secret message. One example of such a suspected computationally infeasible problem used by public key algorithms is the factorization of a number composed solely of two very large prime numbers. The best known public key algorithm is RSA which was invented by Ronald L. Rivest, Adi Shamir, and Leonard Adleman in 1977.

Public key systems are typically used to share secret symmetric keys between two communicating parties so that a faster private key system can be used. In private key cryptographic systems, the sender and receiver share a secret key that is used to both encrypt and decrypt secret messages sent on an insecure channel. Private key systems rely on the principle that it is computationally infeasible to decrypt the encrypted message without knowledge of the private key. Two general classifications of private key ciphers, block ciphers and stream ciphers, are discussed in the following sections.

2.1.1 Private Key Block Ciphers

Private-key (also called symmetric-key) block ciphers are probably the most prominent and important elements in many cryptographic systems. Such a cipher is a function that maps n-bit plaintext blocks into n-bit ciphertext blocks. The function depends on a k-bit key that is usually chosen at random from the key space K. Each key could define a different function output for a given plaintext but this is not necessarily the case - multiple keys may give the same ciphertext result. However, to allow for unique encryption the function must be one-to-one. This means that for a specified key, only one ciphertext results from a specified plaintext [13].

Many modes of operation can be applied to any block cipher and are illustrated in Figure 2.1. The $\oplus$ symbol in this figure represents a bitwise exclusive OR (or XOR) operation. In Electronic Code Book (ECB) mode, the blocks are simply encrypted or decrypted one at a time with a key. If the same key is used, each time a particular plaintext is encrypted it will result in the same ciphertext. But with Cipher Block Chaining (CBC) mode this is not the case. In this mode the previous ciphertext output is XORed with the next plaintext to be encrypted before encryption takes place. The first plaintext is XORed with the ciphertext of a non-secret initialization block, c_0. In this way, multiple ciphertexts for a particular plaintext will not necessarily

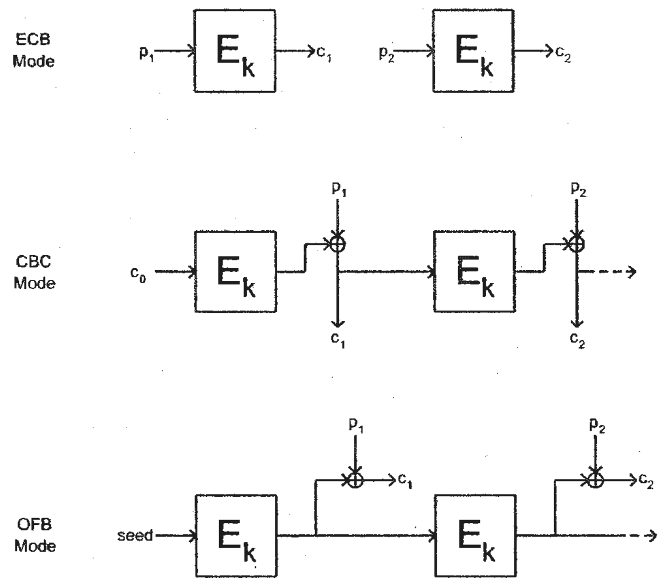

Figure 2.1: Block Cipher Modes of Operation

be equivalent even if the same key is used. Another mode of operation is Output Feedback (OFB) mode in which a pseudo random sequence is produced by the block cipher algorithm. The resulting blocks generated by the algorithm are XORed with the plaintext blocks to produce the ciphertext. Given the block cipher algorithm and a non-secret initialization value or "seed" the pseudo random sequence can be reproduced. In this research only the ECB mode of operation was considered.

2.1.2 Stream Ciphers

In a stream cipher a sequence of bits called the keystream is generated randomly or by some algorithm that generates bits based on an initial seed value and/or previous ciphertext values. This keystream is then combined with the plaintext bits, usually with a simple bitwise XOR operation, to produce the ciphertext as illustrated in Figure 2.2. Decryption is simply the reverse process using the same keystream bits. In general, the most complex portion of a stream cipher system is the keystream generation algorithm. Stream cipher systems are used when it is advantageous to

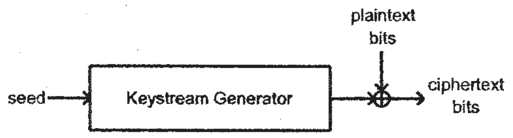

Figure 2.2: Stream Cipher Operation

process data on a bit-by-bit basis where, in such a situation, data would potentially have to be buffered before processing if a block cipher were used. As well, since stream ciphers operate on a bit level, error propagation is limited when compared to block ciphers.

2.2 Data Encryption Standard

In the early 1970s, a banking customer asked IBM to develop a system for encrypting Automated Teller Machine (ATM) data. IBM's team (with some help from the United States National Security Agency (NSA)) developed a new encryption algorithm that was submitted to the United States National Institute of Standards and Technology (NIST). It was adopted in 1977 as a national cryptographic standard: the Data Encryption Standard [14].

DES operates on 64-bit blocks of plaintext and utilizes a 56-bit key. The key is actually supplied to the algorithm as 64-bits, but 8 of these are parity bits. The internal operation of DES is based on two general concepts: product ciphers and Feistel ciphers. In a product cipher, the overall function is composed of a number of simpler operations. Such operations could include bit transpositions, translations, arithmetic operations, modular multiplication and simple substitutions. These simple operations provide no security individually, but together they provide sufficient protection[13]. Fiestel ciphers are a special case of an iterated block cipher in which the ciphertext is calculated by repeated application of the same transformation or "round function".

Figure 2.3 shows the overall structure of DES and, as can be seen, it is composed

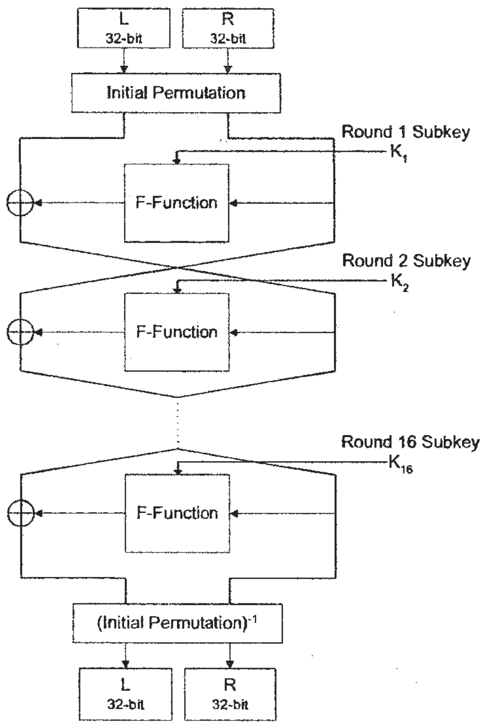

Figure 2.3: DES Structure

of 16 inner rounds. The input block passes through an initial permutation (IP) stage before entering the first round of the cipher and an IP^{-1} stage after round 16. In each round, 32-bits of data enter the F-function. Figure 2.4 illustrates the internals of this function. Inside the F-function, S1-S8 are the 8 substitution mappings, called S-boxes, that map a 6-bit input into a 4-bit output. The 1st and 6th bits of the S-box input are used to select the row of the substitution table while the 2nd to 5th bits are used to select the column of the table. As an example, S-box S1 is given in Table 2.1. From this table, if the input to S1 is 101011 the S-box output would be found in row 3, column 5 which is 1001. As well, in Figure 2.4, E is a fixed

S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Table 2.1: DES S-Box S1

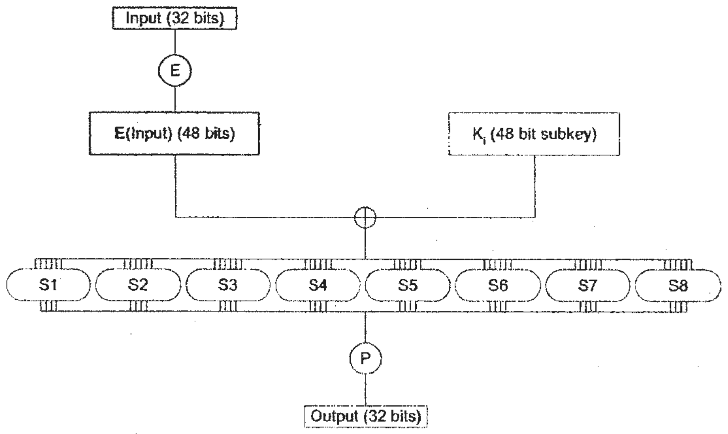

Figure 2.4: DES F-Function

expansion permutation that maps the 32 input bits into 48 bits and P is another permutation over the 32 output bits of the S-Boxes. Tables 2.2 and 2.3 show the outputs of these two operations where the number in a cell represents the bit of the input word that now resides at this location. The tables are read from left to right, top to bottom with the cell (0,0) representing the MSB and cell (8,6) in the E table or (8,4) in the P table representing the LSB of the output. The subkeys K_i, where

E					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Table 2.2: E Expansion

$1 \leq i \leq 16$, are calculated from the original key, K, using a key scheduling algorithm as described in [14]. In the DES algorithm, decryption is accomplished by using the same algorithm with these subkeys in the reverse order.

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Table 2.3: P Permutation

2.3 Advanced Encryption Standard

After a number of years of controversy and successful attempts at breaking DES [15], the United States National Institute of Standards and Technology decided to seek submissions for a new block cipher, the Advanced Encryption Standard, to replace DES. The Rijndael cipher, designed by Vincent Rijmen and Joan Daemen, was chosen as the AES algorithm on October 2, 2000 by NIST [16].

The Rijndael cipher is an iterated block cipher with a variable block length of 128, 192, or 256 bits and a variable key length of 128, 192 or 256 bits. The block and key length are independent of one another. The number of rounds of iteration depends both on the key and block lengths and can be determined from Table 2.4.

	Block Length		
Key Length	128	192	256
128	10	12	14
192	12	12	14
256	14	14	14

Table 2.4: Number of Rounds in Rijndael

A typical round consists of a byte substitution, a column mixing operation, a row shifting operation and a key addition. Each of these steps performs operations on a rectangular array of bytes called the cipher *STATE*. Each column of the *STATE* is 32 bits and the number of columns depends on the block length. Initially, the *STATE* is formed from the input bytes by placing the bytes (from MSB to LSB) into cells $a_{0,0}$,

$a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, etc., where cell $a_{i,j}$ represents the cell of *STATE* a at row i, column j. This arrangement is shown in Figure 2.5. The cipher key is also mapped

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

Figure 2.5: Example of *STATE* for a block size of 128 bits

onto a rectangular array of bytes in the same way. The following sections detail these cipher operations as well as how the produce the overall algorithm.

2.3.1 Byte Substitution

The Rijndael byte substitution operation is a non-linear mapping that is applied to every byte of the cipher *STATE*. The substitution tables, or S-Boxes, operate on 8-bit inputs and produce 8-bit outputs. For the purposes of decryption, the inverse table is used. The S-box mappings can be found in [17].

2.3.2 Row Shift

The row shift operation is a cyclical shift across each of the rows of the cipher *STATE*. The shifts for each row are determined by using Table 2.5 and are a function of the block length. For decryption, the rows are shifted by *(# of columns)-(row shift from Table 2.5)*.

Block Length	Row 0	Row 1	Row 2	Row 3
128	0	1	2	3
192	0	1	2	3
256	0	1	3	4

Table 2.5: Rijndael Row Shifts

2.3.3 Column Mixing

The column mix operation can be pictured as a $GF(2^8)$ multiplication of the columns of $STATE$ with a fixed polynomial $c(x) = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02$. The inverse is found by multiplying the columns of $STATE$ with a fixed polynomial $d(x) = 0B \cdot x^3 + 0D \cdot x^2 + 09 \cdot x + 0E$. Further details of the mathematical principles involved in this operation can be found in the AES specification [17].

2.3.4 Key Addition

At the end of each round, a round subkey is simply bit wise XORed with the $STATE$. The round subkey is derived from the cipher key through a key-scheduling algorithm. In total, the number of round key bits required is equal to the block length multiplied by ($the\ number\ of\ rounds + 1$). Since the key-scheduling algorithm was not implemented, it will not be discussed in this document. Further details of the key scheduling algorithm can be found in the AES specification [17].

2.3.5 Rijndael Operation

Figure 2.6 illustrates a typical Rijndael encryption cycle. As can be seen, the encryption begins with a key addition operation followed by a number of iterations of the "round function" consisting of the byte substitution, shift row, column mixing, and key addition steps described above. The final round of encryption is slightly different from the regular round in that no "mix column" operation is performed. This final round is included in the total number of rounds of iteration found in Table 2.4.

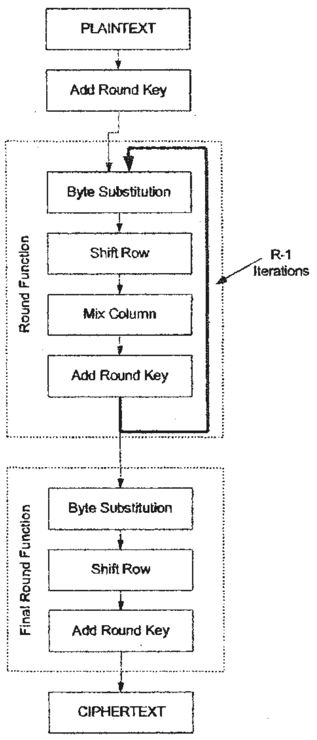

Figure 2.6: Rijndael Encryption

2.4 Bluetooth Encryption Algorithm - E0

In early 1998 a number of telecommunications companies (including Ericsson, Nokia, Intel, and Toshiba) formed a special interest group to develop a low-cost, short-range wireless technology. Named "Bluetooth", this technology would allow end users to eliminate the cumbersome wires connecting their devices [18]. For example, a Bluetooth enabled cellular phone could transmit wireless information to a Bluetooth enabled headset; a printer could wirelessly send data to a personal computer. In 1999 the Bluetooth SIG announced the Bluetooth 1.0 specification and a host of Bluetooth enabled devices began to enter the marketplace.

Within a Bluetooth packet the payload can be encrypted with a stream cipher called E0 that is re-synchronized for every payload [19]. A general framework for Bluetooth encryption can be seen in Figure 2.7. The E0 system handles the keystream generator initialization, keystream generation and payload encrytion/decryption. The

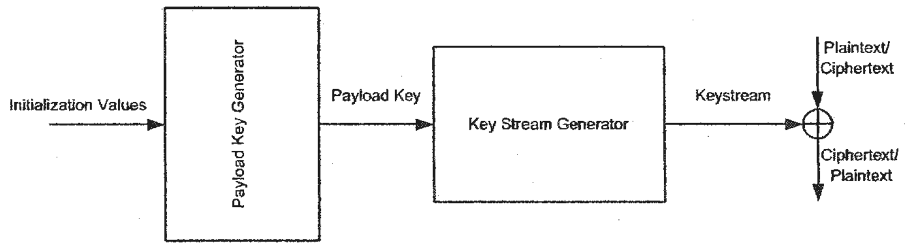

Figure 2.7: Bluetooth Stream Cipher

encryption key, K_C is derived from the current link key, a ciphering offset number and a random number. The payload key is then derived from this encryption key. Key management and generation were not considered in the scope of this research and further details can be found in the Bluetooth Specification [19]. The keystream generator is based on the summation generator proposed by Rueppel. In this system, four Linear Feedback Shift Register (LFSR) outputs are combined by a simple Finite State Machine (FSM) called the "summation combiner". The four LFSRs have lengths of 25, 31, 33, and 39-bits with the feedback polynomials as specified in Table 2.6. Figure 2.8 illustrates the E0 setup. As can be seen, the output of

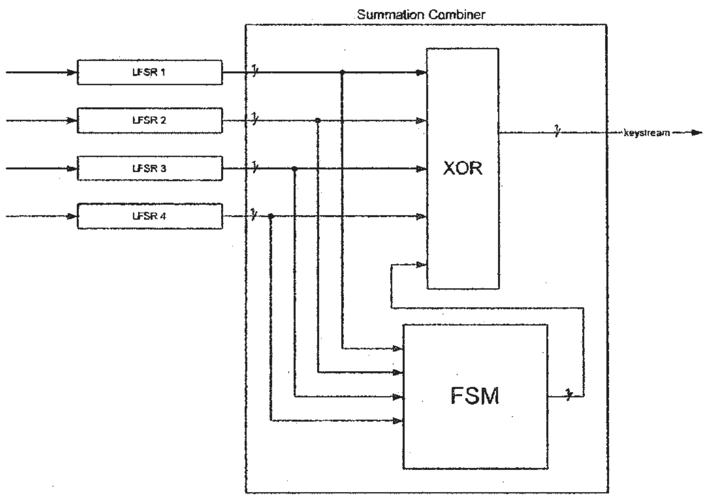

Figure 2.8: E0

the summation combiner is the key stream sequence that is bitwise added to the plaintext/ciphertext.

LFSR	Length	Polynomial
1	25	$t^{25}+t^{20}+t^{12}+t^{8}+1$
2	31	$t^{31}+t^{24}+t^{16}+t^{12}+1$
3	33	$t^{33}+t^{28}+t^{24}+t^{4}+1$
4	39	$t^{39}+t^{36}+t^{28}+t^{4}+1$

Table 2.6: LFSR Feedback Polynomials

2.5 KASUMI

The 3rd Generation Partnership Project (3GPP) was formed in 1998 to facilitate the collaboration of a number of telecommunications standards bodies. Their mandate was to facilitate the development of a global standard for the 3rd generation mobile system based on an evolution of the current GSM networks and the radio access technologies that they support [20].

The 3GPP security architecture includes both a confidentiality algorithm, *f8*, and an integrity algorithm, *f9*, both of which are based on a block cipher called KASUMI [21]. The 3GPP commissioned the Security Algorithms Experts Group (SAGE) to develop a security architecture for 3G networks. SAGE based the KASUMI cipher on the MISTY algorithm that was designed by Mitsubishi Electric Corp. in 1996 [22]. (In fact, 'KASUMI' is the Japanese word for 'MISTY'.)

KASUMI operates on 64-bit blocks of data and utilizes a 128-bit key. Like DES, it has a Feistel structure and is composed of 8 rounds. Figure 2.9 illustrates the top level structure of the KASUMI algorithm. The round function, f_i is composed of a number of subfunctions called FL, FO, and FI that are also associated with round subkeys KL, KO, and KI respectively [23]. The following sections describe these subfunctions in more detail.

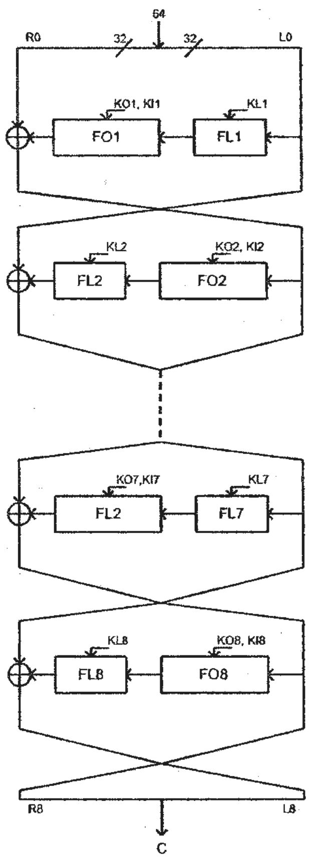

Figure 2.9: KASUMI Algorithm

2.5.1 Function FL

The function FL takes 32-bits of data, I, and a 32-bit subkey KL as its inputs. Both I and KL are split into two 16-bit halves (R, L, KL_R, KL_L) and processed by the following operations where $\wedge$ represents a bitwise AND operation:

$$R' = R \oplus ROL(L \wedge KL_L)$$
$$L' = L \oplus ROL(R' \wedge KL_R)$$

The ROL operation is a single bit rotation to the left. The 32-bit output of FL is the concatenation of R' and L' in the same order.

2.5.2 Function FI

The function FI takes a 16-bit data value, I, and a 16-bit subkey, KI, at its input. However, unlike the functions FL and FO, the data and subkey are split unequally into a 7-bit component and a 9-bit component. In the case of I, the left portion, L, is 9-bits and the right portion, R, is 7-bits whereas for the subkey the left portion, KI_1, is 7-bits and the right, KI_2, is 9-bits.

Two substitution boxes (or S-boxes), $S7$ and $S9$, are used in this function. $S7$ maps a 7-bit input to a 7-bit output and $S9$ maps a 9-bit input to a 9-bit output. Their mappings can be found in [23]. The following series of operations define FI's output:

$$\begin{array}{cc}
L_1 = R & R_1 = S9[L] \oplus ZE(R) \\
L_2 = R_1 \oplus KI_2 & R_2 = S7[L_1] \oplus TR(R_1) \oplus KI_1 \\
L_3 = R_2 & R_3 = S9[L_2] \oplus ZE(R_2) \\
L_4 = S7[L_3] \oplus TR[R_3] & R_4 = R_3
\end{array}$$
$$\text{output is } L_4|R_4$$

The operation $ZE(x)$ pads a 7-bit value to 9-bits by adding two zero bits at the most significant end. $TR(x)$ truncates a 9-bit value to 7-bits by discarding the two most significant bits.

2.5.3 Function FO

The function FO takes a 32-bit data input, I, along with two 48-bit subkeys, KO and KI, as its inputs. As in FL, I is split into two 16-bit halves, R and L. The 48-bit subkeys are each split into three 16-bit subkeys where $KO = KO_1|KO_2|KO_3$ and $KI = KI_1|KI_2|KI_3$.

The following operations complete the function's operation:

```
$L_0 = L$
$R_0 = R$
for $j = 1$ to 3
    $R_j = FI(L_{j-1} \oplus KO_j, KI_j) \oplus R_{j-1}$
    $L_j = R_{j-1}$
output $L_3|R_3$
```

2.5.4 Key Scheduling Algorithm

The key scheduling algorithm in KASUMI, although relatively simple, was not considered for implementation in hardware. Hence, we do not describe it here and further details of this algorithm can be found in [23].

2.6 RC4

RC4 (which supposedly stands for Ron's Code #4) is a proprietary algorithm created by Ron Rivest of RSA Data Security Inc. It is a keystream generator for use in a stream cipher that produces an arbitrarily long pseudo random sequence using a variable length key. In 1994, an anonymous source claimed to have reverse engineered the algorithm and posted their source code on the Internet [12]. The "alleged RC4" code produces an identical keystream to that of the original, but RSA Data Security Inc. claims that their algorithm is still a secret. Despite RSA's claims, this alleged version is widely assumed to be the RC4 standard.

The RC4 algorithm is relatively simple. It operates as a stream cipher where its output is bitwise added to the plaintext to produce ciphertext. The algorithm performs a series of operations utilizing an 8×8 S-box (i.e. a table composed of 256 1-byte values) to produce a byte of output. The following pseudocode sequence describes the operation to produce one byte of output:

```
$i = (i + 1) \bmod 256$
$j = (j + S_i) \bmod 256$
swap $S_i$ and $S_j$
$t = (S_i + S_j) \bmod 256$
$k = S_t$ where $k$ is the 8-bit output
```

where S_i represents the ith byte of the 256 byte S-Box. The S-box is initialized by first filling all entries linearly so that $S_0 = 0, \ldots, S_{255} = 255$. With K representing the key array the following operations are performed to complete the initialization:

```
for ($i = 0$ to $255$)
    $j = (j + S_i + K_{i \bmod 255}) \bmod 255$
    swap $S_i$ and $S_j$
```

Although the original RC4 description dealt with 8-bit words and 8×8 (256 input/output) S-boxes, the algorithm can be easily extended to an n-bit form. For example, a 16-bit RC4 version would have a 16×16 S-box and 16-bit outputs. Since the core of the algorithm is not affected by the size of n, a larger value of n should yield a faster implementation. However, the keystream outputs for different values of n will not be equivalent [12].

Chapter 3

Cryptographic Algorithm Implementation

Traditionally, communication system developers have had three standard implementation options available: ASICs, software running on a general purpose microprocessor, and FPGAs. One of the most common choices is to use a semi-custom or full-custom ASIC. Both of these devices use one of a variety of process technologies, such as CMOS and Gallium Arsenide, to produce analog or digital circuits on chip. Semi-custom ASICs utilize pre-developed blocks that implement complex functions along with custom developed circuits to achieve their final purpose. On the other hand, full-custom ASICs are designed without using any precompiled blocks and can be optimized in terms of both area and performance [24]. Because ASICs are designed to perform a specific set of computations, they can execute them extremely quickly and efficiently. However, after fabrication the circuit cannot be altered. Hence, they must be redesigned and remanufactured if any part of the algorithm is modified or if any part of the circuit is modified. The development cost for an ASIC is typically very high. Therefore, ASICs are only suitable in a high production volume application where high speed is required [8].

The second option is to use a general-purpose microprocessor and implement the algorithm in software. Microprocessors execute a set of instructions to implement an algorithm and by changing these instructions the implementation/algorithm can be altered without changing the associated hardware. However, with this added flexibility comes an overall lower level of performance and power efficiency when compared to an ASIC implementation. Secondly, since the set of available instructions is fixed when the microprocessor is fabricated, any other operations to be performed must be built from these instructions. This will result in a higher execution overhead for some operations when compared to a direct implementation.

The final option when implementing an algorithm is to use a reconfigurable device such as a FPGA. FPGAs consist of arrays of Configurable Logic Blocks (CLBs) that implement the functions of logical gates. The logical functions performed by the CLBs as well as the interconnections between them can be altered by sending signals to the chip. The FPGAs and their CLBs can be reprogrammed repeatedly and long after fabrication [25]. FPGAs now contain millions of gates per chip and can be used to implement very complex computations on a single device. Dehon in [26] showed that reconfigurable technologies had a raw computational density that was an order of magnitude higher than programmable (microprocessor) technologies when performing the same operations from cycle to cycle. Since configurations are written in a Hardware Design Language (HDL) and then mapped to the FPGA, no manufacturing is required and the development cost is much lower than that of an ASIC. However, FPGAs cannot achieve the computational speed or efficiency of an ASIC implementation and FPGA devices cost more to produce than ASICs in high volume applications.

Recent developments in the area of System On Chip (SoC) devices have given system designers much greater flexibility to implement ASICs. The development of soft instruction processors is one such advancement. Soft instruction processors allow the designer to quickly modify a pre-packaged processor core's instruction architecture. This core, when included in an ASIC design, can speed up development time and reduce costs dramatically [27].

Recently, the advent of reconfigurable processors has added yet another option to designers. Such processors incorporate the advantages of microprocessor and FPGA technologies and have shown significant performance increases in the areas of image processing, compression, computational chemistry [25], object tracking, fuzzy controllers [28], music [29] and cryptography. The following sections outline some recent developments in the areas of reconfigurable computing and its applications in the field of cryptography.

3.1 Reconfigurable Processor Architectures

An FPGA can act as a stand alone device in a system just as in ASIC implementations. However, devices that couple a general-purpose microprocessor with a reconfigurable logic device have started to emerge in the marketplace. These devices have the key feature of being able to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution. More importantly it is possible, in some cases, for the configuration to change at run time.

Wittig in [30] describes three general classes of reconfigurable systems:

1. FPGA loosely coupled to a fixed host computer
2. FPGA loosely coupled to a fixed, integrated CPU
3. FPGA closely coupled to a fixed, integrated CPU

While any reconfigurable system will fall under one of these categories, it is sometimes difficult to differentiate between classes 1 and 2. Instead, a classification system based on both the coupling level and type of interaction between the general-purpose processor and reconfigurable logic can be used as outlined below [31].

Stand Alone Processor

In this case, the FPGA acts as a stand-alone processor as described in the previous section. Any communication between the FPGA and the CPU must be done through an I/O interface. Since I/O communication is relatively slow, it is only useful when communication between FPGA and CPU is infrequent.

Attached Processor

When the FPGA acts as an additional processor in a multi-processor system it is classified as an "attached processor". Typically, in this sort of system, the FPGA communicates with the other processors over a common bus. Hence, this form of system is more closely coupled than the stand-alone case since bus transactions are usually much more efficient than I/O operations.

Coprocessor

In some systems, the FPGA may aid the CPU with certain computations and is acting as a coprocessor. Depending on the system, the FPGA can do these computations in parallel with the CPU and this can dramatically improve system performance. As well, in most cases these coprocessors have direct access to the CPU's main memory, further improving system performance.

Reconfigurable Functional Unit

As the level of integration for ASICs has increased the idea of placing both the CPU and the reconfigurable logic, or Reconfigurable Functional Unit (RFU), on the same chip has become a reality. In this arrangement, the RFU can be directly added to the processors execution pipeline in parallel with the existing CPU units. This allows new instructions to be created dynamically in the RFU that can be added to the already existing instruction set. As the amount of logic that can be integrated with the CPU increases, reconfigurable system performance will improve substantially.

It is possible to obtain significant speedups over software implementations by using reconfigurable hardware. However, this is only true when the communication overhead for implementing an algorithm in hardware is small compared to the amount of computation done in the reconfigurable hardware. Wittig in [30] quantified this with the following equations:

$$T_H + T_{OV} < T_S$$
$$\frac{T_H}{T_S} + \frac{T_{OV}}{T_S} < 1$$

where

$$T_{OV} = \ time\ to\ comummunicate\ data\ and\ control\ overhead$$
$$T_H = \ time\ to\ execute\ function\ in\ hardware$$
$$T_S = \ time\ to\ execute\ function\ in\ software$$

This equation is only applicable when it is feasible to implement an algorithm in hardware (i.e., the overall execution time is smaller in hardware than in software). In the fractional form of the above equation, the quantity $\frac{T_H}{T_S}$ represents the actual hardware computational speedup and the fraction $\frac{T_{OV}}{T_S}$ represents the granularity of the application implemented in hardware. Hence, an implementation with a small $\frac{T_{OV}}{T_S}$ ratio indicates that it has a larger grain size since less communication occurred between CPU and reconfigurable hardware. Systems with a small $\frac{T_{OV}}{T_S}$ ratio need a smaller hardware speedup to have the same overall speedup as systems with a larger $\frac{T_{OV}}{T_S}$ ratio. Therefore, the communications overhead plays a vital role in high performance applications utilizing reconfigurable devices.

Until relatively recently, the available ASIC manufacturing processes did not support a level of integration necessary for the development of complex RFU systems. However, reconfigurable systems that closely couple a microprocessor with a relatively large reconfigurable core have begun to emerge in the market. Before considering the implementation of algorithms on one such architecture, it is important to understand some of the architectural features that are common to all reconfigurable microprocessors. The addition of closely coupled reconfigurable logic not only adds new design issues but also complicates decisions that can be made quite easily in a software or ASIC and stand-alone FPGA implementations. The following sections detail some of the architectural options available when developing a system containing such a coupled reconfigurable device.

3.1.1 Microprocessor Architecture

There are a number of general requirements for any fixed CPU that is attached to reconfigurable logic, the most basic being speed and interfacing flexibility. The processor must be fast at executing instructions from its own instruction set as well as providing support for the instructions custom built in reconfigurable logic [32]. In particular, the CPU to reconfigurable logic interface must not be slower than the computational delay of the instructions implemented in reconfigurable hardware (i.e. a high $\frac{T_{OV}}{T_S}$ ratio). Jeschke in [33] has found that the achievable speedup from a reconfigurable system can be severely limited by the CPU-logic interface and suggests that the current state of the art microprocessor should be used in a design. In most cases, a fast Reduced Instruction Set Computer (RISC) CPU is coupled with reconfigurable logic since the RISC CPU's limited instruction set allows designers to implement complex functions in reconfigurable logic on a per application basis.

3.1.2 Logic Block Granularity

Reconfigurable hardware is typically based on a set of computation structures that are repeated to form an array. These structures, commonly called logic blocks or cells, vary in complexity from a very small and simple block that can calculate a function of only two inputs, to a structure that is essentially a 32-bit Arithmetic Logic

Unit (ALU). Some of these blocks are configurable themselves in that the performed operation is chosen from a configuration set. Other blocks perform fixed operations and their configurability lies in their interconnection. The size and complexity of the basic computing block is referred to as the block's granularity [34].

Figure 3.1 [35] shows an example of a fine grained logic block that is found in the Xilinx 3000 series of FPGAs [35]. This type of logic block is useful for fine-

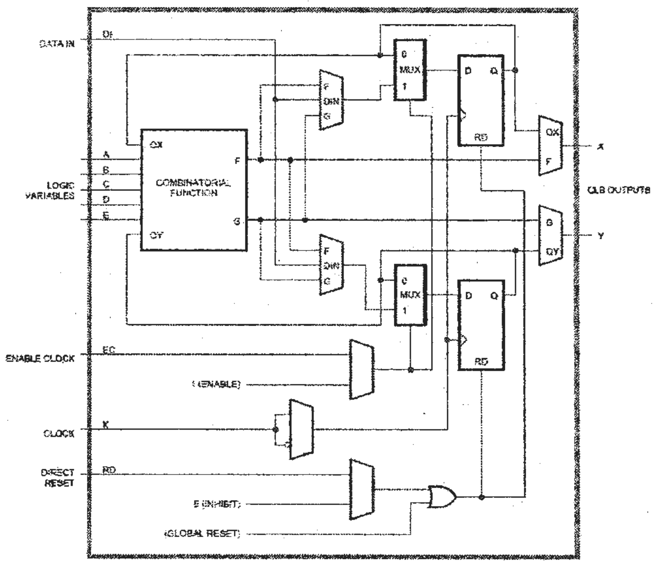

Figure 3.1: Xilinx XC3000 Series Configurable Logic Block

grained bit-level manipulation of data. These kinds of operations are frequently found in encryption and image processing applications. Also, because these cells are fine grained, computation structures of arbitrary bit widths can be created. Figure 3.2 shows an example of a very coarse grained reconfigurable architecture, the Chameleon CS2112 that is discussed further in Chapter 4. The Chameleon CS2112 is not just a coarse grained solution. It is also termed a "heterogeneous" architecture since there are not only data path units, but also multipliers, control logic and data memory residing in the reconfigurable portion of the chip. A "nonheterogenous" architecture, on the other hand, would be composed of totally identical reconfigurable logic cells. Typically, very coarse grained architectures are intended for the implementation of

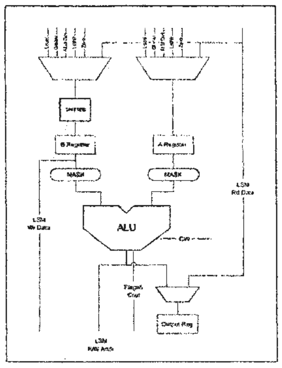

Figure 3.2: CS2112 Example

word-width data path circuits and will perform word sized computations much more quickly than a set of smaller CLBs connected to perform the same function. However, they are inefficient at performing operations on bit level data when compared to fine grained architectures.

3.1.3 Data Interconnection

Another important component of a reconfigurable architecture are the routing structures used within the reconfigurable portion of the design. One group has argued that the interconnect should constitute a much higher proportion of the area in order to allow for successful routing under high logic utilization conditions [36]. However, routing resources occupy a much larger part of the area of an IC than the logic resources. As a result, the most area efficient designs will be those that optimize their use of routing resources rather than the logic resources.

The two primary routing structures used in reconfigurable designs to provide both local and global routing resources are illustrated in Figure 3.3. The first is segmented routing in which short wires accommodate local communications traffic. These short segments can be connected together using switchboxes to emulate longer wires. The second form of routing is hierarchical routing. In this case, routing within a group of logic blocks is at the local level and at the boundaries of these groups longer wires

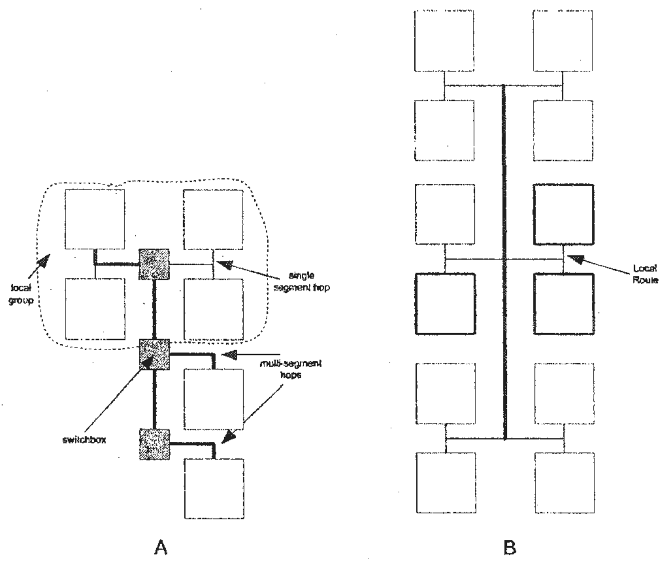

Figure 3.3: Segmented (A) and Hierarchial Routing (B) Examples

are used to connect the groups together. Provided a good mapping has been made in hardware, the most common communication should be local in a hierarchical scheme. Both of the above schemes are referred to as “island-style” routing architectures. A few alternatives use a one-dimensional routing scheme. One example is a bus-based scheme in which only vertical or horizontal busses connect the configurable elements.

In other systems multiple FPGAs are linked to form a reconfigurable device. These require not only an efficient internal routing scheme, but also an efficient external interconnection architecture. These systems are typically used when an algorithm is too large to fit on a single reconfigurable device.

3.1.4 Reconfiguration Models

Traditional FPGA structures have been single-context, allowing only one full-chip configuration to be loaded at a time. However, the designers using reconfigurable systems have found this style of configuration to be too limiting and/or slow to efficiently implement run-time reconfiguration. A number of methods that have been developed to limit the overhead of changing configurations at run-time are discussed below.

In a single context device, configurations are loaded using a serial stream of configuration information. Because only sequential access is supported, any change to a configuration on this type of device requires a complete reprogramming of the entire chip. This type of access does simplify the reconfiguration hardware but it does create a high reconfiguration overhead when only a small part of the configuration needs to be updated. In order to implement run-time reconfiguration using a single context device the configurations must be grouped into contexts, and each full context swapped into and out of the device as needed. Since swapping the contexts involves reconfiguring the entire device, great care must be taken in designing the configurations [37].

A multi-context device includes multiple memory bits for each programming bit location. These memory bits can be thought of as multiple planes of configuration information. One plane of configuration information can be active at a given moment, but the device can quickly switch between different planes of pre-programmed configurations. This system does allow for the loading of a configuration in the background while the active plane is running. In this case, the grouping of configurations into contexts is less critical because of both the background loading capability and the ability to switch contexts rapidly. However, it is still important to ensure that the configurations used in close proximity in time are loaded into the multi-context device at the same time [26].

In some cases, configurations do not occupy the all of the reconfigurable resources, or only a part of an active configuration requires modification. In these cases partial reconfiguration would be useful. In such reconfigurable devices, the underlying programming layer operates like a RAM device where addresses are used to specify the target location of configuration data. In some cases, the unchanged portion of the device can remain active while the new data is being loaded, further hiding configuration latency. A further modification of the above strategy is for the partial reconfiguration to occur in stages [38]. This type of reconfigurable hardware is called pipeline reconfigurable or a stripped device. This type of structure also allows for the overlap of configuration and execution time as one pipeline stage is configured while the others are executing.

While multi-context and partially reconfigurable devices reduce the time required to switch configurations, the fact remains that reconfiguration will occur during program execution. A number of different tactics for reducing configuration overhead have been developed. First, the loading of configurations can be timed such that the configuration overlaps as much as possible with the execution of instructions by the host processor. Second, compression techniques can be introduced to decrease the amount of configuration data that must be transferred to the system. Third, the number of reconfigurations can be reduced through hardware optimizations that keep configurations that will be reused from being unnecessarily replaced by incoming configurations. Fourth, the actual process of transferring the data from the host processor to the reconfigurable hardware can be modified to include a configuration cache, which would provide a faster reconfiguration [39]. Finally, Sakr in [37] proposed employing the use of optical channels to allow fast parallel loading of the reconfiguration control word as well as the migration of the configuration cache off-chip.

3.1.5 Programming Models

Significant gains in performance can be gained through using reconfigurable hardware. However, application programmers will tend to ignore this potential unless they are able to easily incorporate its use into their systems. This requires a software design environment that aids in the creation of configurations for the reconfigurable hardware. This can range from a set of libraries to assist in the manual creation of circuits to a complectly automated circuit design system. While the manual creation of circuits requires a great deal of background knowledge of the system being used, it allows for the creation of designs that are usually more efficient than automatic compilation systems. However, automatic compilation systems provide a simple means for creating an application and, as such, make the use of reconfigurable hardware more accessible.

Another complication in the development of an application for a reconfigurable system is that the program must first be partitioned into sections to be executed on the reconfigurable hardware and in software by the microprocessor. Wittig in [8] described a software environment that any "user-friendly" reconfigurable system should include. This tool would automatically provide the user with the most efficient system configuration; the user would only have to program the system in a high level language. Figure 3.4 illustrates the operation of this system. In the first stage, the

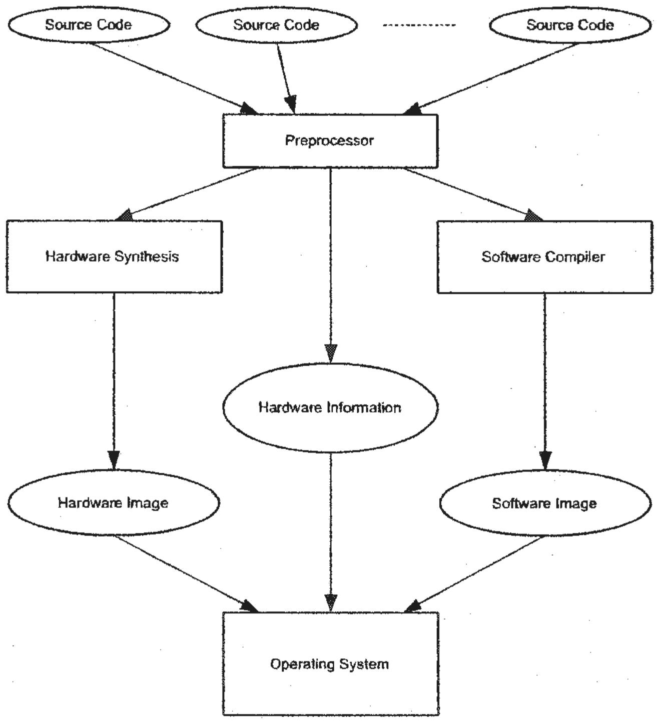

Figure 3.4: Ideal Software Development Environment for Coupled Reconfigurable Devices

preprocessor is used to identify sections of code that are candidates for execution in hardware. Essentially, the preprocessor attempts to perform the hardware/software partitioning. It should be noted that the preprocessor does not necessarily produce an optimal set of hardware functions. Next, the hardware is synthesized and mapped to the reconfigurable logic and the software is compiled. These hardware and software images are then given to the operating system (OS). In this scheme, the OS is responsible for deciding what portions of the software code are to be executed in hardware and schedules operations so as to give the best overall performance. Hence, using this type of software environment, users could run their standard high level language applications on a reconfigurable system just like on a standard, fixed microprocessor, while benefitting from an optimal use of the reconfigurable hardware resources [8].

Existing reconfigurable system software environments do not typically provide the level of automation described above. In systems where the function identification and extraction process is automated, the user is still usually required to select the most desired candidates from a complete list of synthesized functions. In the majority of commercially available systems, the user is required to complete the entire processes of selecting candidate functions, writing the configurations and, in some cases, mapping them to reconfigurable hardware.

3.2 Reconfigurable Computing and Cryptographic Hardware

As stated before, the explosive growth in the Internet and mobile communication has led to increased research and development in the area of cryptography. This research can be categorized into three broad areas based on cryptographic capabilities. In the most specific category are the designs which implement only a single algorithm. Typically, such implementations are developed on either ASICs, FPGAs or in software and are used to evaluate and optimize the performance of a particular algorithm in the chosen medium. At the next level of capability is the cryptographic accelerator. These hardware devices implement multiple cryptographic algorithms and also accelerate processing at the communications protocol level. Finally, in the most general category, are the hardware devices which aim to accelerate cryptographic primitives but not specific algorithms.

When a new cryptographic algorithm is developed and during its useful lifetime, it is scrutinized not only from a security perspective but also from an implementation point of view. The algorithm may be mapped to software, hardware or both, depending on its target application. For example, DES was designed as a general cryptographic standard and, as such, numerous hardware and software implementations have been developed since its release in 1977. Although developed for hardware implementation, DES has been successfully implemented in software with speeds of greater than 500 Kbps [40][41][42]. However, a much larger effort has been devoted to developing high speed hardware implementations of DES. Wilcox et al. in [43] describe an ASIC design that can achieve up to 10 Gbps throughput. As well, Leitold in [44] describes a single chip Triple-DES - a form of the algorithm in which data is encrypted three times - solution that can operate at speeds of up to 155 Mbps. Also, a number of reconfigurable system implementations have been developed over the last 26 years. Trimberger in [45] details the development of a DES FPGA core which has a throughput of up to 12 Gbps. Table 3.1 summarizes some of the results obtained with the other ciphers discussed in Chapter 2.

The most recent development in the communications security marketplace has been the development of devices that accelerate a number of algorithms on a single chip. These devices are usually found in systems as an encryption coprocessor which is handed data to be encrypted/decrypted, an inline processor which views all packets and performs security duties when necessary, or as a portion of the network system

Cipher	Implementation	Throughput
AES with 128-bit blocks	Xilinx Virtex-E FPGA [46]	7000 Mbps
	Xilinx Virtex FPGA [47]	353 Mbps
	ASIC [48]	1820 Mbps
	Altera FPD [49]	900 Mbps
	APEX FPD [49]	570 Mbps
	ASIC [50]	2360 Mbps
	Software (Pentium IV 2 GHz) [51]	approx. 700 Mbps
KASUMI	ASIC [52]	1100 Mbps
E0	ASIC [53]	320 Mbps
	Software [53]	33 Mbps
RC4	Software (DEC 3000/400) [54]	15.4 Mbps

Table 3.1: Cipher Performance Figures

processor itself [55]. Companies such as Broadcom [56], Cavium Networks [57] and Corrent [58] all produce boards aimed at accelerating IPsec and SSL/TLS protocol transactions. These protocols, which are widely used in packet data communications, contain a variety of public and private key algorithms, such as DES and AES, that are used to provide authentication, authenticity and privacy. Although these devices are typically implemented as ASICs, Andoni in [59] presents an FPGA based IPsec accelerator with impressive throughput capability.

Since many of the primitive operations, such as bit permutations, XORs and table lookups, are repeatedly used in most private key cryptographic algorithms, it is very appealing to try to develop an architecture that is optimized for this subset of operations. As well, such architectures usually couple a microprocessor with a reconfigurable core to allow splitting the algorithm across hardware and software in an optimum manner. Since some operations can be more efficiently implemented in software and vice versa this is an important attribute. Also, these architectures allow the developer to add new algorithms as specifications change and give the designer much more flexibility than an ASIC solution. One such example is the CryptoBooster coprocessor developed by Mosanya et al. in [60]. It is a modular architecture that allows a user to load in various cryptographic modules needed to accelerate their application. Another example is the PipeRench architecture developed by Taylor [61]. This architecture is a pipelined reconfigurable fabric that is optimized for many of the operations commonly used in private key cryptography. PipeRench also utilizes a virtual hardware scheme so that large hardware configurations can be supported on limited physical hardware. A third example is CYPRIS, a reconfigurable microprocessor developed by Lockheed Martin Corp [62]. The CYPRIS architecture contains a high speed RISC processor and a reconfigurable logic block on the same die. Its primary goal was to provide security in hand held radio and other radio communication devices. In all cases, a number of ciphers were implemented and significant performance gains were achieved over purely software implementations. However, the algorithms from Chapter 2 were not discussed so performance comparisons could not be made.

Chapter 4

The Chameleon CS2112 Reconfigurable Communications Processor

In the late 1990s, Chameleon Systems Inc. began work on a new reconfigurable processor architecture targeted toward the communications marketplace. Their chip, the Chameleon CS2112 RCP released in 2001, was considered the world's first reconfigurable communications processor. The CS2112's reconfigurable logic was optimized for signal and protocol processing applications and was accompanied by a proprietary set of tools to aid system designers in developing their applications. The following sections detail the architectural features of the CS2112 and give a brief description of the process involved in developing a CS2112 application.

4.1 Chameleon CS2112 Architecture

The CS2112 processor includes an embedded 32-bit RISC-based CPU capable of operating at 100 MHz and a proprietary reconfigurable logic fabric in the device architecture. The workhorse of the CS2112 is its reconfigurable logic. Because the reconfigurable logic is full-custom 32-bit data path oriented, the CS2112 requires far fewer configuration bits than conventional single-bit oriented FPGAs [63]. Each slice on the CS2112 can store two complete sets of configurations (i.e. multi-context device), an active configuration and a background configuration. This type of dual plane system can be quickly swapped in just one clock cycle, enabling the fabric to be easily reconfigured on demand to perform whatever function is required at the current point of execution in the application.

Figure 4.1 depicts the high-level view of the CS2112 architecture. The CPU is an Argonaut RISC Core (ARC) that is a full 32-bit, 4-stage pipelined processor. As

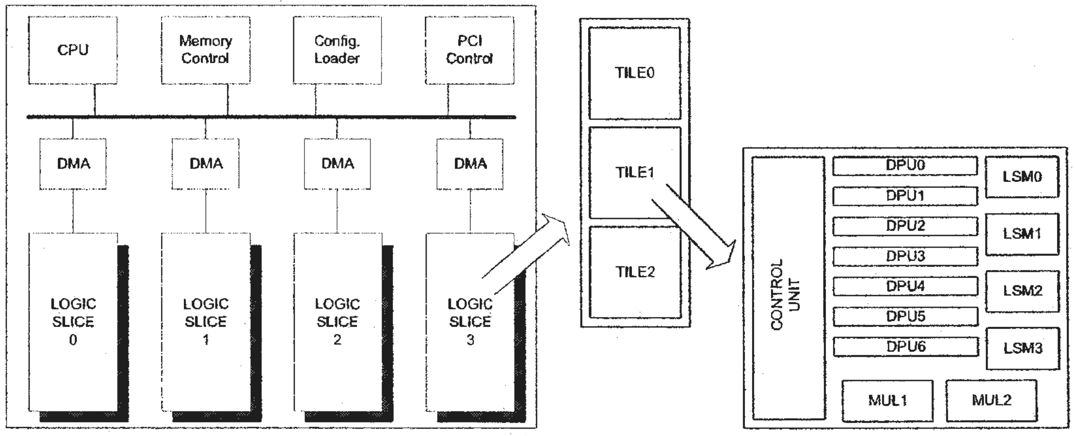

Figure 4.1: CS2112 High Level Architecture

can be seen in Figure 4.1, the reconfigurable fabric is divided into 4 slices and each logic slice is further subdivided into 3 tiles. Each tile is identical and consists of seven 32-bit Data Path Units (DPUs), two 16×24 single-cycle Multiplier Units (MULs), four Local Store Memorys (LSMs) and a Control Logic Unit (CLU). The CS2112 is considered a coarse-grained architecture since its smallest functional unit is a fairly complex ALU.

4.1.1 Datapath Units

The DPU, a detailed view of which is shown in Figure 4.2, is a data processing module that directly supports a variety of C and Verilog operations. Each DPU operation can utilize 2 input operands and produces a single 32-bit result. It supports 32-bit operations, some 16-bit operations and some 16-bit Single Instruction Multiple

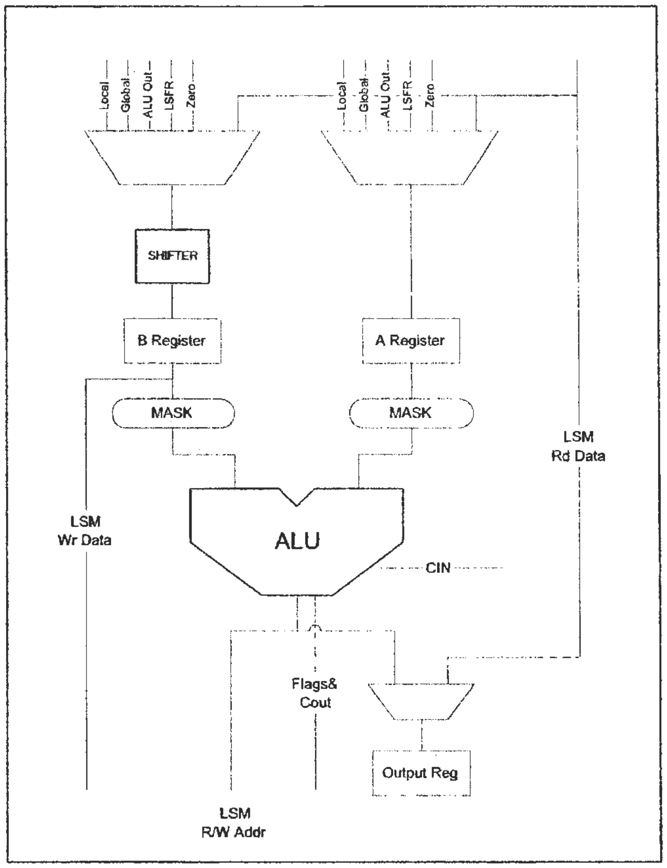

Figure 4.2: CS2112 DPU

Data (SIMD) operations. The DPU also includes a 32-bit barrel shifter that is capable of performing bit shifts, word swaps, byte swaps and word duplication. As well, the DPU contains two 32-bit AND/OR mask operators/registers (one for each input path).

4.1.2 Multipliers

The two $16x24$ single-cycle MULs operate in two modes: 16×16-bit mode and 24×16-bit mode. In the 16×16-bit mode, the MULs implement a signed multiply with a 32-bit result. In 16×24 mode, the 40-bit signed product is truncated to 32-bits by rounding the 8 least significant bits.

4.1.3 Local Store Memories

There are also four 32-bit wide by 128 words deep LSMs per tile as shown in Figure 4.1. LSMs can be chained to build wider and/or deeper memories if required. The LSMs can be accessed by certain DPUs in the same tile as well as by the DMA subsystem. Each LSM has four ports that allow for simultaneous access by the DMA subsystem and DPUs. The DPU access ports can be configured as either 32-bit, 16-bit, or 8-bit ports.

4.1.4 Control Logic Units

The fabric's control structure allows the simultaneous control of all fabric resources and the CLU is constructed to allow state machines to operate in parallel. Within each CLU there is a muxing plane, a Programmable Logic Array (PLA), a number of state register blocks and Control State Memories (CSMs). The muxing plane is used to select control signals for the PLA which has 16 inputs, 32 outputs, and 32 product terms. Each of the outputs of this PLA drives one of the state register blocks. Each of these state register blocks is 4 bits wide and can be used as either state bits of a state machine or as state machine control logic. The CSMs contain configuration information for each of the DPUs or MULs in a tile. Up to eight configurations for a DPU and four for a MUL can be stored in a Control State Memory (CSM). The state machine control logic bits of the state register blocks are used to select the current active configuration from the CSM [63].

4.1.5 Data Path and Control Routing

The Chameleon CS2112 uses a hierarchical routing scheme inside the reconfigurable fabric to route data between DPUs and MULs. Within a slice, nearby DPUs and MULs are connected with a full crossbar interconnection. Vertical intra-slice routes and horizontal inter-slice routes allow DPUs and MULs outside of the local interconnect to communicate. Figure 4.3 [63] illustrates the interconnection scheme. Any data using a path outside of the local interconnects will encounter a single clock delay since inputs to a DPU must be registered if the path utilizes a global routing interconnect.

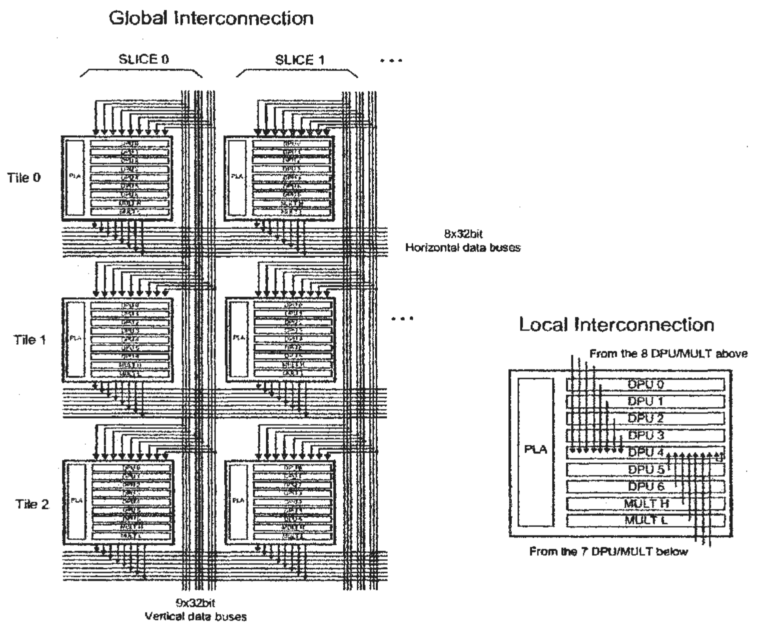

Figure 4.3: CS2112 Fabric Routes

4.2 CS2112 Design Methodology

The development of a hardware fabric function, or kernel, usually proceeds in four phases: the C Code Model Development Phase, the Design Phase, the Synthesis and Mapping Phase, and the Verification and Integration Phase. Chameleon Systems provided a number of tools called the C$\sim$Side^{TM} Tools, to facilitate the development of fabric functions. Figure 4.4 illustrates these phases and their relation to the software tools. Within these tools Chameleon provides a set of behavioral models of tile components (DPUs, MULs, LSMs) to facilitate the development of a fabric

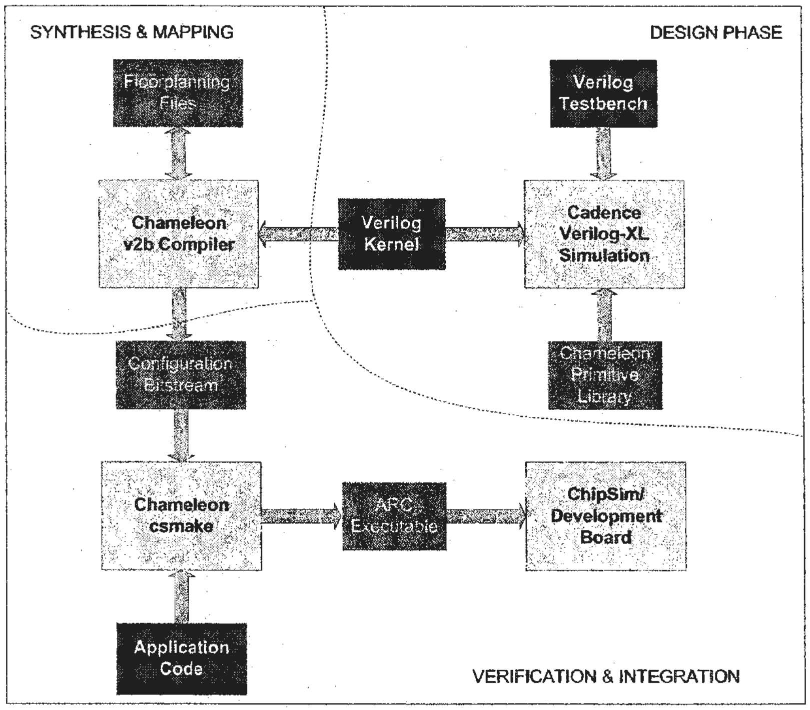

Figure 4.4: Design Flow

function. A detailed knowledge of the reconfigurable fabric is then required to decide on a hardware/software boundary as well as to write an efficient configuration. The following sections provide a more detailed description of the above phases along with some simple design examples.

4.2.1 Software Model

The Chameleon Systems design group recommends that the development of a fabric function, or kernel, begin with the development of a C code model of the system. Once this model is verified, it can be used as a reference model against which the Verilog implementation can be tested.

There are a number of C design requirements that must be met to form a "legal" fabric function. Firstly, the whole function body must be converted to a fabric function. A block of code within a C function cannot be converted. Secondly, a function to be converted must be a leaf function - it cannot call any functions itself. This means recursive calls are also illegal. Thirdly, the function can only communicate with the rest of the program through its arguments and if a function returns a value it must do so through a function parameter. Fourthly, all array function arguments must be aligned to a 128-bit memory boundary. Lastly, floating-point values are not supported within a kernel function and cannot be used as arguments.

4.2.2 Design Phase

After a software model has been completed a behavioral Verilog model is developed. This model is broken down into two main components (as with most digital designs): data path and control. Data path logic is described by instantiating data path elements from the Chameleon primitive library (some Verilog operators can be converted directly) and control logic is described using RTL state machines.

Chameleon Systems provide a set of Verilog hardware primitives to assist in development. They are dpDPU, dpLSM, dpMUL, dpSRB and dpIOB (for representing the chip's I/O pins). These allow the user to have total control over the resources of the RCP. For example, the dpDPU module has 8 40-bit instruction inputs that are used to configure the DPU's operation. With their latest tools release, Chameleon Systems also provides higher level Verilog objects - CS2112_DPU, CS2112_LSM and CS2112_MUL - that encapsulate the previously described primitives to simplify design entry. Further information about the specific use of these primitives can be found in [64].

As stated above, the control logic is implemented in the CLUs of tiles. State machines are essentially implemented in the PLAs with control registers sequencing CSM instructions. The PLA determines the next state based on DPU flags, the output of a DPU/MUL, state registers, or inputs from external data on Programmable I/O (PIO) lines. Two types of state machines can be implemented in the fabric:

FSMs and sequencers. FSMs, in which only a single state is active at a time, can be implemented on the fabric and are typically written as Moore machines; states can be encoded in a variety of ways. Multiple FSMs, generating outputs and fabric control signals, can be active at one time. Communication between these state machines can be accomplished by broadcasting FSM outputs across the fabric using the global routing lines discussed previously. A sequencer is a simpler form of an FSM. Since many of the designs on the CS2112 are highly pipelined, many states are used in the control unit to wait for data to fill or leave a pipeline and the state transitions in these sections unconditionally move to the next state. The use of a sequencer to implement this portion of the control logic reduces the hardware required to implement the design in the CLU.

As an example, consider the implementation of function which rotates the 32-bit input data by 11 bits to the left. No rotate function is directly available in a DPU; however, logical shifts to the right or left can be performed by the DPU hardware. A single DPU with two instructions could be used to perform the operation as shown in Figure 4.5, assuming the input data is held at the input for 2 clock cycles. The first instruction logically shifts the data on the "B" side input of the DPU by 11 bits to the left and passes the result in the DPU output register. The second instruction takes the same input data and logically shifts it to the right by 21 bits before the ALU ORs it with the result of the previous instruction. The output of the OR operation, which is stored in the DPU output register, is the 11-bit rotated result. A simple FSM can be used to toggle from instruction 0 to instruction 1 after a clock cycle since each of the above instructions is performed in a single clock cycle. This operation can also be done in two DPUs, each of which implements one of the instructions above. A pipeline register is added to the second DPU's instruction to buffer the input data for one clock cycle. In this case, no FSM is necessary to control the operation since each DPU has only one configuration. However, in both cases the rotation operation

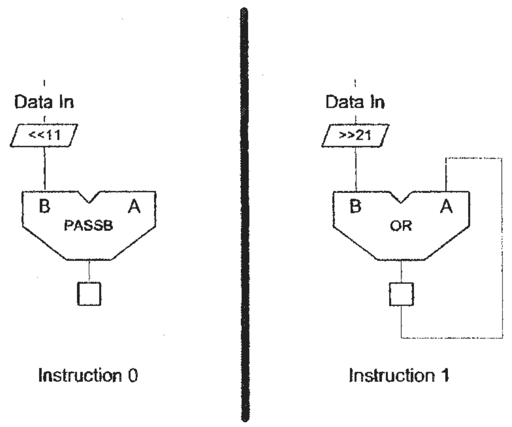

Figure 4.5: Single DPU Rotate Left by 11 Bits

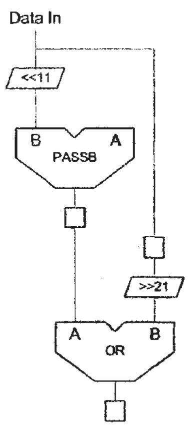

Figure 4.6: Pipelined Rotate Left by 11 Bits

takes two clock cycles to complete. Verilog modules, which implement both of the examples above using the provided hardware primitives, can be found in Appendix A.

4.2.3 Synthesis and Mapping Phase

In the synthesis and mapping phase, the Verilog kernel is compiled by the Chameleon v2b tool to create the configuration bitstream. This compilation process involves synthesis, mapping, and placement. In this process the elements of the Verilog description are mapped to specific fabric resources. The v2b tool's placement algorithm usually does not produce a routable kernel for complex designs. Chameleon provides a "worksheet" on which the designer can manually decide on the placement of their data path elements. This worksheet can then be used in combination with a graphical placement tool to achieve a routable design [64]. Figure 4.7 is an example screenshot

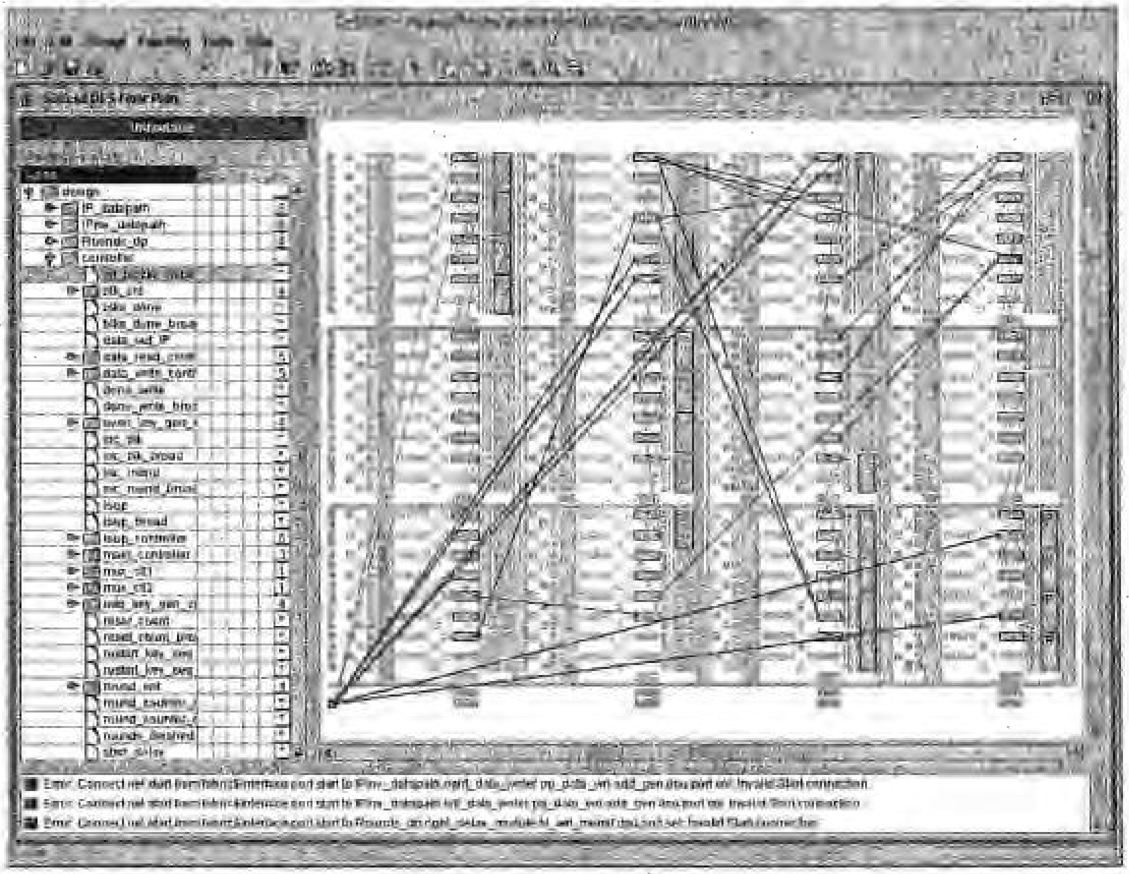

Figure 4.7: C~Side^{TM} Graphical Floorplanner

of the placement tool's graphical floorplanner. Fabric elements such as DPUs and MULs can be moved to different slices or tiles on this screen while the tool decides on the new data routes required to accommodate the changes.

4.2.4 Integration and Verification Phase

In the integration and verification phase, the kernel bitstream is linked with application code, written in C, for verification of the bitstream and integration into the overall application. Chameleon provides a custom C preprocessor and a number of software libraries, called the eBIOS libraries, to facilitate application development. The developer can use the provided preprocessor to simplify the fabric interface. In the application code two "pragma" lines are inserted to substitute a software function with an equivalent kernel function - one to define the function and another to make the call. For example, the following code:

```
#pragma CMLN_FUNC_DEF spne(int in dp.sbox1.lsm[N], int in dp.sbox2.lsm[N],
                           int in dp.sbox3.lsm[N], int in dp.sbox4.lsm[N],
                           int in dp.xor_dpu.dpu.o, int in dp.key_lsm.lsm[8],
                           int out *dp.xor_dpu.dpu.o)
```

defines a kernel function called 'spne' with 5 32-bit array inputs, a single 32-bit input and a single 32-bit output. To make a call to this function, the following line would be placed in the application code in place of the software function call:

```
#pragma CMLN_FUNC_CALL spne()
```

The preprocessor then expands this into the eBIOS calls necessary to configure the fabric resources, start the hardware function and transfer any data to and from the fabric [64]. The application developer can also use the eBIOS fabric interface library directly to utilize their fabric function. The preprocessor output code for the above example is given in Appendix B. Once the application code has been compiled, Chameleon provides both a software chip simulator, as well as a test board which houses a CS2112, to verify the final application.

Chapter 5

Data Encryption Standard Implementation

In this chapter, the design and implementation of three DES kernels are discussed. The development of a DES kernel began by dividing the kernel into three distinct blocks as shown in Figure 5.1, with multiple iterations of the middle block completing the "rounds" of the cipher. The following sections discuss the development of:

- A purely iterative design
- A design involving multiple kernels and pipelined data
- A consolidated design that supports pipelined data

The implementations described below vary functionally in terms of the path that data takes through the kernel as well as the quantity of data that passes through the kernel at one time.

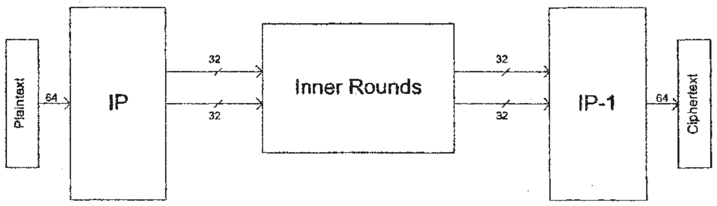

Figure 5.1: Iterative DES Kernel

5.1 Iterative Kernel

The first attempt at implementing DES on the CS2112 involved the development of a fabric function that would encrypt a single 64-bit plaintext. As such, it would mimic the capability of a purely software implementation. After a single plaintext had been passed into the fabric from the software application it would pass through the IP stage hardware. After passing through the IP stage, the data would then iterate in the inner round hardware 16 times before passing through the IP^{-1} hardware. The encrypted data would then be passed from the fabric back to the software application.

In order to simplify the design, the key scheduling algorithm was not implemented in the CS2112 fabric. Instead, the complete set of 16 round subkeys was to be constructed by the software application and passed into the hardware function where they were stored in LSMs. Hence, if multiple plaintexts were to be encrypted with the same key, the same set of round subkeys would be used in each encryption and the key scheduling algorithm need not be executed if the subkeys had been held in microprocessor memory.

5.1.1 Architecture Description

Since DES's inception in the 1970s, a number of hardware and software implementations of the algorithm have been developed and published. Hardware implementations of DES typically involve bit-level operations, particularly in the IP and IP$^-$1 portions of the algorithm. The DPUs, however, provide 32-bit functionality and performing bit-level operations in the DPUs is quite costly in terms of fabric utilization. Therefore, a high speed 32-bit C++ software implementation, developed by Richard Outerbridge, was used as a basis for the architecture of the kernels described below [12]. In this software implementation, the 64-bit permutations in the IP and IP^{-1} stages and the 48-bit operations in the F-function are all performed using 32-bit operations and operands.

As stated above, the iterative kernel was to operate on a single 64-bit plaintext. The C++ implementation chosen as a basis for development contained a simple single plaintext encryption function that was replaced by the iterative kernel. The iterative DES kernel can be looked at as being composed of 5 main blocks as shown in Figure 5.2. The following sections give an overview of the architecture of these blocks and their associated control.

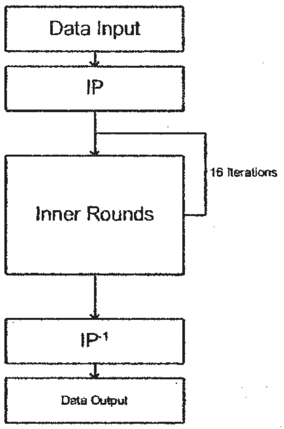

Figure 5.2: Iterative DES Kernel Blocks

Data Input and Output

Data to be processed by CS2112 kernels can either be passed into the fabric and stored in DPU registers, loaded into LSMs or placed on PIO inputs. In the iterative kernel case, only a single 64-bit plaintext, which could be split into two 32-bit halves, was to be processed by the fabric at one time. Hence, the initial data values were simply loaded into the input registers of the first two DPUs of the IP stage. The output of the kernel is again two 32-bit halves that are retrieved from the output registers of the last two DPUs of the IP^{-1} stage.

IP and IP^{-1}

At the bit level as described in the standard, the IP and IP^{-1} stages are simple wire crossings. Using 32-bit operations, a permutation across 64-bits is not so trivial. However, utilizing 32-bit operations these permutations can be accomplished using a number of bit shifts, simple logical operations, and bit masks. The pseudocode sequences that describe the operations necessary to perform these two permutations are shown in Algorithms 5.1.1and 5.1.2.

Algorithm 5.1.1 IP

```
work = ((left >> 4) ^ right) & 0x0F0F0F0F;
right ^= work;
left ^= (work << 4);
work = ((left >> 16) ^ right) & 0x0000FFFF;
right ^= work;
left ^= (work << 16);
work = ((right >> 2) ^ left) & 0x33333333;
left ^= work;
right ^= (work << 2);
work = ((right >> 8) ^ left) & 0x00FF00FF;
left ^= work;
right ^= (work << 8);
right = ((right << 1) | ((right >> 31) & 1)) & 0xFFFFFFFF;
work = (left ^ right) & 0xAAAAAAAA;
left ^= work;
right ^= work;
left = ((left << 1) | ((left >> 31) & 1)) & 0xFFFFFFFF;
```

Algorithm 5.1.2 IP^{-1}

```
right = (right << 31) | (right >> 1);
work = (left ^ right) & 0xAAAAAAAA;
left ^= work;
right ^= work;
left = (left << 31) | (left >> 1);
work = ((left >> 8) ^ right) & 0x00FF00FF;
right ^= work;
left ^= (work << 8);
work = ((left >> 2) ^ right) & 0x33333333;
right ^= work;
left ^= (work << 2);
work = ((right >> 16) ^ left) & 0x0000FFFF;
left ^= work;
right ^= (work << 16);
work = ((right >> 4) ^ left) & 0x0F0F0F0F;
left ^= work;
right ^= (work << 4);
```

As an example of the mapping processes used to convert the above pseudocode (using C constructs) to hardware, let us examine the first three instructions of the IP stage. Only three DPUs, each with a single configuration, are required to implement this code in hardware as shown in Figure 5.3. For this hardware to work correctly, the

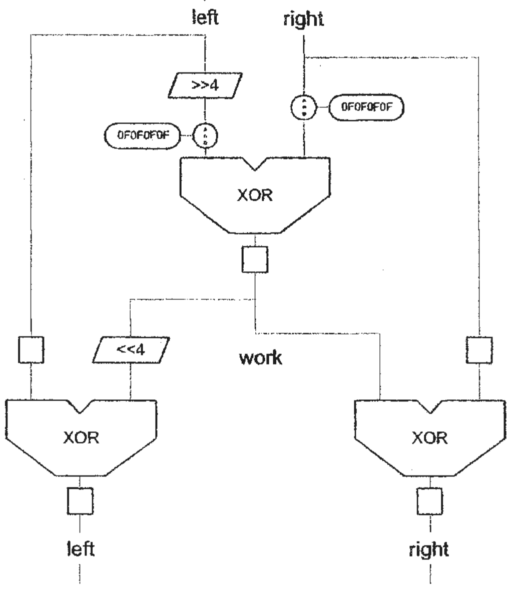

Figure 5.3: Example Hardware Mapping for a Subset of IP Stage Pseudocode

inputs on the DPUs performing steps 2 and 3 needed to buffer the initial values of *left* and *right* to allow for the 1 clock cycle delay in calculating the *work* value. Continuing the mapping process as described above, the the IP and IP^{-1} were mapped to the CS2112 fabric. The complete hardware configuration for the IP stage is shown in Figure 5.4. All of the DPUs shown require only a single configuration and no control inputs. Also, this section of hardware is able to operate on a continuous stream of data on the two inputs and will produce a continuous stream of output data after an initial 14 clock cycle delay. The complete IP stage data path Verilog module is given in Appendix C as a further example of CS2112 design implementation.

Inner Round

Since the operations inside of the F-function operate on data of widths larger than 32-bits, it could not be implemented directly using fabric resources. Instead, the E

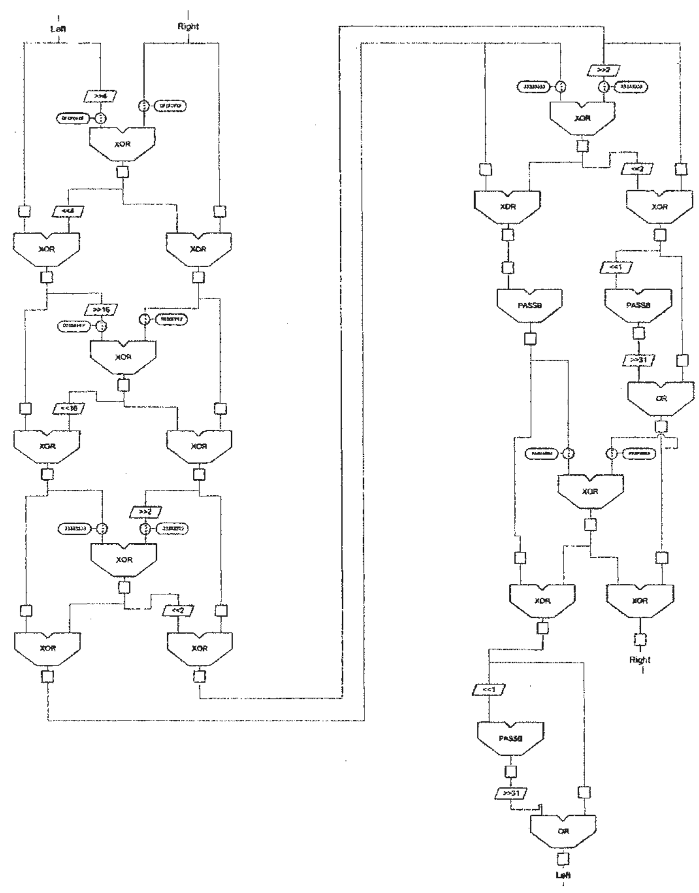

Figure 5.4: IP Hardware Configuration

expansion, subkey addition, S-box substitution and P permutation were performed as in the C code model and the pseudocode for these operations is presented in Algorithm 5.1.3. In this case, the E expansion is accomplished by passing the odd and even S-boxes modified versions of the 32-bit *right* operand. This requires a modified key generation algorithm that splits each round subkey into two 32-bit values instead of a single 48-bit value. Therefore, the algorithm now uses 2 subkeys per round for a total of 32 subkeys. The odd S-boxes are passed segments a 4-bit rotated version of the *right* operand which has been XORed with the first round subkey. The even S-boxes are passed segments of a non-rotated version of *right* which has been XORed with the second round subkey. These two groups of operations perform the E expansion and subkey addition portion of the F-function.

Algorithm 5.1.3 F-Function Internals

```
work  = (right << 28) | (right >> 4);
work ^= *keys++;
fval  = SP7[ work        & 0x3fL];
fval |= SP5[(work >>  8) & 0x3fL];
fval |= SP3[(work >> 16) & 0x3fL];
fval |= SP1[(work >> 24) & 0x3fL];
work  = right ^ *keys++;
fval |= SP8[ work        & 0x3fL];
fval |= SP6[(work >>  8) & 0x3fL];
fval |= SP4[(work >> 16) & 0x3fL];
fval |= SP2[(work >> 24) & 0x3fL];
left ^= fval;
```

Also, in the DES algorithm description, the S-boxes produced 8 4-bit outputs which then passed through the P permutation to form the 32-bit F-function output. As stated previously, bit permutations are not efficiently implemented in the 32-bit data path available. However, in this case, it is possible to combine the permutation into the S-Box outputs, as shown in Figure 5.5, since the LSMs produce 32-bit values. Hence, the inner round function can be further subdivided into three blocks as shown

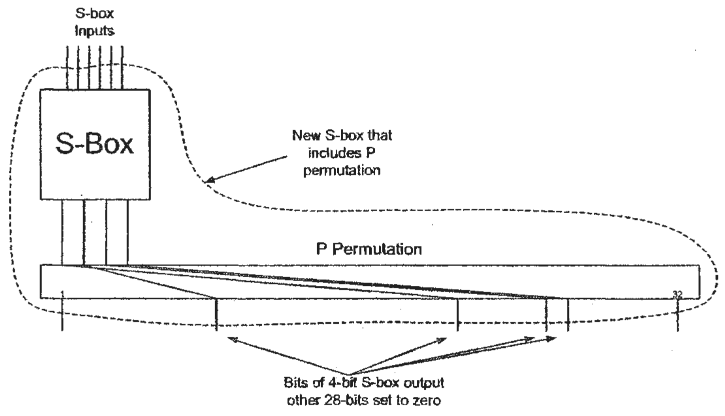
S-box Inputs
S-Box
New S-box that includes P permutation
P Permutation
32
Bits of 4-bit S-box output other 28-bits set to zero

Figure 5.5: S-box and P Permutation Combination

in Figure 5.6. The S-boxes, which form the main component of the inner round, are implemented in LSMs that contain 32-bit values with a single DPU for access. These DPUs perform shift and mask operations on their inputs so that the proper 6-bit segment of the input occupies bits 2-7 of the LSM address. The segment must be moved to bits 2-7 of the address since the DPU can only be bytewise addressed. As

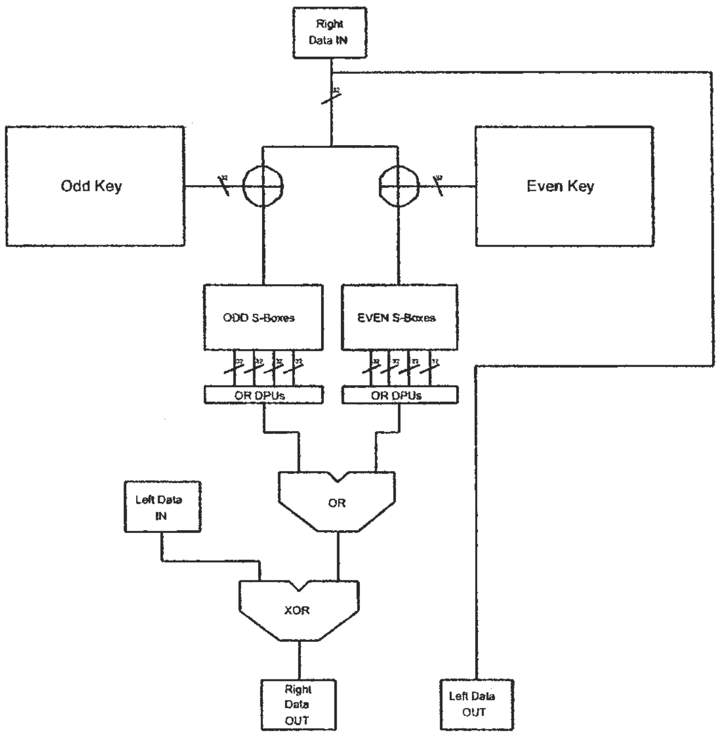

Figure 5.6: DES Round Function Block Diagram

an example, S-Box 7 requires bits 0-5 of the *work* variable. Therefore, the addressing DPU must shift the work variable by 2 bits to the left and then mask the input with $0xfc$.

Also, in this portion of the kernel, key production units handle the loading of subkeys into the data path so that they can be XORed with the input for the round. A simple control unit handles the sequencing of this operation. As well, a number of DPUs are required to perform the above mentioned XORs, the assembling of the S-Box outputs into a single 32-bit value, and the XOR of the output of the round with the other half of the input data. These units comprise the supporting logic block. The total configuration for the inner round portion of the fabric is shown in Figure 5.7. It should be noted that two configurations are required for the key generation DPUs to generate and hold the subkey values.

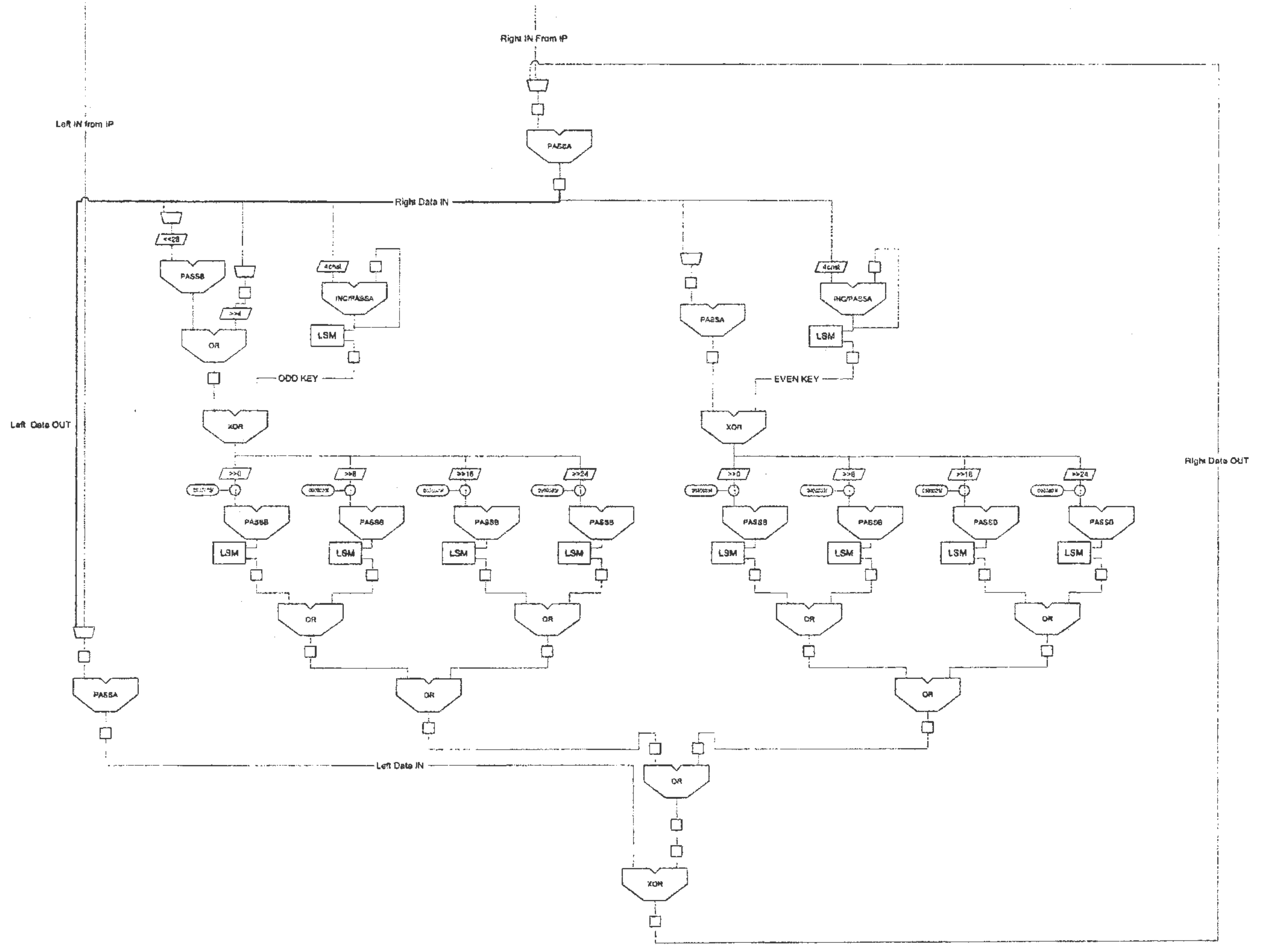

Figure 5.7: DES Round Function Fabric Configuration

Control

The control unit for this kernel was divided into a master controller and a number of slave controllers. The master controller receives the start signal from the ARC, sequences the slave controllers and generates the done signal once the encryption has been completed. The IP and IP^{-1} portions of the kernel each have a simple sequencer which signals the master controller when the data has finished passing through each stage. The inner round portion of the kernel is comprised of three slave controllers which handle the iteration of the data through the multiple rounds and the subkey sequencing.

5.1.2 Synthesis and Mapping

As this was the first complex implementation I developed for the CS2112, there were a number of problems which prevented the placement and routing of the kernel on fabric. Although there were enough functional units (DPUs, LSMs, etc) available to accommodate the kernel, the limited global data and control routes prevented the completion of the mapping process. As well, the FSMs in a slice only have access to the PLAs within a tile and limited inter-slice routes are available for the communication of state bits between tiles in a slice. The master and slave controllers developed required more product term resources and inter-slice routes than available in their slice and could not be mapped to the fabric.

Although the synthesis process was not completed and a complete kernel produced, the DPU, LSM, and MUL utilization can be found via the Verilog model. However it is difficult to obtain an accurate estimate of the PLA and global route usage without a finalized kernel. The data in Table 5.1 gives a summary of the fabric utilization for the design. It should be noted that this design uses 96% of the available DPUs in the fabric.

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Totals
DPU	20	21	20	20	81
LSM	2	5	5	2	14
MUL	0	0	0	0	0

Table 5.1: Iterative Kernel Resource Utilization

5.1.3 Testing and Performance

Since a working kernel could not be completed for this design, it could only be tested at the Verilog model stage. The completed Verilog model was tested via a behavioral Verilog testbench using test vectors generated by the C model used with the design. The contents of the data output LSMs could not be accessed directly by the testbench. Therefore, the data values entering the data output module were used to verify correct operation, assuming the LSM writing process occurred without error.

The performance of the kernel was estimated from the Verilog model by determining the number of clock cycles to process a single 64-bit plaintext in the VerilogXL simulator. Approximately 233 clock cycles were needed to encrypt a single 64-bit plaintext. Hence if the CS2112 is running at 100 MHz the throughput is approximately 27.5 Mbits/sec. It should be noted, however, that this performance figure does not include the overhead involved with loading the kernel into the fabric as well as the time required to load the data into and out of the fabric.

5.2 Multiple Pipelined Kernel

The encryption of singular plaintexts, as discussed in the previous section, limits the performance capability of a kernel since it does not exploit the pipelining or parallelizing capability of the reconfigurable logic. Therefore, a second design that attempted to process a stream of plaintexts was developed. Initial hardware estimates indicated that a completely pipelined and loop unrolled version the entire DES algorithm would exceed the hardware available in the CS2112 fabric. For example, a fully loop unrolled and pipelined kernel would require $16 \times 8 = 128$ S-Boxes, each containing $64 \times 4 = 256$ bits of data. Due to limitations in LSM addressing and size, only a single S-Box can be placed in an LSM so a total of 128 LSMs would be needed for a fully unrolled and pipelined design. With only 48 LSMs available on the fabric a fully unrolled version is an impossibility.

As stated in Section 4.1, the CS2112 fabric has two configuration planes that can be swapped in a single clock cycle. When this swap occurs, the data located in the LSMs remains in place and is available to the new active configuration for processing. By using this dynamic reconfiguration ability and splitting the DES algorithm across multiple kernels, each of the individual kernels would have access to additional fabric resources. This extra available hardware can then be used to improve the throughput of the individual kernels and, in turn, the complete function via the pipelining of data path elements. As stated above, the DES algorithm is naturally partitioned into three main blocks. These blocks - IP, IP^{-1} and the "Inner Rounds" - were implemented as individual kernels, each of which processes a maximum of 128 plaintexts before the following kernel is made active or the IP^{-1} kernel completes. Figure 5.8 illustrates the DES kernel swapping process. The IP and IP^{-1} kernels require a block of 128 64-bit

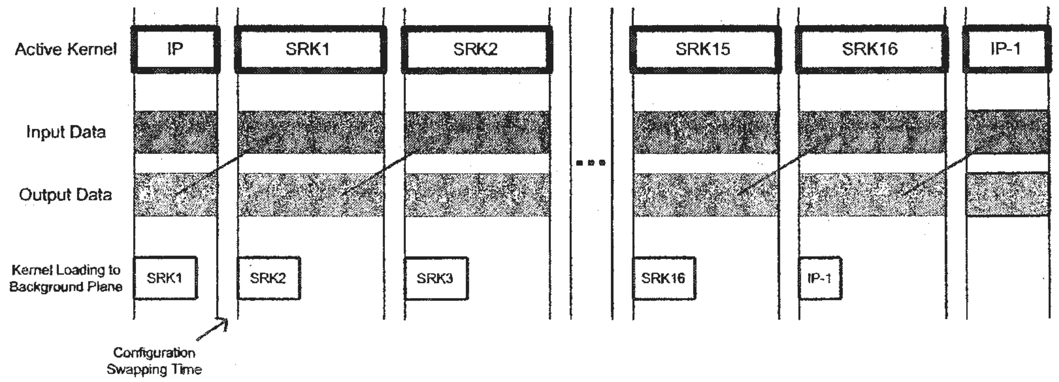

Figure 5.8: Pipelined Multi-Kernel DES Kernel Swapping

data values at their input and they produce 128 64-bit outputs. The "inner rounds" are implemented as Single Round Kernels (SRKs) that complete a single round of

DES per invocation. For reasons discussed in the following sections, two different SRK fabric layouts are needed to produced the desired output, although they are functionally identical.

As with the iterative design discussed in Section 5.1 the key scheduling algorithm was not implemented in the CS2112 fabric. Instead, the subkeys were calculated by the software application and the subkeys relevant to a particular invocation of the SRK were passed into the function for use. As well, all 128 blocks had to be encrypted using the same key.

5.2.1 Architecture Descriptions

The iterative kernel architecture described above was used as a basis for the development of a pipelined multi-kernel design. The IP and IP^{-1} stages previously developed already supported pipelined data and required little modification to produce individual IP and IP^{-1} kernels. The SRK required more modification to the previously developed inner round hardware section since it only supported processing a single word at a time. The following sections describe these kernels in more detail.

IP and IP^{-1} Kernels

As stated above, the previously developed portions of the iterative kernel already were able to process pipelined input data. These sections of the data path hardware were separated from the iterative design and modified to produce a stand alone kernel for both the IP and IP^{-1} stages. Hence, this new design needed to process a total of 128 plaintexts and each individual kernel must therefore process 128 input data blocks. Data input units that produced a stream of 128 plaintexts were added to the beginning of the previously developed data paths. As well, an output unit was added to accommodate the storage of a stream of 128 outputs in an LSM.

The above kernels required very simple control hardware to process a stream of data. After the start pulse was received, each kernel's data input unit would begin producing the input data from an LSM. Once a counter determined that the head of the data had reached the output unit, it was told to begin writing to the output LSM and after all 128 blocks were stored the done signal was asserted.

Single Round Kernel

Of the three kernels developed in this design, the SRK involved the most modification to the previously developed iterative data path and control elements. In the previous design, the left side data was held in a DPU register before being XORed with the data that passed through the F-function portion of the data path. Also, the right side data was held since it was to become the new left side data in the next round. Therefore, the previously developed hardware could only support the processing of a single data value during an iteration. However, since the SRK was to process multiple data values, it would be more efficient to process more than one data value at a time. After the development of the iterative kernel it was obvious that the fabric could not accommodate the hardware required to implement all 16 inner rounds of DES in the SRK. Instead of processing a continuous stream of data as in the IP and IP^{-1} kernels, the SRK was designed to process 128 64-bit inputs in smaller blocks.

Internally, the hardware was arranged to form a circular pipeline as illustrated in Figure 5.9. The SRK contained 13 pipeline stages and was able to accommodate the

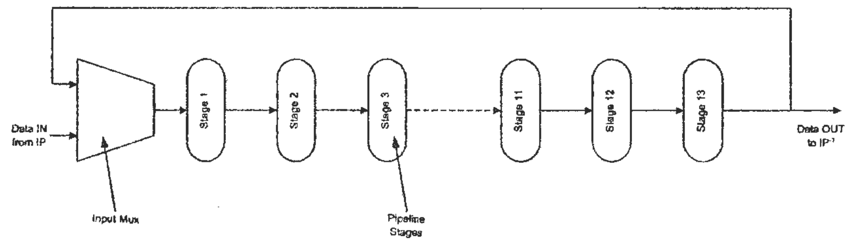

Figure 5.9: Circular Pipeline Concept

processing of a block of up to 13 data values at a time. Hence, to process all 128 inputs the SRK would have to process 9 full subblocks (i.e. 13 data values) and one partial subblock of data. To create such a circular pipeline, two delay structures were added to the right and left side of the data path as shown in Figure 5.10. Each of

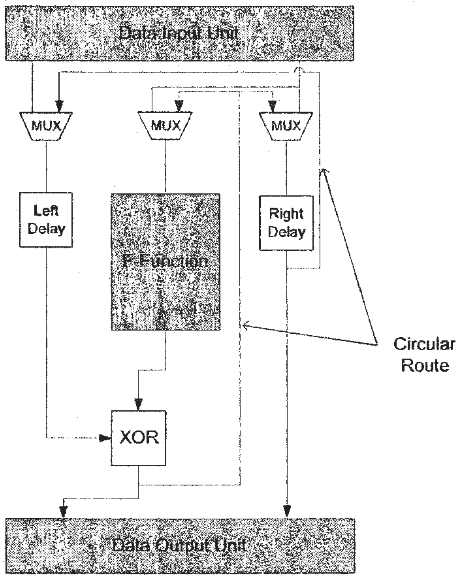

Figure 5.10: SRK Delay Illustration

these units used an LSM buffer unit to delay the data by 11 clock cycles on the left and 12 on the right. The one clock cycle difference was necessary since the left data was XORed with the output of the S-boxes before the round is complete. Other than the addition of these delay units, no further modifications to the iterative data path were necessary to complete the SRK data path.

The control unit in the SRK case was very similar to the portion of the previously developed iterative control unit. In this case, the control unit was again responsible for sequencing the production of the subkeys from the key generation units. Now the control unit was also responsible for the loading and writing of blocks of data into and out of the circular pipeline. Since each kernel is completing only a single round of the algorithm, there was no need for counting the round number in this case.

5.2.2 Synthesis and Mapping

The three kernels involved in this design were more easily mapped to the CS2112 hardware as a result of their smaller size when compared with the unified iterative design. As well, since less control logic was required to sequence each kernel's operation, no problems were encountered with control logic synthesis and mapping. However, a modification had to be made at this stage of the design process due to a property of kernel swapping. As stated above, when a kernel is swapped from the background plane into the active plane the LSM contents remain in place. This meant that the SRK kernels could not be identical from one round to the next. Instead, two separate mappings of the kernel hardware were developed so that the next kernel to become active would read its data from the LSM to which the current active kernel was writing data. No functional modifications were required to accommodate these separate mappings. Tables 5.2 through 5.4 detail the resource utilizations of the three kernels involved in this design. Notice from these tables that the hardware utilization within each kernel is substantially lower that the previously discussed iterative design. Also, note that the delay structures added to the SRK did not significantly increase the kernels hardware utilization.

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Total
DPU	20	4	0	0	24
LSM	2	2	0	0	4
MUL	0	0	0	0	0
State Bits	3	0	0	0	3

Table 5.2: IP Kernel Resource Utilization

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Total
DPU	16	14	0	0	30
LSM	6	6	0	0	12
MUL	0	0	0	0	0
State Bits	14	8	0	0	22

Table 5.3: SRK Kernel Resource Utilization

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Total
DPU	20	7	0	0	7
LSM	2	2	0	0	4
MUL	0	0	0	0	0
State Bits	3	0	0	0	3

Table 5.4: IP^{-1} Kernel Resource Utilization

5.2.3 Testing and Performance

After the synthesis and mapping process was completed, the kernels were linked to application code for testing. In this case the eBIOS library calls were written manually into the C code as opposed to using the `\#pragma` calls discussed in Section 4.2.4 because the provided tools did not handle kernel swapping efficiently. The design was tested by first generating plaintext/ciphertext pairs, using the software version of the function, and then checking the hardware outputs against these values. Using both the chip simulator and development board, the finished application was tested and returned correct results in all cases.

The overall performance of the final application running on the development board could not be accurately measured due to a software "bug" in the libraries provided by Chameleon Systems with their development environment. Since the company stopped production of the CS2112 in early 2002, a fix was not provided by the company. Instead, the performance of the kernels was estimated from the Verilog model. As before, this does not take into account the kernel configuration loading overhead nor does it take into account the overhead involved in switching kernels. Although the kernels could be swapped from the background plane into the active plane in a single clock cycle and the data being processed remained in the LSMs, after the SRKs were swapped new subkeys had to be loaded into the kernel's key production LSMs. The IP kernel and and IP^{-1} kernels required 146 and 151 clock cycles respectively to process 128 64-bit plaintexts. The SRKs required 2380 clock cycles to process 128 64-bit plaintexts. Hence, ignoring configuration swapping overhead a total of

2677 clock cycles were required to encrypt 128 plaintexts using the multiple pipelined kernels. Therefore, with the CS2112 running at 100 MHz, the overall throughput was approximately 306 Mbits/sec.

5.3 Pipelined Kernel

The design and implementation of a pipelined multiple kernel version of DES led to the development of a third implementation. Although the IP, IP^{-1} and single round kernels of the previous design each occupied a significant portion of the reconfigurable fabric, an attempt was made to unify these kernels and improve the overall performance of the design. Again, as described previously, this kernel was to process a complete block of 128 plaintexts using a single key. Also, as with previous designs the round subkeys were to be calculated by the ARC processor and passed into the kernel for processing. However, in this case no kernel swapping was required so the overhead involved in switching kernels was eliminated.

5.3.1 Architecture Description

The data path portion of the pipelined kernel simply reused the hardware from the multiple kernel design. The IP, IP^{-1} and SRK kernels were combined into a single design. As with the SRK, the kernel processed 128 inputs in blocks of 13 plaintexts. However, the IP and IP^{-1} portions of the kernel now processed these smaller blocks instead of the continuous stream of data that their stand alone kernel forms processed. Figure 5.11 shows a snapshot of the unified design's data path. The majority of the work involved in this design came in the form of control unit development and careful hardware mapping. The control unit had to perform 4 major tasks. Firstly, the controller had to signal the input data generators to output a block of 13 plaintexts. Secondly, it had to determine when a block of data had completely passed through

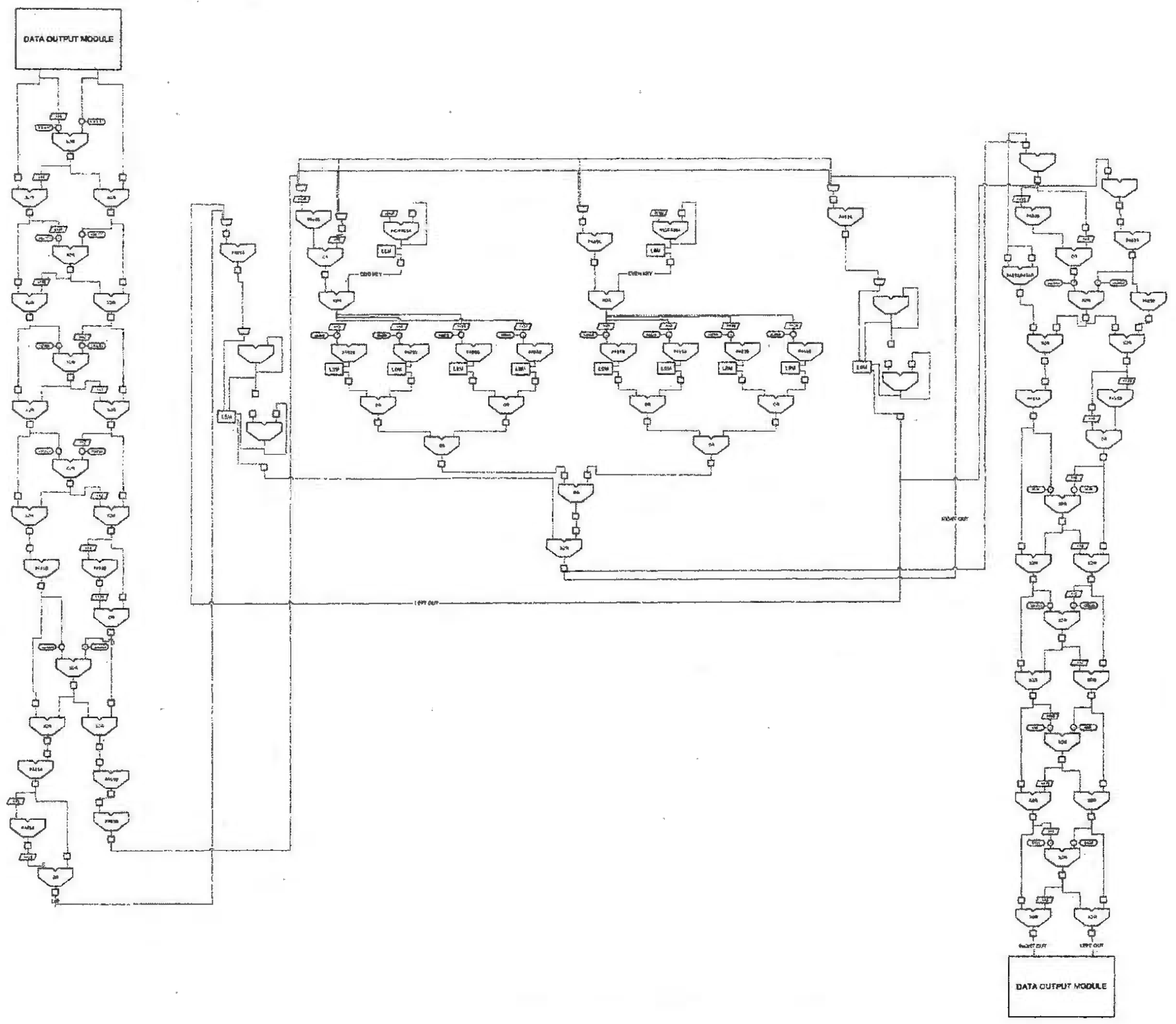

Figure 5.11: Pipelined DES Data Path

the IP stage so that the circular pipeline performing the round calculations could close. Thirdly, the control unit had to signal the key generation hardware to produce the correct sequence of round subkeys. Finally, when the data that had completed all 16 rounds of processing passed through the IP^{-1} stage of the algorithm, the data writer units were signalled to write the block of 13 plaintexts to LSMs. Since the data path occupied 80 of the 84 available DPUs in the fabric and the free DPU locations were fixed, it was very difficult to develop a controller that used DPUs as counters to sequence operations. The first control unit developed used entirely state bits and PLA resources to sequence tasks, but it quickly overloaded the control resources available since a large number of states were required. Therefore, a second design was developed that utilized the remaining free DPUs to perform counting operations. While this change complicated control unit placement, it simplified the control hardware sufficiently to allow for the synthesis and mapping stage to proceed.

5.3.2 Synthesis and Mapping

As with the previous designs, after the architectural issues were resolved, the Verilog design was synthesized and mapped onto the CS2112 after it was fully tested. The manual mapping process required much more time to complete with this design since approximately 98.8% of the DPUs were used and a significant amount of global data and control routing was required. However, the design was eventually successfully mapped to the fabric with the final floorplan as shown in Figure 5.12. Table 5.5 shows the overall resource usage for the pipelined kernel.

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Total
DPU	21	21	21	20	83
LSM	2	6	6	2	16
MUL	0	0	0	0	0
State Bits	9	20	3	5	27

Table 5.5: Pipelined Kernel Resource Utilization

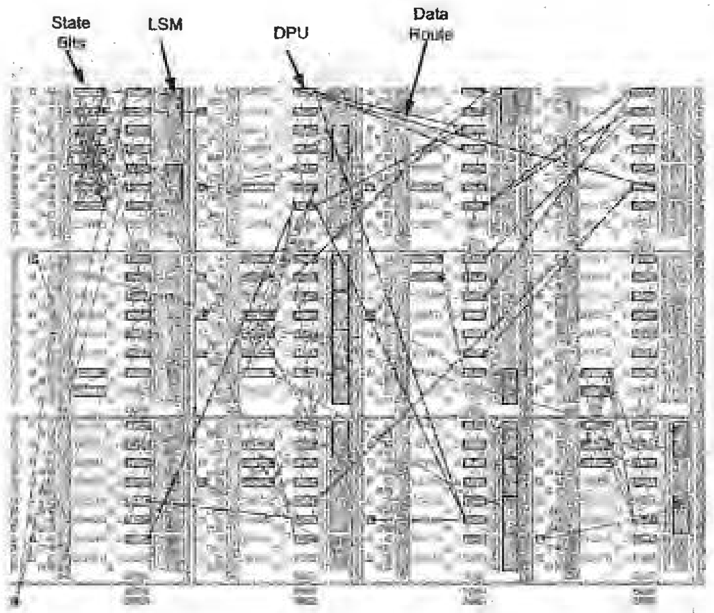

Figure 5.12: Pipelined DES Floorplan

5.3.3 Testing and Performance

After the kernel bitstream was produced, it was linked to the previously developed application test code. The Verilog testbench and C application test code for the pipelined design are given, as an example, in Appendix D. In this case, however, the kernel did not perform encryptions correctly and was corrupting data. After some investigation, it was determined that the kernel itself was correct but a hardware or software “bug” prevented the correct routing of data inside a slice. This error was eventually attributed to the v2b compiler and could not be resolved. Hence the completed application could not be fully tested.

For the above reason, and for the problem discussed in Section 5.2.3, the performance of the kernel had to be estimated based on the Verilog model. The pipelined kernel required 2540 clock cycles to encrypt 128 64-bit plaintexts. Hence, with the CS2112 running at 100 MHz, the kernel throughput was estimated to be 322.5 Mbits/sec.

5.4 Summary

The above sections presented three implementations of DES on the Chameleon CS2112. A 32-bit C implementation was used as the basis for all three kernels. This code performed the IP and IP^{-1} operations using a series of 32-bit instructions and modified the F-function operations so that 32-bit operands could be used. The first kernel encrypted a single 64-bit plaintext and did not utilize the CS2112 fabric's pipelining capability. This implementation, although not fully completed, was estimated to have a throughput of 27.5 Mbits/sec. The next kernel attempted to simplify the overall design by splitting the implementation into three kernels. This gave each kernel access to a greater number of fabric resources and simplified the placement and routing process. Also, each of these kernels utilized the pipelining capability of the CS2112 fabric to improve performance. When combined, the throughput of the multiple kernel design was estimated at 306 Mbits/sec. The final design attempted to unify the multiple kernels developed in the previous implementation. The unified design showed an improved throughput of 322 Mbits/sec.

Chapter 6

Rijndael Implementation

In this chapter, we examine the development of two different implementations of Rijndael, the cipher selected as the Advanced Encryption Standard by the United States National Institute of Standards and Technology. The development of the Rijndael kernels began with a 32-bit software model that had a fixed key length of 128 bits and plaintext block length of 128 bits. From Table 2.4 it can be seen that a total of 10 rounds were required to perform an encryption. Initially it may look somewhat complex to implement the round operations on a 32-bit machine efficiently. However, the Rijndael specification document [17] details a method by which the entire round function can be replaced by lookups into four 256-entry tables (T0-T3) with entries that are 32-bits wide.

The Lookup Table (LUT) strategy can be described by the following equation for the round function:

$$e_j = k_j \oplus T_0\left[a_{0,j}\right] \oplus T_1\left[a_{1,j-C1}\right] \oplus T_2\left[a_{2,j-C2}\right] \oplus T_3\left[a_{3,j-C3}\right]$$

Where e_j is the column of the output $STATE$ for that round, $a_{i,j}$ is the input $STATE$ at row i, column j, k_i is the column of the expanded subkey, and T_k is a LUT. The values of $C1$, $C2$ and $C3$ are fixed at 1, 2, and 3 respectively. Hence for a 128-bit input plaintext, 16 table lookups are required to find the 4 columns of the output $STATE$ per round.

As stated in the algorithm description, in the final round the mix column step is removed. There are two methods to perform this final round using LUTs. Firstly, a second set of tables ($TFINAL0 - TFINAL3$) can be used for the final round with again 16 lookups into four 256-entry tables with each entry 32-bits wide. Secondly, the same set of tables can be used for all rounds; in the final round bit masks and shifts can be used to eliminate the mix column step from the table outputs. Decryption can be accomplished in a similar way using a different set of LUTs. Fortunately, numerous C-code models were freely available and an implementation written by Vincent Rijmen was chosen. In this code, LUTs were used to perform the round function and the final round was accomplished using a second set of tables. This code served as a basis for the development of the two kernels discussed in the following sections.

6.1 Iterative Kernel

As with the previously discussed DES implementations the development of a Rijndael kernel began with a purely iterative implementation. This kernel was to process a single 128-bit plaintext using a 128-bit key. Since the number of DPUs required to perform the round operations was too large to fit into the CS2112 fabric, a LUT strategy was used with a second set of tables to implement the final round. Hence, 2 LSMs were required to implement each T table and 2 LSMs for each $TFINAL$ table for a total of 16 LSMs to represent both round functions. To perform a complete round in parallel a total of 64 LSMs would be required because 16 lookups were needed in parallel (4 lookups into 4 sets of tables) with 32 LSMs for the regular round and 32 LSMs for the final round functions. However, only 48 LSMs were available on the fabric, and only a partial round could be completed in parallel.

The first design focused on an iterative kernel, with no pipelining ability, that used a single set of tables for the regular round and another set for the final round. Each pair of tables, T_i and $TFINAL_i$, was connected to a single address generator DPU with the values of T_i in the lower 256 locations and $TFINAL_i$ in the upper 256 locations. Therefore, in the final round 0x00000400 was added to the input address to switch to the upper table outputs. Figure 6.1 illustrates these two address generator

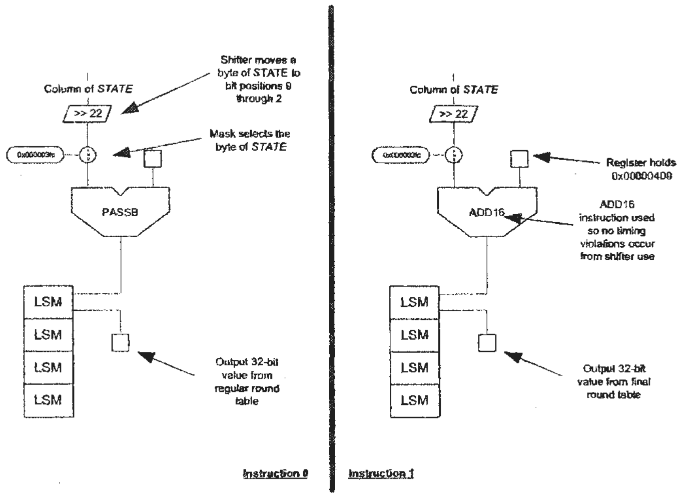
Column of STATE
>> 22
Shifter moves a byte of STATE to bit positions 9 through 2
0x000003fc
Mask selects the byte of STATE
PASSB
LSM
LSM
LSM
LSM
Output 32-bit value from regular round table
Instruction 0
Instruction 1
Column of STATE
>> 22
0x000003fc
Register holds 0x00000400
ADD16
ADD16 instruction used so no timing violations occur from shifter use
LSM
LSM
LSM
LSM
Output 32-bit value from final round table

Figure 6.1: Table Address Generator Instructions

DPU configurations.

After the design of the table lookup units was complete, it was a fairly simple matter to complete the design. Figure 6.2 shows the DPU connections required to perform an encryption (NOTE: the muxing layer DPU inputs are round dependant). Three additional LSMs were used to hold data. To simplify the initial "add round key" operation, the 4 plaintext columns (32 bits each) were stored in an LSM. The 11 128-bit expanded round keys were also stored in an LSM as 4 32-bit values occupying 44 locations in the LSM. Each entry in the key LSM was basically a column of the expanded key. As with DES, the subkey generation algorithm could not be implemented in parallel with the round function so subkeys were generated in software on the ARC processor and passed to the fabric. Finally, the ciphertext exited the

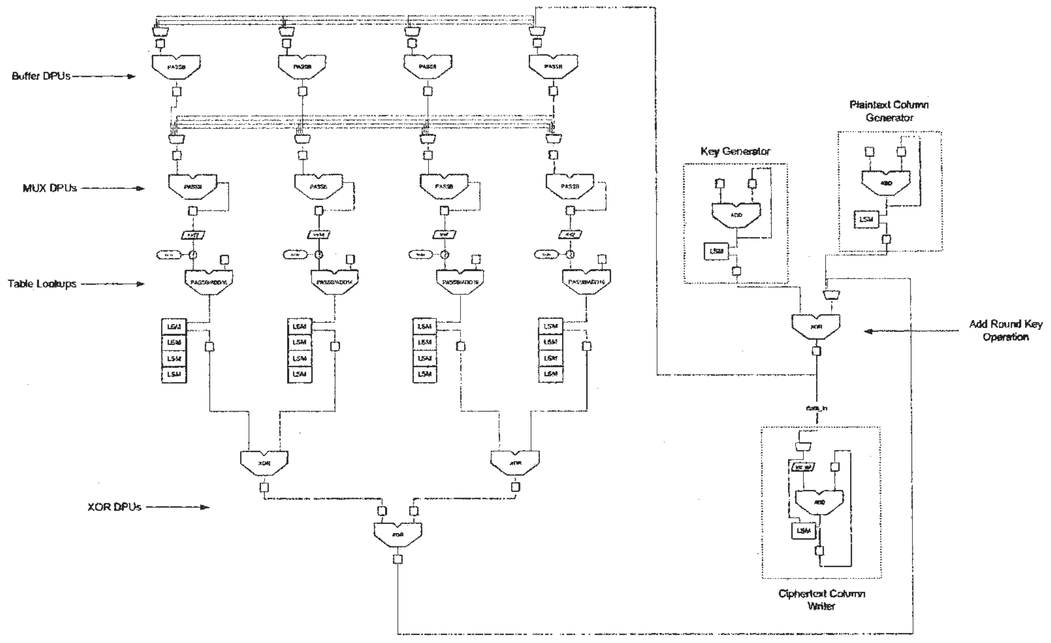

Figure 6.2: Iterative Rijndael Kernel Data Path Configuration

round function as a series of columns that were written to a third LSM.

A round of Rijndael proceeded in a number of stages and can be followed in Figure 6.2. The buffer DPUs were used to hold the columns of the *STATE* for the current round. The buffer DPUs loaded their particular column of *STATE* as the column exited "add round key" operation. Once all four columns had been loaded into the buffers, the MUX DPUs perform a sequence of operations to load the correct columns of *STATE* into its associated table lookup unit. The table outputs were XORed and then passed through an "add round key" operation, with 4 columns passing through sequentially. This output was then either written as ciphertext, if 10 rounds had completed, or passed through another round as described above. Control logic was added to the design to sequence these operations.

6.1.1 Synthesis and Mapping

The iterative kernel was synthesized and mapped using the C~Side^{TM} tools without any major difficulty. A view of the finalized floorplan taken from the manual routing tool is shown in Figure 6.3, along with a summary of the hardware usage in Table 6.1. Note that roughly 50% of the fabric resources were used by the iterative kernel.

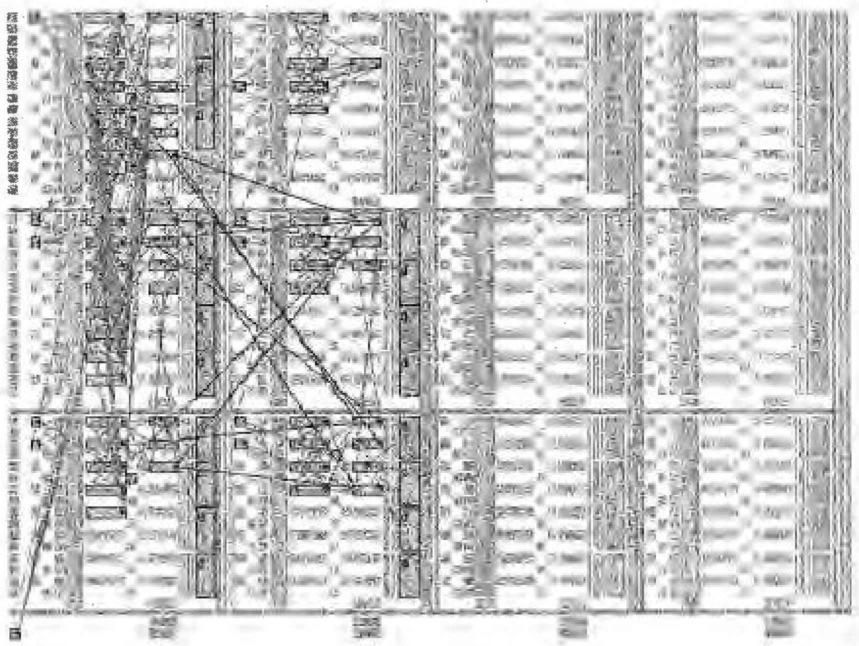

Figure 6.3: Iterative Rijndael kernel floorplan

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Total
DPU	12	8	0	0	20
LSM	11	8	0	0	19
MUL	0	0	0	0	0
State Bits	27	14	0	0	41

Table 6.1: Resource utilization for iterative Rijndael design

6.1.2 Testing and Performance

As with the DES kernels, a software bug prevented measurement of the AES iterative kernel's performance in the final application. Instead, the clock cycles to complete a 128-bit plaintext encryption were found from the Verilog simulation. The iterative kernel can encrypt a single 128-bit plaintext with a 128-bit key in 170 clock cycles. Therefore, ignoring configuration overhead, the throughput was approximately 75 Mbits/sec at a clock speed of 100 MHz.

6.2 Pipelined Kernel

Since the iterative kernel resource utilization was relatively low, a pipelined version of the kernel was investigated. To develop a fully pipelined version of 10 round Rijndael, each round of table lookups would have to be done in parallel and all 10 rounds would have to fit on the fabric. Since each round needed 16 lookups, 160 tables were required. Each table would occupy 2 LSMs in the CS2112 fabric so a total of 320 LSMs would be needed to fully pipeline the algorithm. Obviously, this is not possible with the current chip. Therefore, a circular pipeline strategy was used similar to that used in the DES design.

The initial goal was to be able to output a single column of the output $STATE$ in a slice. This required the completion of all four table lookups within a single slice. In the iterative design, two tables were used to represent the two round functions and this required 16 LSMs to store the tables. In order to fit all 4 lookups in a single slice, the second method of computing the final round, mentioned in Section 6, was used. To achieve this, the operations performed by the lookup table DPUs and the XOR DPUs had to be changed. As well, each of the tables would have to take in a different input. Figure 6.4 illustrates these changes in the third table lookup DPU and XOR DPU following it. Similar modifications were made to the rest of the DPU configurations.

After these modifications, it was possible to compute an entire column of the output state in a single slice. Hence, across four slices, the full 4 columns of the

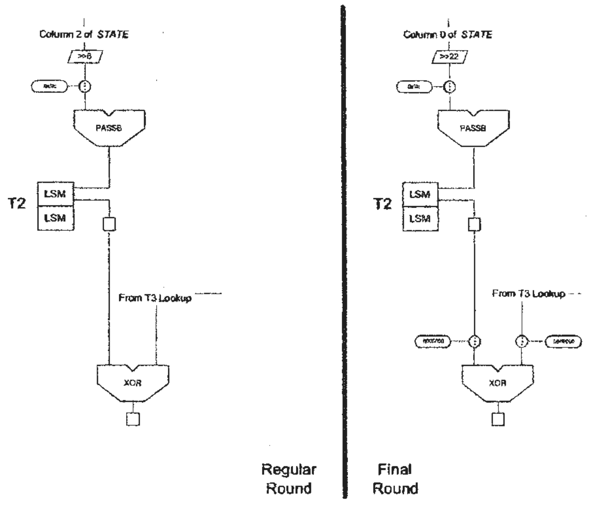

Figure 6.4: Pipelined Table Address Generator Instructions

output $STATE$ could be computed. The design was, therefore, partitioned into four distinct modules with each module computing a single column of e. Each of these modules included an LSM holding a number of columns of state, another LSM holding the expanded keys for that column, and a third LSM for holding the completed ciphertext. Figure 6.5 shows the DPU/LSM connections for the data path of slice 0. (Note: final round configuration is not shown) This structure was essentially the same as shown previously in the iterative design, but now there was no need for a muxing layer. The circular pipeline in this case can hold up to 10 blocks of data. This structure was then repeated across the other three slices to complete the kernel.

Within a slice, data flowed through a number of stages to complete a round and can be followed in Figure 6.5. The block of 128-bit input plaintexts were divided into columns and loaded into their appropriate LSMs, with the slice 0 plaintext column generator holding input data column0, slice 1 holding column1, and so on. A block of 10 columns was then started out of the plaintext column generator in each slice. After exiting the column generator, the columns of $STATE$ passed through an initial "add round subkey" operation. This occurred in all slices simultaneously. After exiting this "add round key" operation, the blocks passed through a buffer DPU.

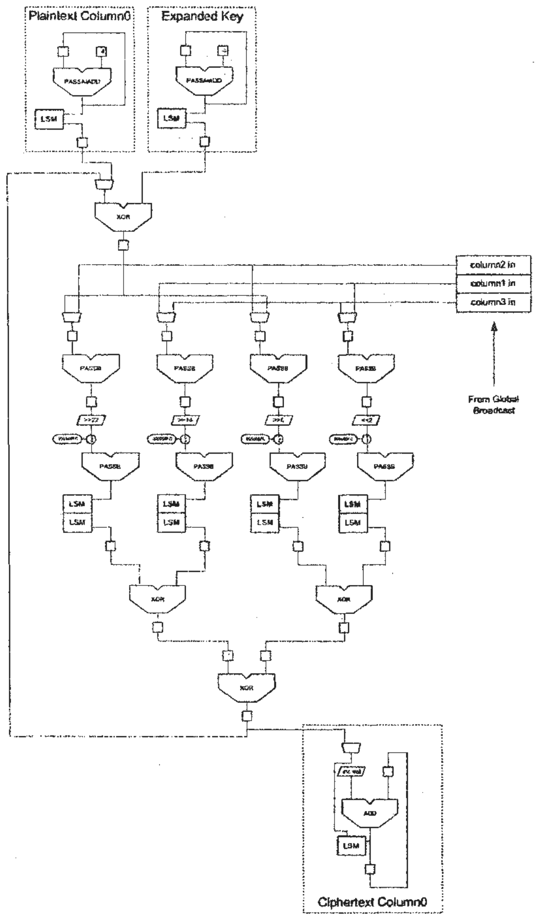

Figure 6.5: Slice 0 Data Path Configuration for Regular Round

The buffers held all four of the columns of $STATE$ that were exiting the initial "add round subkey" operation in each slice. Three of these columns were passed to the slice via global connections with the other slices. On the next clock cycle the columns of $STATE$ in these buffer DPUs were passed to the LUT portion of the pipeline. After passing through the "add round subkey" operation again, the new columns of $STATE$ were loaded into the buffers as described above. To complete the final round, the instructions associated with the buffers, LUTs and XOR units were changed as the head data in the pipeline began to pass through. The completed ciphertext columns in each slice were then written to LSMs before being passed back to the ARC processor.

6.2.1 Synthesis and Mapping

Since the pipelined design was slice based, a single slice of the design was first synthesized, mapped and tested before a complete kernel was produced. The single slice was mapped to the fabric without any problems. When the other slices were added, however, the limited global routing lines required that the resources in slices 2 and 3 be mapped as a mirror image of slices 0 and 1. The finalized floorplan is shown in Figure 6.6. Table 6.2 lists the hardware usage for the pipelined kernel across all slices. Note that the pipelined kernel DPU usage jumped to almost 95%.

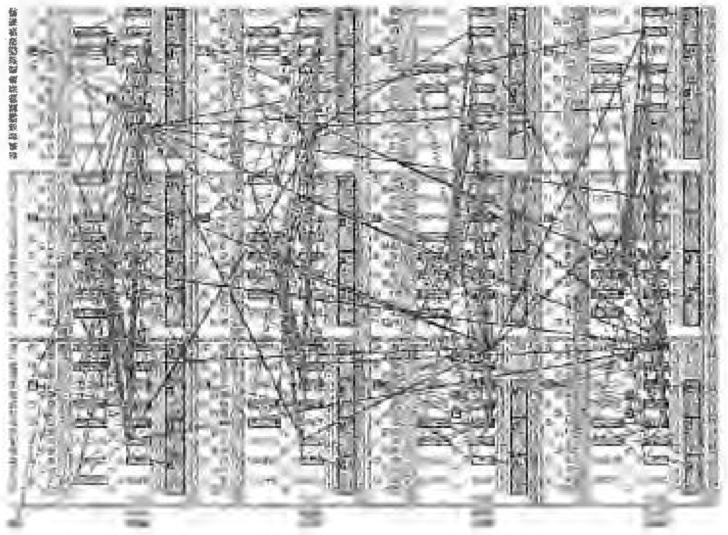

Figure 6.6: Pipelined Rijndael kernel floorplan

Resource	Slice 0	Slice 1	Slice 2	Slice 3	Total
DPU	21	20	20	20	81
LSM	11	11	11	11	44
MUL	0	0	0	0	0
State Bits	17	12	12	12	53

Table 6.2: Resource utilization for pipelined Rijndael design

6.2.2 Testing and Performance

Again using the Verilog simulation results, the pipelined Rijndael kernel could encrypt ten 128-bit plaintexts using the same 128-bit key in 114 clock cycles. Therefore ignoring configuration overhead, this kernel had a throughput of 1.1 Gbits/sec if the CS2112's clock speed was 100 Mhz.

6.3 Summary

This chapter discussed the development of two kernels that implemented Rijndael. Although a Rijndael kernel could have been developed using bit and byte-level operations, as with the previously described DES designs, a 32-bit implementation method was used to allow more efficient mapping to the CS2112 hardware. The first design processed a single 128-bit plaintext using a 128-bit key using a LUT strategy in which one set of tables completed the first 9 rounds and another the final round. The overall throughput of this design was estimated to be 75 Mbits/sec. The second implementation utilized the pipelining capability of the CS2112 to improve throughput. The design was split across the four logic slices so that each slice performed identical operations, with the data interconnections varying, to produce 32 bits of the ciphertext. To lower LSM usage the double LUT strategy, used in the previous implementation, was changed to use only a single set of LUTs to implement all rounds. The throughput of this pipelined design was estimated to be 1.1 Gbits/sec.

Chapter 7

Bluetooth, KASUMI and RC4

After the implementations of DES and Rijndael were completed, three other encryption algorithms were considered for implementation on the CS2112 fabric. The three algorithms - E0, KASUMI, and RC4 - were chosen because they are part of popular wireless standards. A complete software application, utilizing the hardware kernel, as with DES and Rijndael was not developed for these algorithms. The goal of this work was to evaluate the performance of these kernels in terms of hardware complexity and speed. Therefore, an outline of each kernel's data path logic and control unit was developed from which performance estimates could be made. As well, these design outlines could be used as a basis for future work in this area. The following sections describe the results of this work.

7.1 Bluetooth Encryption Algorithm - E0

After the successful development of two block cipher kernels, an attempt was made to implement a stream cipher on the CS2112. The first algorithm chosen for analysis was the recently developed E0 keystream generator. At first, it appeared as though the fabric would be able to easily accommodate E0. However, after more careful analysis its implementation proved to be much more difficult than first anticipated.

As discussed in Section 2.4 the data path of the keystream generator can be split into two main parts - the LFSRs and the summation combiner. The summation combiner is a simple FSM and can easily be implemented in a LSM lookup table. As well, the 25 and 31 bit LFSRs can each be implemented in a single DPU using the LFSR mode of operation available in the DPUs. The polynomials as given in the standard would have to be modified as the DPUs assume a Galois LFSR structure whereas the standard specifies a Fibonacci LFSR. The only modification required to convert the Fibonacci form to a Galois form is a change to the initial fill vector and a reversal of the tap weights [12]. However, the LFSR mode of operation for DPUs could only accommodate LFSRs which were less than 32 bits wide so a single DPU implementation of the four LFSRs in the keystream generator was not possible.

Since E0 is targeted specifically at bitwise hardware implementation, very few software implementations exist for the algorithm. In fact, only a single software implementation of E0 developed by Saarinen [65] could be found. The initialization portion of Saarinen's implementation was quite complex and did not lend itself to a high speed implementation on the CS2112. It was decided that only the post initialization portion of the algorithm that produces the keystream bits would be implemented in the fabric with the initialization values calculated in the ARC. The keystream bit generation portion of the software implementation is shown in Algorithm 7.1.1.

As a further complication, Saarinen's code uses 64-bit integers to represent the LFSRs. Since the DPUs are 32-bits wide the LFSR clocking operations to generate e0_r1 to e0_r4 need to be decomposed into a number of 32-bit operations. As an example, consider the code for generating the next value of e0_r4 in Algorithm 7.1.1. These operations can be further subdivided into the operations for generating the new Most Significant Bit (MSB) of the LFSR and merging it into the LFSR. The MSB is generated by XORing the bits of the LFSR as selected by the feedback polynomial. In this case, bits 38, 35, 27 and 3 are selected and XORed before the merge. Since

Algorithm 7.1.1 e0_clock() function

```
int e0_clock()
{
  int t;

  e0_r1 = ((e0_r1 << 1) & 0x1fffffe) |
    (((e0_r1 >> 7) ^ (e0_r1 >> 11) ^ (e0_r1 >> 19) ^ (e0_r1 >> 24)) & 1);
  e0_r2 = ((e0_r2 << 1) & 0x7fffffe) |
    (((e0_r2 >> 11) ^ (e0_r2 >> 15) ^ (e0_r2 >> 23) ^ (e0_r2 >> 30)) & 1);
  e0_r3 = ((e0_r3 << 1) & 0x1ffffffe) |
    (((e0_r3 >> 32) ^ (e0_r3 >> 27) ^ (e0_r3 >> 23) ^ (e0_r3 >> 3)) & 1);
  e0_r4 = ((e0_r4 << 1) & 0x7ffffffffe) |
    (((e0_r4 >> 38) ^ (e0_r4 >> 35) ^ (e0_r4 >> 27) ^ (e0_r4 >> 3)) & 1);

  e0_x = ((e0_r1 >> 23) & 1) | ((e0_r2 >> 22) & 2) |
    ((e0_r3 >> 29) & 4) | ((e0_r4 >> 28) & 8);

  e0_state = e0_fsm[e0_state][e0_x];
  t = e0_x ^ (e0_x >> 2);
  t ^= t >> 1;

  return (t ^ (e0_state >> 2)) & 1;
}
```

the DPUs can only operate on 32-bit data, the LFSR must be split across 2 DPUs. Further, DPUs can then be used to select the bits and then XOR them together as shown in Figure 7.1. Once the MSB had been generated it could be merged into the 2

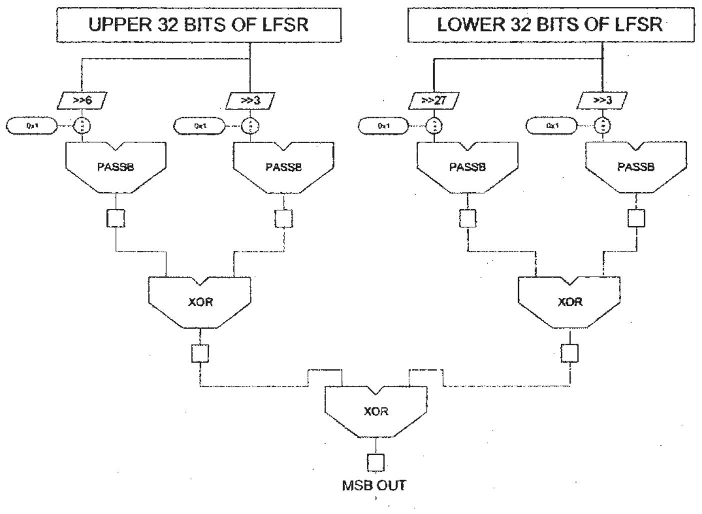

Figure 7.1: MSB Generation Data Path Structure

DPU wide LFSR as shown in Figure 7.2. The other LFSRs could be implemented in a similar fashion or where possible a single DPU could be used to save fabric resources.

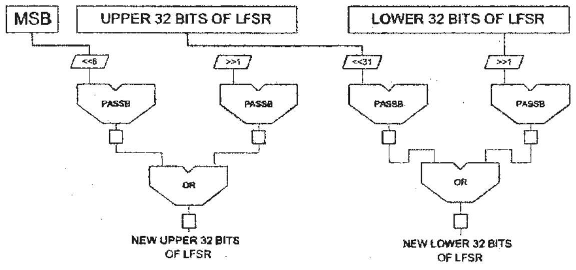

Figure 7.2: Shifting LFSR and Combining with New MSB

The overall performance of the algorithm is now very much limited by the performance of the LFSR hardware. Although a single DPU LFSR could be used in two cases, the multiple DPU LFSRs would limit the speed of operation even with clocking done in parallel since all of the LFSRs must be finished before the algorithm proceeded. Therefore, after considering the other operations required to complete a round, it would take approximately 14 fabric clock cycles to generate a single keystream bit. Hence, the overall performance of such a kernel would be approximately 7.14 Mbits/second at a clock speed of 100MHz.

7.2 KASUMI

As discussed in Section 2.5, KASUMI is an iterated block cipher with a Feistel structure and, as such, is very similar to DES. Many of the design principles used in the development of the previous DES kernels were applied in the preliminary development of a KASUMI design. Since the key generation algorithm was again relatively complex in itself, it was not to be implemented in hardware. Therefore, as in previous designs, it was assumed that the microprocessor would calculate and pass an entire set of round subkeys into the fabric. As in the DES kernels, the S-boxes inside the FI function were to be implemented inside LSMs as lookup tables rather than as boolean logic. Also, as with previous designs, the initial focus was the development of a purely iterative KASUMI design which would process a single 64-bit plaintext.

While performance would not be optimal in the purely iterative case, the data path proved to be much simpler and easier to develop than a pipelined design. As well, the iterative implementations usually prove to be a excellent basis for the development of a pipelined kernel.

After the external interface and software/hardware boundary had been decided, the development of the KASUMI data path began by breaking it into the natural boundaries imposed by KASUMI's subfunctions. The FI subfunction was considered first with later functions building upon this data path hardware. Figure 7.3 shows the fabric resources required to implement FI. As stated in Section 2.5.2, FI contains

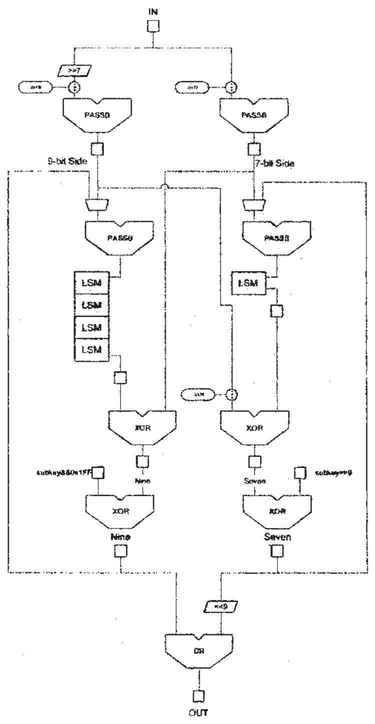

Figure 7.3: FI Subfunction

two S-boxes which are of unequal sizes. The 7-bit input/output S-box could be implemented in a single LSM with a single DPU for accessing the table. The 9-bit S-box could not be stored in a single LSM and was held in 4 chained LSMs with a single DPU for accessing the table. The zero pad and truncate functions, ZE and

TR, required no additional hardware since the 7-bit values were already zero padded out to 32-bits and the 9-bit value could be truncated with a simple mask operation. The 16-bit subkeys KI_j were also stored in an LSM with two DPUs taking this subkey and splitting it into 9-bit and 7-bit components before XORing with their respective data. Once the key addition had been completed, the 7 and 9-bit components were simply recombined to form the 16-bit output.

The FO subfunction contains three iterations of the previously described FI subfunction. Since this is an iterative design, only a single copy of the FI hardware was required in the fabric. In a pipelined implementation multiple copies of this hardware would be necessary. The hardware required for this function was relatively simple in that the 32-bit data was simply split into two 16-bit halves before passing through three iterations of key additions and the subfunction FI. Again, as in the previous function, the 16-bit subkeys were stored in an LSM and retrieved at the appropriate time. Figure 7.4 illustrates the reconfigurable fabric data path configuration. After all iterations were complete, the resultant 16-bit halves were recombined to form the 32-bit output value.

The third and final subfunction of the KASUMI algorithm, FL, did not depend on the previously described functions. As with FO, the FL function takes a 32-bit input and produces a 32-bit output. Figure 7.5 details the data path elements required to implement this function. Since the output of the first instruction as described in Section 2.5.1 was used in the following instruction, the two instructions could not be completed in parallel. Hence, the hardware for only a single pipelined rotate, as described in Section 4.2.2, was necessary. As with previous functions, the subkeys used by FL were held in an LSM and were accessed by a single DPU. The remaining DPUs split the incoming data into its two 16-bit halves, held the new "right" and "left" values and performed the other logical operations required by the function definition.

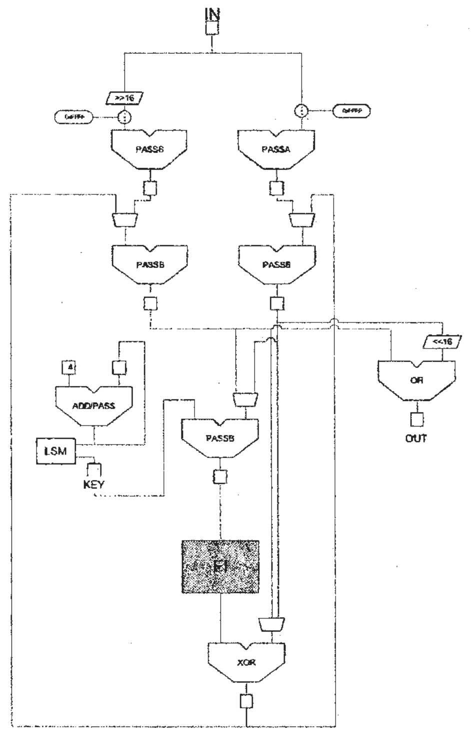

Figure 7.4: KASUMI *FO* Subfunction

Once all subfunctions had been mapped to fabric resources, they were assembled to complete the entire algorithm. Since this implementation was again iterative, only a single copy of the hardware for each subfunction was necessary. If any subfunction was required multiple times, the same hardware was simply reused in each execution. Figure 7.6 shows the complete KASUMI data path. Note that the order of execution of FL and $F0$ depended on the current round and required an extra connection between the two subfunctions on the data path. Also, the single 64-bit input and output data was stored in two DPUs as in previously described designs.

The performance of this KASUMI implementation was estimated by following a procedure very similar to the way it was designed. First, the execution time of each of the subfunction data paths was estimated; then these estimates were combined to find the kernel's overall performance. The subfunctions FI and FL were estimated to take 14 and 11 fabric clock cycles, respectively. FO's total execution time was dependant

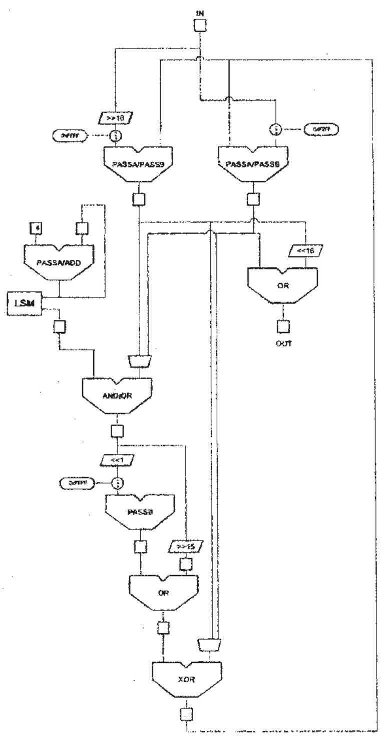

Figure 7.5: KASUMI FL Subfunction

on FI and was estimated at $11 + FI \times 3 = 11 + 14 \times 3 = 53$ clock cycles. Hence, the overall KASUMI kernel would require $4+16+8\times(FL+FO) = 16+8\times(11+53) = 528$ clock cycles to encrypt 64-bits of data. With the CS2112 running at a clock speed of 100 MHz, the kernel would have an approximate throughput of 12.03 Mbits/s. Of course, this estimate ignores the configuration and data transfer overhead that would be present in a finalized application.

In order to improve the above performance figure, the algorithm's loops would have to be unrolled where possible and a pipelined data path developed. For example, the three iterations of the FI subfunction in FO could be done in three repetitions of the FI hardware instead of a single reused one. However, as with previous designs the CS2112 fabric does not have enough resources available to support a fully unrolled and pipelined version of KASUMI. In the best case scenario a single round of the algorithm could be pipelined, allowing a circular pipeline arrangement as described in the DES

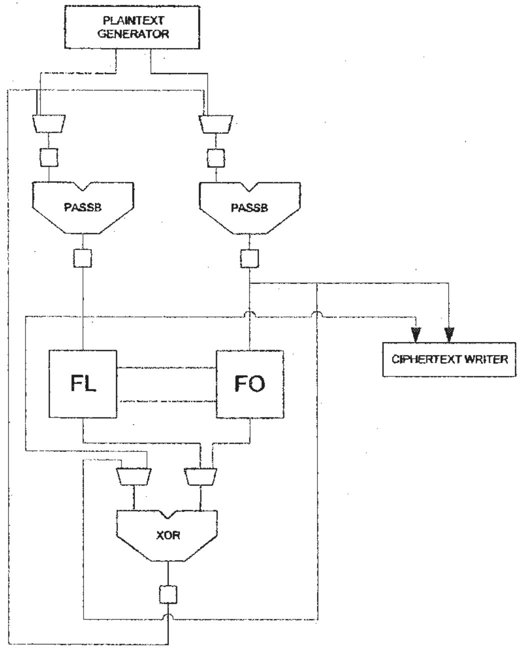

Figure 7.6: Full KASUMI Data Path

and AES kernels, but without further investigation no performance estimates can be made.

7.3 RC4

The RC4 algorithm was a very simple algorithm requiring few computational operations. The beginning of the algorithm involved a setup phase in which the S-box was initialized using the private key. While an important part of the algorithm, this phase was only performed once at the start of encryption. Hence, this phase was to be implemented in software with the initialized S-box being passed into a kernel that produced the output key sequence. RC4 was implemented on the CS2112 in an iterative kernel quite easily; however, there was some difficulty in developing a high speed implementation. As in the previous sections, this design focused primarily on the development of the kernel data path. Only a rough outline of the control unit was developed.

The iterative kernel will produce a single 8-bit output with every iteration by performing the operations in the keystream stage of the algorithm. Since this phase is centered around a single substitution box, the design itself was developed in a similar manner. A high level diagram of the RC4 data path is shown in Figure 7.7. The data path memory requirement for the RC4 algorithm is extremely low and, as

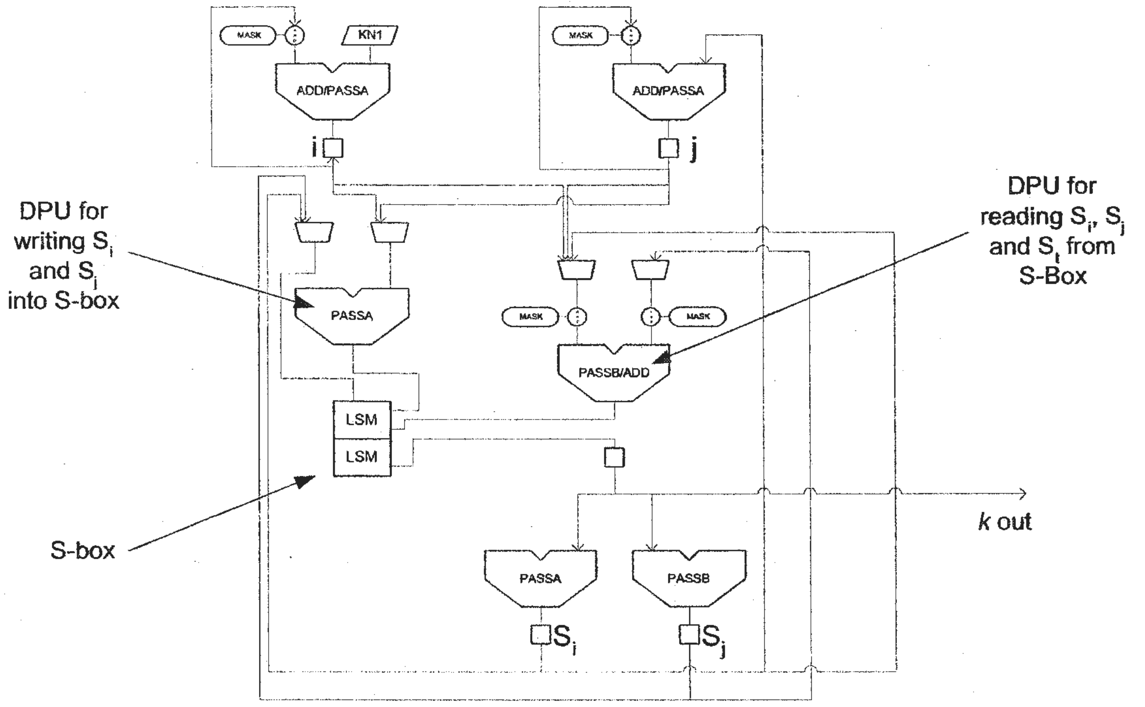
MASK
KN1
ADD/PASSA
i
MASK
ADD/PASSA
j
DPU for writing S_i and S_j into S-box
DPU for reading S_i, S_j and S_t from S-Box
PASSA
MASK
MASK
PASSB/ADD
LSM
LSM
S-box
k out
PASSA
PASSB
S_i
S_j

Figure 7.7: Iterative RC4 Kernel High Level Diagram

can be seen in Figure 7.7, the single 8×8 S-box used in the algorithm can be placed in a pair of chained LSMs with a 32-bit output port or a single LSM with a 8-bit output port. The previous configuration was chosen in this case for reasons discussed below. These LSMs are accessed by a pair of DPUs functioning as reader/writer units. Also, the variables i, j, S_i, and S_j can be held in DPUs. However, some refinement of this approach is possible which both saves space and time.

The variable i must be held in a DPU that either holds the current value or increments i by 1. A mask on the input of this DPU was used above to accomplish the "mod 255" operation as shown in Figure 7.7. This would introduce a single clock cycle period after an increment where the data at the output is not valid because in the clock cycle when the ADD operation takes place, the output of the ALU is not masked. This could be avoided if the reader/writer units consistently mask their inputs to perform the modulo operation before the address is passed to the LSM. Hence a combination of input masking on the reader/writer DPUs along with masking on the i DPU would work correctly. However, since we were using two LSMs in 32-bit output mode, another interesting method to eliminate the above masking presented itself. The LSMs only use bits 2-9 of the provided 32-bit address in 32-bit output port mode (bits 2-10 when two LSMs are chained together) to address a 32-bit position in memory. Hence, the LSMs themselves can be used to mask their input addresses and no input masking was required on either the j DPU or the reader/writer DPUs. This would not have been possible if the LSM was configured in 8-bit mode, since bits 0-9 would have been used to lookup a byte. In that case the above masking would have been required. The output of the j DPU can be treated similarly.

In the final step of the algorithm, a lookup into position $S_i + S_j$ produces the 8-bit output for a round. This addition was initially computed in a separate DPU. However, this extra DPU was not necessary since the reader DPU could perform the addition before passing the result to the LSM. The LSM addressing mechanism described above handles the modulo operation.

The performance limiting factor of this RC4 implementation was its iterative nature. A particular stage of the algorithm cannot proceed until the preceding stage has completed. So, for example, the calculation of j and the subsequent lookup of S_j cannot occur until the new value of i has been calculated. The diagram in Figure 7.8 illustrates these timing constraints with the elapsed clock cycles on the horizontal axis and time increasing from left to right. There are also instances within a round where some parallelization can be achieved to reduce the latency. This parallelization can

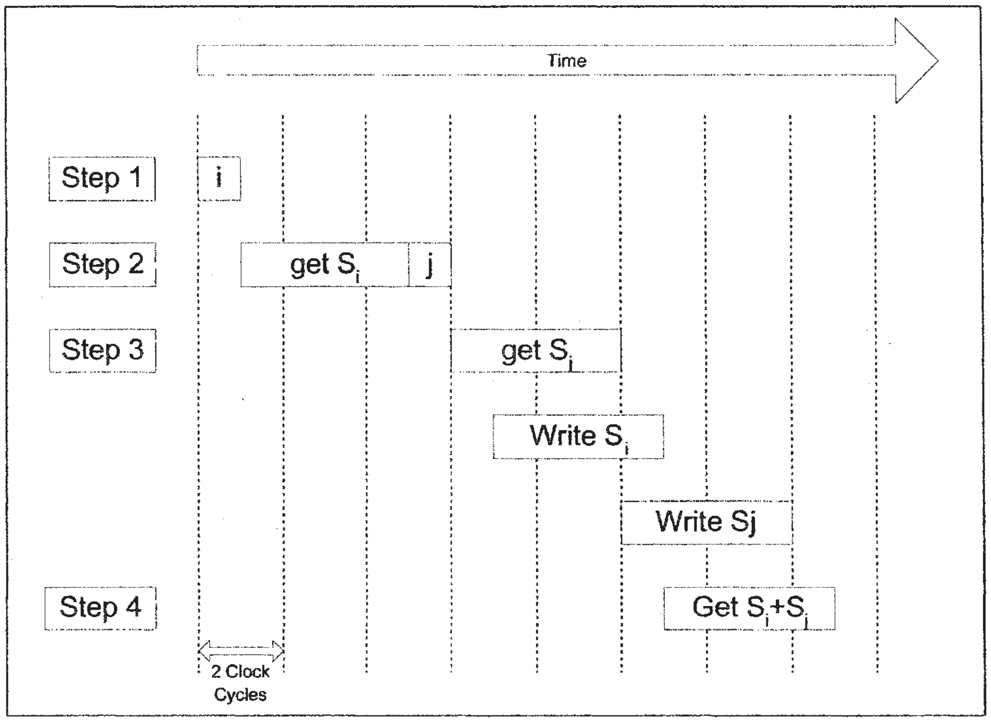

Figure 7.8: RC4 Operation Timing

be seen as overlap between two operations in Figure 7.8. The first round of RC4 takes 15 clock cycles if the fabric setup time is ignored. Another level of parallelization can be achieved between rounds since the computations for the second round can start before the first round has completed. Hence after the first round, data was produced at a rate of 8-bits every 10 clock cycles. Therefore, at 100 MHz the final throughput was approximately 80 Mbits/sec. It is very difficult to improve the performance of the RC4 kernel beyond that of the implementation described above because of the sequential nature of the operations involved. In fact, the greatest limitation to the algorithms performance is the swap of table values. Without this write, the table lookups could be interleaved or multiple copies of the table could be accessed at the same time to achieve a much higher performance figure. However, the swap of table values is a necessary part of the algorithm and cannot be ignored.

7.4 Summary

This chapter presented the high level designs for three cryptographic algorithms. The first algorithm discussed was E0, a stream cipher used to provide security within the Bluetooth protocol. The keystream generation portion of the algorithm could be implemented using a number of DPUs to represent the LFSRs and a simple lookup table, stored in an LSM, to implement the FSM. This E0 design was estimated to have a throughput of 7.14 Mbits/sec. The second algorithm studied was KASUMI, a block cipher used in the 3rd generation GSM standard. A high level design of an iterative kernel was developed in a similar manner to the DES kernels, as both ciphers have a Feistel structure and utilize S-boxes. The developed KASUMI design was estimated to have a throughput of 12.03 Mbits/sec. Finally, the stream cipher system RC4, an algorithm used in many security applications, was analysed. Although RC4 is targeted toward high speed software implementation and mapped easily to the CS2112 fabric, its structure could not fully take advantage of the CS2112's pipelining capability. The high level design indicated that a RC4 kernel would have a throughput of approximately 80 Mbits/sec.

Chapter 8

Conclusions

Through the course of this research a number of cryptographic algorithm implementations were investigated on the Chameleon Systems CS2112 Reconfigurable Communications Processor. Both DES and AES were investigated thoroughly with multiple working kernels developed in each case. Preliminary design work was also completed for the E0, KASUMI and RC4 algorithms and estimates of kernel performance were made using these preliminary designs. The results of this work are summarized below.

In total, three DES kernels were developed with two kernels passing the synthesis and mapping phase. These were tested with both the chip simulator and development board after a final application was completed. Unfortunately, difficulties were encountered which prevented a purely iterative version of the kernel from being mapped to the CS2112 fabric resources. Also, the synthesized pipelined kernel malfunctioned due to either an error in the CS2112 synthesis tool or a bug in the CS2112 chip itself. The iterative kernel, as expected, performed poorly with a throughput of only 27.5 Mbits/sec. The multiple pipelined kernel and pipelined kernel implementations both performed respectably with throughputs of 306 and 322 Mbits/sec, respectively. The difference in throughput resulted from the data output writes that each of the intermediate kernels must perform in the multiple kernel case. This gap should be much wider on the actual chip since in the multiple kernel case the new key values, as well as data, must be loaded into the newly activated kernels before processing proceeds. This would add further delay to the multiple kernel solution and, as a result, its throughput should decrease.

Two AES or Rijndael kernels were developed and tested with the chip simulator and development board. Again the iterative version of the kernel performed quite poorly with a throughput of only 75 Mbits/sec. However, the pipelined kernel produced a throughput of 1.1 Gbits/sec - the highest throughput of any kernel developed during the course of this research. With approximately 96.4% of the available DPUs used in this kernel, it would be difficult to improve the performance beyond this point.

As stated above, preliminary analysis was also completed on the ciphers E0, KASUMI, and RC4. The E0 kernel, with LFSRs having widths greater than 32-bits was very costly to implement on the CS2112 in terms of both hardware and time. As well, the initialization sequence was too complex to implement using the CS2112's available control units. Hence, if initialization was completed in software and the starting values loaded into the kernel the throughput was estimated to be 7.14 Mbits/sec. This is well above the Bluetooth version 1.1 specification which supports a maximum throughput of 720 Kbps. However, the E0 kernel developed only implements a portion of the Bluetooth security architecture. The KASUMI kernel was also quite difficult to develop given the limited hardware resources available. The 12.03 Mbits/sec throughput of the iterative kernel could be greatly improved by unrolling the loops and developing a circular pipeline solution as in the DES case. In fact, its performance could be very close to that of the pipelined DES kernel if more hardware resources were available. RC4 presented a slightly different challenge since it is an algorithm targeted toward purely software implementation. However, this performance figure is still above the 2.4 Mbits/sec connection speed available with 3G devices. The table value swap portion of the RC4 algorithm proved to be a limiting factor in the development of a high speed kernel and the preliminary design's estimated throughput of 80 Mbits/sec is almost maximal. Although the speeds obtained in these preliminary designs were not as high as those obtained with DES and AES they should prove to be a good basis for future implementations on this or similar processors.

In general, the Chameleon CS2112 performed quite well in cryptographic applications, considering it was originally developed for digital signal processing. However, there were a number of factors which limit the speed and complexity of algorithms implemented in the CS2112's fabric. Firstly, many of the developed kernels were limited in size by the amount of available fabric hardware. Resources such as DPUs and LSMs were quickly used up. If the DES, AES and KASUMI algorithms could have been fully unrolled the kernels could have operated on a continuous stream of data and would have produce a full ciphertext per clock cycle after the pipeline had filled. As well, the available control resources were not adequate to develop complex control units. Secondly, the global and local routing matrix was not extensive enough to allow the mapping of complex designs. A large kernel was difficult, if not impossible, to successfully map to the fabric and required careful design and placement practices. Thirdly, the tools provided for automated synthesis and placement of datapath and control resources were very poor. In all cases, manual placement of resources was required and with large designs this process was quite complex. As well, many of the operations performed by the DPU elements are not required in cryptographic applications and only a subset of their functionality was used. Finally, no hardware was provided for performing simple bit level operations such as permutations. Hence, without further modifications to the architecture, the chip is constrained in cryptographic applications.

The original intent of this research was to evaluate the performance of the CS2112 in encryption applications and this work as been completed. However, the following are recommendations made for further research:

1. It is recommended that further algorithms are implemented on the CS2112 and that other modes of data input, such as PIO are investigated. However, since Chameleon Systems Inc. has ceased operations this may not be a possibility.

2. It is recommended that implementations which process data at a packet level be developed. Again, this may not be a possibility.

3. It is recommended that the CS2112 architecture be used as the basis for a new architecture targeted toward cryptographic applications. With modification, the CS2112 fabric's functional units and structure could produce a high performance cryptographic processor.

References

[1] *Internet Domain Survey.* http://www.isc.org/ds: Internet Software Consortium, 2003.

[2] *Telecommunications Industry Association Homepage.* http://www.tiaonline.org: Telecommunications Industry Association, 2003.

[3] *CERT Coordination Center.* http://www.cert.org: CarnegieMellon Software Engineering Institute, 2003.

[4] *CERT/CC Statistics 1998 - 2003.* http://www.cert.org/stats/cert_stats.html: CarnegieMellon Software Engineering Institute, 2003.

[5] *10 Gigabit Ethernet Alliance Homepage.* http://www.10gea.org/index.htm: 10 Gigabit Ethernet Alliance, 2003.

[6] D. A. Buell, *Splash 2: FPGAs in a custom computing machine.* IEEE Computer Society Press, 1996.

[7] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider, "Teramac - configurable custom computing," in *Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines*, pp. 32–38, April 1995.

[8] R. D. Wittig, "OneChip: An FPGA Processor with Reconfigurable Logic," M.A.Sc, University of Toronto, 1995.

[9] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 12–21, IEEE Computer Society Press, 1997.

[10] M. Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. J. Kurdahi, "Design and implementation of the morphosys reconfigurable computing processor," *Journal of VLSI Signal Processing Systems*, vol. 24, no. 2, pp. 147–164, 2000.

[11] A. G. Konheim, *Cryptography: A Primer.* New York: John Wiley & Sons, 1981.

[12] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, 1996.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[14] D. Coppersmith, "The Data Encryption Standard and its Strength Against Attacks," *IBM Journal of Research and Development*, vol. 38, May 1994.

[15] *EFF DES Cracker Project.* `http://www.eff.org/descracker`: Electronic Frontier Foundation, 2001.

[16] J. J. G. Savard, "The Advanced Encryption Standard (Rijndael)," tech. rep., World Wide Web, `http://home.ecn.ab.ca/~jsavard/crypto/co040701.htm`, 2000.

[17] *Specification for the Advanced Encryption Standard (AES).* `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`: National Institute of Standards and Technology, 2001.

[18] *The Official Bluetooth Wireless Info Site.* `http://www.bluetooth.com`: Bluetooth Special Interest Group, 2003.

[19] D. Meyers, ed., *Bluetooth Specification Version 1.1: Volume 1.* http://www.bluetooth.org/foundry/specification/document/Bluetooth_Core_%1.1_vol_1/en/1/Bluetooth_Core_1.1_vol_1.zip: Bluetooth SIG, 2003.

[20] *3rd Generation Partnership Project Homepage.* http://www.3gpp.org: 3rd Generation Partnership Project, 2002.

[21] *3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 1: f8 and f9 Specification.* ftp://ftp.3gpp.org/specs/2002-06/Rel-5/35_series/35201-500.zip: 3rd Generation Partnership Project, 2002.

[22] M. Matsui and T. Tokita, "MISTY, KASUMI, and Camilla Cipher Algorithm Development," *Mitsubishi ADVANCE*, vol. 100, December 2002.

[23] *3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification.* ftp://ftp.3gpp.org/specs/2002-06/Rel-5/35_series/35202-500.zip: 3rd Generation Partnership Project, 2002.

[24] J. P. Huber and M. W. Rosneck, *Successful ASIC Design the First Time Through.* Van Nostrand Reinhold, 1999.

[25] J. Becker, A. Kirschbaum, F. Renner, and M. Glesner, "Perspectives of Reconfigurable Computing in Research, Industry and Education," in *FPL '98* (R. Hartenstein and A. Keevallik, eds.), vol. 1482 of *LNCS*, (Berlin Heidelberg), pp. 39–48, Springer-Verlag, 1998.

[26] A. DeHon, "Role of Reconfigurable Computing," tech. rep., World Wide Web, `http://www.cs.berkeley.edu/~amd/reconfig_com_roundtable_oct96/`, October 1996.

[27] J. S. Rinaldi, "System On Chip: Taking the 'Hard' Out of Hardware," PDF File 87K, World Wide Web, `http://www.rtaautomation.com/documents/socoverview.pdf`, 1999.

[28] R. Tessier and W. Burleson, "Reconfigurable Computing For Digital Signal Processing: A Survey," *Journal of VLSI Signal Processing*, vol. 28, no. 1, 2001.

[29] F. Granville, "Composing Music on the PC: A New Gig for Reconfigurable Computing," *EDN*, vol. 41, December 1996.

[30] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," in *Proceedings of the IEEE Symposium on FPGAs For Custom Computing Machines*, 1996.

[31] J. E. Carrillo, "Evaluation of the OneChip Reconfigurable Processor," M.A.Sc, University of Toronto, 2000.

[32] J. E. Carrillo and P. Chow, "The Effect of Reconfigurable Units in Superscalar Processors," in *International Symposium on Field-Programmable Gate Arrays*, February 2001.

[33] R. Jeschke, "An FPGA-Based Reconfigurable Coprocessor for the IBM PC," m.a.sc, University of Toronto, 1994.

[34] P. Grahm and B. Nelson, "Reconfigurable Processors for High-Performance, Embedded Digital Signal Processing," in *Proceedings of the Ninth International Workshop on Field Programmable Logic and Applications*, August 1999.

[35] *XC3000 Series Field Programmable Product Description.* `http://direct.xilinx.com/bvdocs/publications/3000.pdf`: Xilinx Inc., 1998.

[36] A. Takahara, "More wires and Fewer LUTs: A design methodology for FPGAs," in *ACM/SIGDA International Symposium on FPGAs*, 1998.

[37] M. F. Sakr, "Reconfigurable Processor Employing Optical Channels," in *Proceedings of the 1998 International Topical Meeting on Optics in Computing (OC '98)*, 1998.

[38] W. Luk, "Pipeline Morphing and Virtual Pipelining," in *Field Programmable Logic and Applications* (W. Luk, P. Cheung, and M. Gleesner, eds.), vol. 1304 of *LNCS*, (Berlin), pp. 111–120, Springer-Verlag, 1997.

[39] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *Proceedings of the IEEE*, vol. 48, pp. 615–628, April 1998.

[40] E. Ciner, M. Liberatori, and L. Lopardo, "Data Encryption Standard Implementation," *Latin American Applied Research*, vol. 30, no. 2, pp. 179–184, 2000.

[41] S. Shepherd, "A High Speed Software Implementation of the Data Encryption Standard," *Computers and Security*, vol. 14, no. 4, 1995.

[42] A. Pfitzmann and R. Abmann, "More Efficient Software Implementations of DES," *Computer Security*, vol. 12, no. 5, 1993.

[43] D. C. Wilcox, L. G. Pierson, P. J. Robertson, E. L. Witzke, and K. Gass, "A DES ASIC suitable for Network Encryption at 10 Gbps and Beyond," in *Cryptographic Hardware and Embedded Systems CHES '99* (C. Koc and C. Paar, eds.), vol. 1717 of *LNCS*, (Berlin), pp. 37–48, Springer-Verlag, 1999.

[44] H. Leitold, W. Mayerwieser, U. Payer, K. C. Posch, R. Posch, and J. Wolkerstorfer, "A 155 Mbps triple-DES network encryptor," in *Cryptographic Hardware and Embedded Systems CHES 2000*, vol. 1965 of *LNCS*, 2000.

[45] S. Trimberger, R. Pang, and A. Singh, "A 12 Gbps DES encryptor/Decryptor core in an FPGA," in *Cryptographic Hardware and Embedded Systems CHES 2000*, vol. 1965 of *LNCS*, 2000.

[46] M. McLoone and J. V. McCanny, "Single-Chip FPGA Implementation of the Advanced Encryption Standard Algorithm," in *Field-Programmable Logic and Applications*, vol. 2147 of *LNCS*, 2001.

[47] A. Dandalis, V. K. Prasanna, and J. D. Rolim, "A Comparative Study of Performance of AES Final Candidates Using FPGAs," in *Cryptographic Hardware and Embedded Systems CHES 2000*, vol. 1965 of *LNCS*, 2000.

[48] H. Kuo and I. Verbauwhede, "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm," in *Cryptographic Hardware and Embedded Systems CHES 2001*, vol. 2162 of *LNCS*, 2000.

[49] V. Fischer and M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware," in *Cryptographic Hardware and Embedded Systems CHES 2001*, vol. 2162 of *LNCS*, 2000.

[50] A. Lutz, J. Triechler, F. Gurkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner, "2Gbit/s hardware realizations of RIJNDAEL and SERPENT: A comparative analysis," in *Cryptographic Hardware and Embedded Systems CHES 2002*, vol. 2523 of *LNCS*, 2000.

[51] B. Gladman, "Cryptography Technology," tech. rep., World Wide Web, `http://fp.gladman.plus.com/cryptography_technology/index.htm`, 2003.

[52] A. Satoh and S. Morioka, "Small and High-Speed Hardware Architectures for the 3GPP Standard Cipher KASUMI," in *Information Security*, vol. 2433 of *LNCS*, 2000.

[53] P. Kitsos, N. Sklavos, K. Papadomanolakis, and O. Koufopavlou, "Hardware Implementation of Bluetooth Security," *IEEE Pervasive Computing*, 2003.

[54] M. Roe, "Performance of Block Ciphers and Hash Functions - One Year Later," in *Fast Software Encryption*, pp. 359–362, 1994.

[55] L. E. Frenzel, "Cryptochips Help Eliminate The Security Bottleneck," *Electronic Design*, March 2003.

[56] *Broadcom Corporation Worldwide Homepage.* `http://www.broadcom.com`: Broadcom Corportation, 2003.

[57] *Cavium Networks Homepage.* `http://www.cavium.com`: Cavium Networks, 2003.

[58] *Corrent - Internet Security at Light Speed.* `http://www.corrent.com`: Corrent Inc., 2003.

[59] I. Andoni, P. Chodowiec, and J. Radzikowski, "Hardware Implementation of IPSec Cryptographic Transformations," tech. rep., George Mason University, 2001.

[60] E. Monsanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez, "CryptoBooster: A Reconfigurable Modular Cryptograhpic Coprocessor," in *Cryptographic Hardware and Embedded Systems CHES '99*, vol. 1717 of *LNCS*, 1999.

[61] R. R. Taylor and S. C. Goldstein, "A High-Performance Flexible Architecture for Cryptography," in *Cryptographic Hardware and Embedded Systems CHES '99*, vol. 1717 of *LNCS*, 1999.

[62] M. Stebinsky, "CYPRIS An Application Specific REconfigurable Processor," tech. rep., World Wide Web, http://klabs.org/richcontent/MAPLDCon98/Papers/pab1_stebinsky.pdf, 1998.

[63] Chameleon Systems Inc., *CS2112 Reconfigurable Communications Processor Data Book v1.3*, July 2001.

[64] Chameleon Systems Inc., *CS2112 Reconfigurable Communications Processor Users Manual v1.3*, July 2001.

[65] M. O. Saarinen, "A Software Implementation of the Bluetooth Encryption Algorithm E0," tech. rep., World Wide Web, http://www.jyu.fi/~mjos/e0.c, 2002.

Appendix A

Verilog Examples

A.1 Rotate 11 bits Left Module

```
//
//Title: Rotate Data by 11 Left
//Author: Andrew Cook
//
//Description: - Rotates data on data_in0 input by 11 to the left
//             - rot_ctl selects between two instructions to perform rotate
//             - instruction 0 must be followed by instruction 1 1 clock cycle later
//
`include "CS2112_Instructions.include"

module rotateleftby11(clk, rst, data_in0, rotated_out, rot_ctl);

   input clk, rst;
   input [31:0] data_in0;
   input rot_ctl;
   output [31:0] rotated_out;


   wire [31:0] shifted111;

   //setup the input registers and output registers
   defparam rot_dpu.A_REG_INITIAL_VALUE = 32'h0;
   defparam rot_dpu.B_REG_INITIAL_VALUE = 32'h0;
   defparam rot_dpu.O_REG_INITIAL_VALUE = 32'h0;

   // define the instructions

   // Instruction 0
   defparam rot_dpu.INSTRUCTION_0 = (`OPA_NO_REG | `B0_IN | `OPB_NO_REG |
                                     `LSL | `SHFT_AMT_11 | `ALU_PASSB |
                                     `LOAD_O_REG );

   defparam rot_dpu.INSTRUCTION_1 = (`A0_IN | `OPA_NO_REG | `B0_IN |
                                     `OPB_NO_REG | `LSR | `SHFT_AMT_21 |
                                     `ALU_OR | `LOAD_O_REG );


   CS2112_DPU rot_dpu(
      .rst(rst),
      .clk(clk),
      //A B inputs
      .b_in0(data_in0),
      .a_in0(rotated_out),
      //Dpu Output
      .dpu_output(rotated_out),
      //CSM address
```

```
        .csm_addr({2'b0,rot_ctl}),
        //lsm connections
        .lsm_addr(),
        .data_from_lsm()
    );
endmodule
```

A.2 Pipelined Rotate 11 bits Left Module

```
//
//Title: Pipelined Rotate by 11
//Author: Andrew Cook
//
//Description: - Rotates data on data_in0 input by 11 bits to the left
//             - no control lines necessary and rotated data appears 2 clock cycles after
//               entering rotator hardware
//
`include "CS2112_Instructions.include"
module piperotateleftby11(clk, rst, data_in0, rotated_out);

    input clk, rst;
    input [31:0] data_in0;
    output [31:0] rotated_out;


    wire [31:0] shifted111;
    //------------------------------------------------
    // shift left by 11
    //------------------------------------------------

    defparam shift_left11_dpu.A_REG_INITIAL_VALUE = 32'h0;
    defparam shift_left11_dpu.B_REG_INITIAL_VALUE = 32'h0;
    defparam shift_left11_dpu.O_REG_INITIAL_VALUE = 32'h0;

    // Instruction 0
    defparam shift_left11_dpu.INSTRUCTION_0 = (`OPA_NO_REG | `B0_IN | `OPB_NO_REG |
                                               `LSL | `SHFT_AMT_11 | `ALU_PASSB |
                                               `LOAD_O_REG );

    CS2112_DPU shift_left11_dpu(
        .rst(rst),
        .clk(clk),
        //A B inputs
        .b_in0(data_in0),
        //Dpu Output
        .dpu_output(shifted111),
        //CSM address
        .csm_addr(3'b000),
        //lsm connections
        .lsm_addr(),
        .data_from_lsm()
    );

    //------------------------------------------------
    // shift right by 21
    //------------------------------------------------

    defparam shift_right21_dpu.A_REG_INITIAL_VALUE = 32'h0;
    defparam shift_right21_dpu.B_REG_INITIAL_VALUE = 32'h0;
    defparam shift_right21_dpu.O_REG_INITIAL_VALUE = 32'h0;

    // Instruction 0
    defparam shift_right21_dpu.INSTRUCTION_0 = (`A0_IN | `OPA_NO_REG | `OPB_REG |
                                                `LOAD_B_REG | `LSR | `SHFT_AMT_21 |
                                                `ALU_OR | `LOAD_O_REG );
    CS2112_DPU shift_right21_dpu(
        .rst(rst),
        .clk(clk),
        //A B inputs
        .a_in0(shifted111),
        .b_in0(data_in0),
        //Dpu Output
        .dpu_output(rotated_out),
        //CSM address
        .csm_addr(3'b000),
        //lsm connections
        .lsm_addr(),
        .data_from_lsm()
    );
endmodule
```

Appendix B

Chameleon Preprocessor Example

B.1 Original Code

```
#define N 16
/*********************************************/
/*define the chameleon hardware function here*/
/*********************************************/

#pragma CMLN_FUNC_DEF spne(int in dp.sbox1.lsm[N], int in dp.sbox2.lsm[N],
                          int in dp.sbox3.lsm[N], int in dp.sbox4.lsm[N],
                          int in dp.xor_dpu.dpu.o, int in dp.key_lsm.lsm[8],
                          int out *dp.xor_dpu.dpu.o)

int main(int argc, char *argv[])
{
        /*give a plaintext*/

        unsigned int p;
        unsigned int cref;
        unsigned int ctest;

        int i,j;
        /*generate a plaintext*/
        for (j=0; j<10; j++)
        {
                p = rand();


                /*generate a random key matrix*/
                for(i = 0; i<4; i++) keys[i] = rand();

                /*************************************/
                /*call the chameleon hardware function*/
                /*************************************/
                #pragma CMLN_FUNC_CALL spne() SLICES=(0:1)
                        spne(SP1, SP2, SP3, SP4, p, keys, &ctest);

                spne(SP1, SP2, SP3, SP4, p, keys, &cref);

                if (ctest == cref)
                {
                        asm volatile ("mov r8, 0x10");
                }
                else
                {
                        asm volatile ("mov r8, 0x20");
                }
        }
}
```

B.2 Chameleon Preprocessor Output

```
#define N 16

/*******************************************/
/*define the chameleon hardware function here*/
/*******************************************/
// #pragma CMLN_FUNC_DEF spne(int in dp.sbox1.lsm[N], int in dp.sbox2.lsm[N],
                              int in dp.sbox3.lsm[N], int in dp.sbox4.lsm[N],
                              int in dp.xor_dpu.dpu.o, int in dp.key_lsm.lsm[8],
                              int out *dp.xor_dpu.dpu.o)


int main(int argc, char *argv[])
{
        /*give a plaintext*/

        unsigned int p;
        unsigned int cref;
        unsigned int ctest;

        int i,j;
        /*generate a plaintext*/
        for (j=0; j<10; j++)
        {
                p = rand();


                /*generate a random key matrix*/
                for(i = 0; i<4; i++) keys[i] = rand();


                  /*************************************/
                  /*call the chameleon hardware function*/
                  /*************************************/
                  // #pragma CMLN_FUNC_CALL spne() SLICES=(0:1)
                        /*spne(SP1, SP2, SP3, SP4, p, keys, &ctest);*/

        /* Beginning CMLN EBIOS Primitives */
        {
          spne_cmln_0 = CMLN_ALLOCATE_SLICE(0,1,CMLN_SEQ);
          CMLN_LOAD_CONFIG(S0CFGADR, 1, CMLN_ACTIVE_PLANE, spne_CMLN_CONFIG_BIT_STREAM);
          /* Load arrays to hide load-config latency */
          CMLN_MOVE_VECTOR(S0DCSR0, CMLN_READ, SP1, 4*(N), 0x600);
          CMLN_MOVE_VECTOR(S0DCSR1, CMLN_READ, SP2, 4*(N), 0x800);
          CMLN_MOVE_VECTOR(S0DCSR2, CMLN_READ, SP3, 4*(N), 0x400);
          CMLN_MOVE_VECTOR(S0DCSR3, CMLN_READ, SP4, 4*(N), 0x200);
          CMLN_MOVE_VECTOR(S1DCSR0, CMLN_READ, keys, 32, 0x0);
          CMLN_SET_DMA_ALLOCATED(spne_cmln_0, 0x1f);
          CMLN_WAIT_FOR_CONFIG_LOADED(S0CSR, CMLN_ACTIVE_PLANE, CMLN_POLLING);
          CMLN_MOVE_SCALAR(CMLN_READ, (unsigned long) p, S0T0DPU5O);
          CMLN_WAIT_FOR_DMA(spne_cmln_0, CMLN_ALL_DMAS, CMLN_POLLING);
          CMLN_FIRE_SLICE(spne_cmln_0, S0CSR, 1, CMLN_SEQ);
          CMLN_WAIT_FOR_SLICE(S0CSR, CMLN_POLLING);
          CMLN_MOVE_SCALAR(CMLN_WRITE, (unsigned long) &ctest, S0T0DPU5O);
          CMLN_DEALLOCATE_SLICE(spne_cmln_0);
        }
        /* Ending CMLN Primitives */

                spne(SP1, SP2, SP3, SP4, p, keys, &cref);

                if (ctest == cref)
                {
                        asm volatile ("mov␣r8,␣0x10");
                }
                else
                {
                        asm volatile ("mov␣r8,␣0x20");
                }
        }
}
```

Appendix C

IP Data Path Verilog Module

```
//
// Initial Permutation Datapath for DES
// Author: Andrew Cook
// Created: feb 26, 2002
//
// Description:
//    IP Datapath for DES... Operates on 128 plaintext values in two LSMs..

`include "CS2112_Instructions.include"

module IP_dp(clk, rst, data_gen_ctl, right_data_out, left_data_out);

   input clk;
   input rst;
   input [1:0] data_gen_ctl;


   output [31:0] right_data_out, left_data_out;


   //internal wiring
   wire [31:0] right_in, left_in;
   wire [31:0] dpu1_out, dpu2_out, dpu3_out, dpu4_out;
   wire [31:0] dpu5_out, dpu6_out, dpu7_out, dpu8_out;
   wire [31:0] dpu9_out, dpu10_out, dpu11_out, dpu12_out;
   wire [31:0] dpu13_out, dpu14_out, dpu15_out, dpu16_out;
   wire [31:0] dpu17_out, dpu18_out, dpu19_out, dpu20_out;
   wire [31:0] dpu21_out, dpu22_out, dpu23_out;


   //-----------------------------------------------------------
   // DATA GENERATORS
   //-----------------------------------------------------------

   pp_rd_addr_gen right_data(.clk(clk),
                              .rst(rst),
                              .data_out(right_in),
                              .add_gen_ctl(data_gen_ctl) );

   pp_rd_addr_gen left_data(.clk(clk),
                             .rst(rst),
                             .data_out(left_in),
                             .add_gen_ctl(data_gen_ctl) );



   //-----------------------------------------------------------
```

```
// Inner IP DPU's
//-------------------------------------------------------------------

defparam dpu1.A_REG_INITIAL_VALUE = 32'h0f0f0f0f;
//mask for left input
defparam dpu1.B_REG_INITIAL_VALUE = 32'h0f0f0f0f;
defparam dpu1.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu1.INSTRUCTION_0 = ('A0_IN | 'OPA_AND_MASK | 'B0_IN | 'OPB_AND_MASK |
                                'LSR |'SHFT_AMT_4 | 'ALU_XOR | 'LOAD_O_REG );

CS2112_DPU dpu1(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(right_in),
    .b_in0(left_in),
    //Dpu Output
    .dpu_output(dpu1_out),
    //CSM address
    .csm_addr(3'd0),
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu2.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu2.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu2.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu2.INSTRUCTION_0 = ('A0_IN | 'OPA_REG | 'LOAD_A_REG | 'B0_IN | 'OPB_NO_REG | 'LSL |
                                'SHFT_AMT_4 | 'ALU_XOR | 'LOAD_O_REG );

CS2112_DPU dpu2(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(left_in),
    .b_in0(dpu1_out),
    //Dpu Output
    .dpu_output(dpu2_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu3.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu3.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu3.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu3.INSTRUCTION_0 = ('A0_IN | 'OPA_REG | 'LOAD_A_REG | 'B0_IN | 'OPB_NO_REG |
                                'ALU_XOR | 'LOAD_O_REG );

CS2112_DPU dpu3(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(right_in),
    .b_in0(dpu1_out),
    //Dpu Output
    .dpu_output(dpu3_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu4.A_REG_INITIAL_VALUE = 32'h0000ffff;
//mask for b input
defparam dpu4.B_REG_INITIAL_VALUE = 32'h0000ffff;
defparam dpu4.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu4.INSTRUCTION_0 = ('A0_IN | 'OPA_AND_MASK | 'B0_IN | 'OPB_AND_MASK | 'LSR |
                               'SHFT_AMT_16 | 'ALU_XOR | 'LOAD_O_REG );


CS2112_DPU dpu4(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu3_out),
    .b_in0(dpu2_out),
    //Dpu Output
    .dpu_output(dpu4_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu5.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu5.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu5.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu5.INSTRUCTION_0 = ('A0_IN | 'OPA_REG | 'LOAD_A_REG | 'B0_IN | 'OPB_NO_REG | 'LSL |
                               'SHFT_AMT_16 | 'ALU_XOR | 'LOAD_O_REG );


CS2112_DPU dpu5(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu2_out),
    .b_in0(dpu4_out),
    //Dpu Output
    .dpu_output(dpu5_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu6.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu6.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu6.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu6.INSTRUCTION_0 = ('A0_IN | 'OPA_REG | 'LOAD_A_REG | 'B0_IN | 'OPB_NO_REG |
                               'ALU_XOR | 'LOAD_O_REG );

CS2112_DPU dpu6(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu3_out),
    .b_in0(dpu4_out),
    //Dpu Output
    .dpu_output(dpu6_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu7.A_REG_INITIAL_VALUE = 32'h33333333;
//mask for left input
defparam dpu7.B_REG_INITIAL_VALUE = 32'h33333333;
defparam dpu7.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu7.INSTRUCTION_0 = (`A0_IN | `OPA_AND_MASK | `B0_IN | `OPB_AND_MASK | `LSR |
                               `SHFT_AMT_2 | `ALU_XOR | `LOAD_O_REG );


CS2112_DPU dpu7(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu5_out),
    .b_in0(dpu6_out),
    //Dpu Output
    .dpu_output(dpu7_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu8.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu8.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu8.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu8.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `LOAD_A_REG | `B0_IN | `OPB_NO_REG |
                               `ALU_XOR | `LOAD_O_REG );

CS2112_DPU dpu8(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu5_out),
    .b_in0(dpu7_out),
    //Dpu Output
    .dpu_output(dpu8_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu9.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu9.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu9.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu9.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `LOAD_A_REG | `B0_IN | `OPB_NO_REG | `LSL |
                                 `SHFT_AMT_2 | `ALU_XOR | `LOAD_O_REG );


CS2112_DPU dpu9(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu6_out),
    .b_in0(dpu7_out),
    //Dpu Output
    .dpu_output(dpu9_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu10.A_REG_INITIAL_VALUE = 32'h00ff00ff;
//mask for left input
defparam dpu10.B_REG_INITIAL_VALUE = 32'h00ff00ff;
defparam dpu10.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu10.INSTRUCTION_0 = (`A0_IN | `OPA_AND_MASK | `B0_IN | `OPB_AND_MASK | `LSR |
                                 `SHFT_AMT_8 | `ALU_XOR | `LOAD_O_REG );


CS2112_DPU dpu10(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu8_out),
    .b_in0(dpu9_out),
    //Dpu Output
    .dpu_output(dpu10_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu12.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu12.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu12.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu12.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `LOAD_A_REG | `B0_IN | `OPB_NO_REG | `LSL |
                                 `SHFT_AMT_8 | `ALU_XOR | `LOAD_O_REG );

CS2112_DPU dpu12(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu9_out),
    .b_in0(dpu10_out),
    //Dpu Output
    .dpu_output(dpu12_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu11.A_REG_INITIAL_VALUE = 32'h0;
```

```
//mask for left input
defparam dpu11.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu11.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu11.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `LOAD_A_REG | `B0_IN | `OPB_NO_REG |
                                `ALU_XOR | `LOAD_O_REG );

CS2112_DPU dpu11(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu8_out),
    .b_in0(dpu10_out),
    //Dpu Output
    .dpu_output(dpu11_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);



defparam dpu13.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu13.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu13.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu13.INSTRUCTION_0 = (`B0_IN | `OPB_REG | `LOAD_B_REG |
                                `ALU_PASSB | `LOAD_O_REG );
CS2112_DPU dpu13(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(),
    .b_in0(dpu11_out),
    //Dpu Output
    .dpu_output(dpu13_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);



defparam dpu14.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu14.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu14.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu14.INSTRUCTION_0 = (`B0_IN | `OPB_NO_REG | `LSL | `SHFT_AMT_1 |
                                `ALU_PASSB | `LOAD_O_REG );
CS2112_DPU dpu14(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(),
    .b_in0(dpu12_out),
    //Dpu Output
    .dpu_output(dpu14_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);


defparam dpu15.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu15.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu15.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu15.INSTRUCTION_0 = (`A0_IN | `OPA_NO_REG | `B0_IN | `OPB_REG | `LOAD_B_REG |
                                `LSR | `SHFT_AMT_31 | `ALU_OR | `LOAD_O_REG );

CS2112_DPU dpu15(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu14_out),
    .b_in0(dpu12_out),
    //Dpu Output
    .dpu_output(dpu15_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);


defparam dpu16.A_REG_INITIAL_VALUE = 32'haaaaaaaa;
//mask for left input
defparam dpu16.B_REG_INITIAL_VALUE = 32'haaaaaaaa;
defparam dpu16.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu16.INSTRUCTION_0 = (`A0_IN | `OPA_AND_MASK | `B0_IN | `OPB_AND_MASK |
                                `ALU_XOR | `LOAD_O_REG );

CS2112_DPU dpu16(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu13_out),
    .b_in0(dpu15_out),
    //Dpu Output
    .dpu_output(dpu16_out),
    //CSM address
    .csm_addr(3'd0),
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu17.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu17.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu17.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu17.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `LOAD_A_REG | `B0_IN | `OPB_NO_REG |
                                `ALU_XOR | `LOAD_O_REG );

CS2112_DPU dpu17(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu13_out),
    .b_in0(dpu16_out),
    //Dpu Output
    .dpu_output(dpu17_out),
    //CSM address
    .csm_addr(3'd0),
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu18.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu18.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu18.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu18.INSTRUCTION_0 = (`A0_IN | `OPA_REG | `LOAD_A_REG | `B0_IN | `OPB_NO_REG |
                                `ALU_XOR | `LOAD_O_REG );

CS2112_DPU dpu18(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(dpu15_out),
    .b_in0(dpu16_out),
    //Dpu Output
    .dpu_output(dpu18_out),
    //CSM address
    .csm_addr(3'd0),
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);


defparam dpu19.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu19.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu19.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu19.INSTRUCTION_0 = (`B0_IN | `OPB_REG | `LOAD_B_REG |
                                `ALU_PASSB | `LOAD_O_REG );
CS2112_DPU dpu19(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(),
    .b_in0(dpu17_out),
    //Dpu Output
    .dpu_output(dpu19_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu20.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu20.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu20.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu20.INSTRUCTION_0 = (`B0_IN |   `OPB_REG | `LOAD_B_REG |
                                    `ALU_PASSB | `LOAD_O_REG );
CS2112_DPU dpu20(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(),
    .b_in0(dpu18_out),
    //Dpu Output
    .dpu_output(dpu20_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);


defparam dpu21.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu21.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu21.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu21.INSTRUCTION_0 = (`B0_IN |  `OPB_REG | `LOAD_B_REG |
                                    `ALU_PASSB | `LOAD_O_REG );
CS2112_DPU dpu21(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(),
    .b_in0(dpu20_out),
    //Dpu Output
    .dpu_output(right_data_out),
    //CSM address
    .csm_addr(3'd0),        //lsm connections
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu22.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu22.B_REG_INITIAL_VALUE = 32'h0;
defparam dpu22.O_REG_INITIAL_VALUE = 32'h0;

// Instruction 0
defparam dpu22.INSTRUCTION_0 = (`A0_IN | `OPA_NO_REG | `B0_IN | `OPB_NO_REG |
                                    `LSL | `SHFT_AMT_1 | `ALU_PASSB | `LOAD_O_REG );


CS2112_DPU dpu22(
    .rst(rst),
    .clk(clk),
    //A B inputs
    .a_in0(),
    .b_in0(dpu19_out),
    //Dpu Output
    .dpu_output(dpu22_out),
    //CSM address
    .csm_addr(3'd0),
    //lsm connections
    .lsm_addr(),
    .data_from_lsm()
);

defparam dpu23.A_REG_INITIAL_VALUE = 32'h0;
//mask for left input
defparam dpu23.B_REG_INITIAL_VALUE = 32'h0;
```

```
    defparam dpu23.O_REG_INITIAL_VALUE = 32'h0;

    // Instruction 0
    defparam dpu23.INSTRUCTION_0 = (`A0_IN | `OPA_NO_REG | `B0_IN | `OPB_REG | `LOAD_B_REG |
                                     `LSR | `SHFT_AMT_31 | `ALU_OR | `LOAD_O_REG );

    CS2112_DPU dpu23(
        .rst(rst),
        .clk(clk),
        //A B inputs
        .a_in0(dpu22_out),
        .b_in0(dpu19_out),
        //Dpu Output
        .dpu_output(left_data_out),
        //CSM address
        .csm_addr(3'd0),
        //lsm connections
        .lsm_addr(),
        .data_from_lsm()
    );


endmodule
```

Appendix D

Testbench Examples

D.1 Verilog Testbench

```
%
% Pipelined DES Verilog Testbench
%
% Created by: Andrew Cook
%

module DEStb;

   reg clk, rst;
   reg start;
   wire done;

   % pipelined des module
   DES destotal(.clk(clk),
                .rst(rst),
                .start(start),
                .done(done) );

   initial clk <= 1; always @(clk) clk <= #5 ~clk;

   initial begin
       rst = 0;
       start = 0;
       %input plaintexts
       `include "data2.include"
       %input s-boxes and subkeys
       `include "s_box.include"
       `include "key_lsm.include"
       #10;
       rst = 1'b1;
       #10;
       rst = 1'b0;
       #80;
       %start the fabric function
       start = 1'b1;
       #10;
       start = 1'b0;
       #4000
        $finish;
    end

   %output signals to signalscan file
   initial begin
        $shm_open("DES.shm");
        $shm_probe("AS", destotal);
    end
```

```
endmodule
```

D.2 C Testbench

```
/* DES Testbench Main
 * Created by: Andrew Cook
 *
 * Modifications also made to d3des.c to allow chameleon hardware calls
 */


#include "d3des.h"

int
main (void)
{
        char key[8] = {0x01,0x23,0x45,0x67,0x89,0x0ab,0x0cd,0x0ef};
        int numPlaintexts = 15;
        char plaintext[numPlaintexts*8];
        char ciphertext[numPlaintexts*8];
        char ciphertextsoft[numPlaintexts*8];
        char singlept[8];
        char singlect[8];
        int i,j;
        int okay = 1;

        /*initialize the plaintexts*/
        for (i=0; i < numPlaintexts*8; i++)
        {
                plaintext[i]= (char) i;
        }

        /* call the CS2112 hardware function in d3des.c */
        deskey(key,EN0);
        desblock(plaintext, ciphertext, numPlaintexts);

        /*now do it using software function for a check*/

        for (i=0; i<numPlaintexts; i++)
        {
                for(j=0; j<8; j++)
                {
                        singlept[j]=plaintext[8*i+j];
                }
                /* still a software function called des */
                des(singlept, singlect);
                for(j=0; j<8; j++)
                {
                        ciphertextsoft[i*8+j]=singlect[j];
                }
        }

        for (i=0; i<numPlaintexts*8; i++)
        {
                if (ciphertext[i] != ciphertextsoft[i])
                {
                        okay = 0;
                }
        }

        /* following code is for Development Module Testing */
        if (okay == 1)
        {
                asm volatile ("mov␣r8,␣0x10");
        }
        else
        {
                asm volatile ("mov␣r8,␣0xff");
        }

        return(0);
}

/* Pipelined DES Testbench
 * Original Code Modified by: Andrew Cook
 */


/* D3DES (V5.09) -
 *
 * A portable, public domain, version of the Data Encryption Standard.
 *
 * Written with Symantec's THINK (Lightspeed) C by Richard Outerbridge.
 * Thanks to: Dan Hoey for his excellent Initial and Inverse permutation
 * code;  Jim Gillogly & Phil Karn for the DES key schedule code; Dennis
 * Ferguson, Eric Young and Dana How for comparing notes; and Ray Lau,
 * for humouring me on.
 *
 * Copyright (c) 1988,1989,1990,1991,1992 by Richard Outerbridge.
 * (GEnie : OUTER; CIS : [71755,204]) Graven Imagery, 1992.
 */


#include "d3des.h"
#include <stdio.h>


#pragma CMLN_FUNC_DEF DES(in long IP_datapath.right_data.lsm_0.lsm[16], in long IP_datapath.left_da

static void scrunch(unsigned char *, unsigned long *);
static void unscrun(unsigned long *, unsigned char *);
static void desfunc(unsigned long *, unsigned long *);
static void cookey(unsigned long *);

unsigned long __attribute__((aligned (16))) leftplaintexts[13];
unsigned long __attribute__((aligned (16))) rightplaintexts[13];
unsigned long __attribute__((aligned (16))) evenkeys[16];
unsigned long __attribute__((aligned (16))) oddkeys[16];
unsigned long __attribute__((aligned (16))) leftciphertexts[13];
unsigned long __attribute__((aligned (16))) rightciphertexts[13];
static unsigned long __attribute__((aligned (16))) SP1[64] = {
        0x01010400L, 0x00000000L, 0x00010000L, 0x01010404L,
        0x01010004L, 0x00010404L, 0x00000004L, 0x00010000L,
        0x00000400L, 0x01010400L, 0x01010404L, 0x00000400L,
        0x01000404L, 0x01010004L, 0x01000000L, 0x00000004L,
        0x00000404L, 0x01000400L, 0x01000400L, 0x00010400L,
        0x00010400L, 0x01010000L, 0x01010000L, 0x01000404L,
        0x00010004L, 0x01000004L, 0x01000004L, 0x00010004L,
        0x00000000L, 0x00000404L, 0x00010404L, 0x01000000L,
        0x00010000L, 0x01010404L, 0x00000004L, 0x01010000L,
        0x01010400L, 0x01000000L, 0x01000000L, 0x00000400L,
        0x01010004L, 0x00010000L, 0x00010400L, 0x01000004L,
        0x00000400L, 0x00000004L, 0x01000404L, 0x00010404L,
        0x01010404L, 0x00010004L, 0x01010000L, 0x01000404L,
        0x01000004L, 0x00000404L, 0x00010404L, 0x01010400L,
        0x00000404L, 0x01000400L, 0x01000400L, 0x00000000L,
        0x00010004L, 0x00010400L, 0x00000000L, 0x01010004L };

static unsigned long __attribute__((aligned (16))) SP2[64] = {
        0x80108020L, 0x80008000L, 0x00008000L, 0x00108020L,
        0x00100000L, 0x00000020L, 0x80100020L, 0x80008020L,
        0x80000020L, 0x80108020L, 0x80108000L, 0x80000000L,
        0x80008000L, 0x00100000L, 0x00000020L, 0x80100020L,
        0x00108000L, 0x00100020L, 0x80008020L, 0x00000000L,
        0x80000000L, 0x00008000L, 0x00108020L, 0x80100000L,
        0x00100020L, 0x80000020L, 0x00000000L, 0x00108000L,
        0x00008020L, 0x80108000L, 0x80100000L, 0x00008020L,
        0x00000000L, 0x00108020L, 0x80100020L, 0x00100000L,
        0x80008020L, 0x80100000L, 0x80108000L, 0x00008000L,
        0x80100000L, 0x80008000L, 0x00000020L, 0x80108020L,
        0x00108020L, 0x00000020L, 0x00008000L, 0x80000000L,
        0x00008020L, 0x80108000L, 0x00100000L, 0x80000020L,
        0x00100020L, 0x80008020L, 0x80000020L, 0x00100020L,
        0x00108000L, 0x00000000L, 0x80008000L, 0x00008020L,
        0x80000000L, 0x80100020L, 0x80108020L, 0x00108000L };

static unsigned long __attribute__((aligned (16))) SP3[64] = {
        0x00000208L, 0x08020200L, 0x00000000L, 0x08020008L,
        0x08000200L, 0x00000000L, 0x00020208L, 0x08000200L,
        0x00020008L, 0x08000008L, 0x08000008L, 0x00020000L,
        0x08020208L, 0x00020008L, 0x08020000L, 0x00000208L,
        0x08000000L, 0x00000008L, 0x08020200L, 0x00000200L,
        0x00020200L, 0x08020000L, 0x08020008L, 0x00020208L,
        0x08000208L, 0x00020200L, 0x00020000L, 0x08000208L,
        0x00000008L, 0x08020208L, 0x00000200L, 0x08000000L,
        0x08020200L, 0x08000000L, 0x00020008L, 0x00000208L,
        0x00020000L, 0x08020200L, 0x08000200L, 0x00000000L,
        0x00000200L, 0x00020008L, 0x08020208L, 0x08000200L,
        0x08000008L, 0x00000200L, 0x00000000L, 0x08020008L,
        0x08000208L, 0x00020000L, 0x08000000L, 0x08020208L,
        0x00000008L, 0x00020208L, 0x00020200L, 0x08000008L,
        0x08020000L, 0x08000208L, 0x00000208L, 0x08020000L,
        0x00020208L, 0x00000008L, 0x08020008L, 0x00020200L };

static unsigned long __attribute__((aligned (16))) SP4[64] = {
        0x00802001L, 0x00002081L, 0x00002081L, 0x00000080L,
        0x00802080L, 0x00800081L, 0x00800001L, 0x00002001L,
        0x00000000L, 0x00802000L, 0x00802000L, 0x00802081L,
        0x00000081L, 0x00000000L, 0x00800080L, 0x00800001L,
        0x00000001L, 0x00002000L, 0x00800000L, 0x00802001L,
        0x00000080L, 0x00800000L, 0x00002001L, 0x00002080L,
        0x00800081L, 0x00000001L, 0x00002080L, 0x00800080L,
        0x00002000L, 0x00802080L, 0x00802081L, 0x00000081L,
        0x00800080L, 0x00800001L, 0x00802000L, 0x00802081L,
        0x00000081L, 0x00000000L, 0x00000000L, 0x00802000L,
        0x00002080L, 0x00800080L, 0x00800081L, 0x00000001L,
        0x00802001L, 0x00002081L, 0x00002081L, 0x00000080L,
        0x00802081L, 0x00000081L, 0x00000001L, 0x00002000L,
        0x00800001L, 0x00002001L, 0x00802080L, 0x00800081L,
        0x00002001L, 0x00002080L, 0x00800000L, 0x00802001L,
        0x00000080L, 0x00800000L, 0x00002000L, 0x00802080L };

static unsigned long __attribute__((aligned (16))) SP5[64] = {
        0x00000100L, 0x02080100L, 0x02080000L, 0x42000100L,
        0x00080000L, 0x00000100L, 0x40000000L, 0x02080000L,
        0x40080100L, 0x00080000L, 0x02000100L, 0x40080100L,
        0x42000100L, 0x42080000L, 0x00080100L, 0x40000000L,
        0x02000000L, 0x40080000L, 0x40080000L, 0x00000000L,
        0x40000100L, 0x42080100L, 0x42080100L, 0x02000100L,
        0x42080000L, 0x40000100L, 0x00000000L, 0x42000000L,
        0x02080100L, 0x02000000L, 0x42000000L, 0x00080100L,
        0x00080000L, 0x42000100L, 0x00000100L, 0x02000000L,
        0x40000000L, 0x02080000L, 0x42000100L, 0x40080100L,
        0x02000100L, 0x40000000L, 0x42080000L, 0x02080100L,
        0x40080100L, 0x00000100L, 0x02000000L, 0x42080000L,
        0x42080100L, 0x00080100L, 0x42000000L, 0x42080100L,
        0x02080000L, 0x00000000L, 0x40080000L, 0x42000000L,
        0x00080100L, 0x02000100L, 0x40000100L, 0x00080000L,
        0x00000000L, 0x40080000L, 0x02080100L, 0x40000100L };
```

```
static unsigned long __attribute__((aligned (16))) SP6[64] = {
        0x20000010L, 0x20400000L, 0x00004000L, 0x20404010L,
        0x20400000L, 0x00000010L, 0x20404010L, 0x00400000L,
        0x20004000L, 0x00404010L, 0x00400000L, 0x20000010L,
        0x00400010L, 0x20004000L, 0x20000000L, 0x00004010L,
        0x00000000L, 0x00400010L, 0x20004010L, 0x00004000L,
        0x00404000L, 0x20004010L, 0x00000010L, 0x20400010L,
        0x20400010L, 0x00000000L, 0x00404010L, 0x20404000L,
        0x00004010L, 0x00404000L, 0x20404000L, 0x20000000L,
        0x20004000L, 0x00000010L, 0x20400010L, 0x00404000L,
        0x20404010L, 0x00400000L, 0x00004010L, 0x20000010L,
        0x00400000L, 0x20004000L, 0x20000000L, 0x00004010L,
        0x20000010L, 0x20404010L, 0x00404000L, 0x20400000L,
        0x00404010L, 0x20404000L, 0x00000000L, 0x20400010L,
        0x00000010L, 0x00004000L, 0x20400000L, 0x00404010L,
        0x00004000L, 0x00400010L, 0x20004010L, 0x00000000L,
        0x20404000L, 0x20000000L, 0x00400010L, 0x20004010L };

static unsigned long __attribute__((aligned (16))) SP7[64] = {
        0x00200000L, 0x04200002L, 0x04000802L, 0x00000000L,
        0x00000800L, 0x04000802L, 0x00200802L, 0x04200800L,
        0x04200802L, 0x00200000L, 0x00000000L, 0x04000002L,
        0x00000002L, 0x04000000L, 0x04200002L, 0x00000802L,
        0x04000800L, 0x00200802L, 0x00200002L, 0x04000800L,
        0x04000002L, 0x04200000L, 0x04200800L, 0x00200002L,
        0x04200000L, 0x00000800L, 0x00000802L, 0x04200802L,
        0x00200800L, 0x00000002L, 0x04000000L, 0x00200800L,
        0x04000000L, 0x00200800L, 0x00200000L, 0x04000802L,
        0x04000802L, 0x04200002L, 0x04200002L, 0x00000002L,
        0x00200002L, 0x04000000L, 0x04000800L, 0x00200000L,
        0x04200800L, 0x00000802L, 0x00200802L, 0x04200800L,
        0x00000802L, 0x04000002L, 0x04200802L, 0x04200000L,
        0x00200800L, 0x00000000L, 0x00000002L, 0x04200802L,
        0x00000000L, 0x00200802L, 0x04200000L, 0x00000800L,
        0x04000002L, 0x04000800L, 0x00000800L, 0x00200002L };

static unsigned long __attribute__((aligned (16))) SP8[64] = {
        0x10001040L, 0x00001000L, 0x00040000L, 0x10041040L,
        0x10000000L, 0x10001040L, 0x00000040L, 0x10000000L,
        0x00040040L, 0x10040000L, 0x10041040L, 0x00041000L,
        0x10041000L, 0x00041040L, 0x00001000L, 0x00000040L,
        0x10040000L, 0x10000040L, 0x10001000L, 0x00001040L,
        0x00041000L, 0x00040040L, 0x10040040L, 0x10041000L,
        0x00001040L, 0x00000000L, 0x00000000L, 0x10040040L,
        0x10000040L, 0x10001000L, 0x00041040L, 0x00040000L,
        0x00041040L, 0x00040000L, 0x10041000L, 0x00001000L,
        0x00000040L, 0x10040040L, 0x00001000L, 0x00041040L,
        0x10001000L, 0x00000040L, 0x10000040L, 0x10040000L,
        0x10040040L, 0x10000000L, 0x00040000L, 0x10001040L,
        0x00000000L, 0x10041040L, 0x00040040L, 0x10000040L,
        0x10040000L, 0x10001000L, 0x10001040L, 0x00000000L,
        0x10041040L, 0x00041000L, 0x00041000L, 0x00001040L,
        0x00001040L, 0x00040040L, 0x10000000L, 0x10041000L };

static unsigned long KnL[32] = { 0L };
static unsigned long KnR[32] = { 0L };
static unsigned long Kn3[32] = { 0L };
static unsigned char Df_Key[24] = {
        0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
        0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10,
        0x89,0xab,0xcd,0xef,0x01,0x23,0x45,0x67 };

static unsigned short bytebit[8]        = {
        0200, 0100, 040, 020, 010, 04, 02, 01 };

static unsigned long bigbyte[24] = {
        0x800000L,      0x400000L,      0x200000L,      0x100000L,
        0x80000L ,          0x40000L ,          0x20000L ,          0x10000L ,
        0x8000L ,           0x4000L ,           0x2000L ,           0x1000L ,
        0x800L ,            0x400L ,            0x200L ,            0x100L ,
        0x80L ,             0x40L ,             0x20L ,             0x10L ,
        0x8L ,              0x4L ,              0x2L ,              0x1L     };

/* Use the key schedule specified in the Standard (ANSI X3.92-1981). */

static unsigned char pc1[56] = {
        56, 48, 40, 32, 24, 16,  8,      0, 57, 49, 41, 33, 25, 17,
         9,  1, 58, 50, 42, 34, 26,     18, 10,  2, 59, 51, 43, 35,
        62, 54, 46, 38, 30, 22, 14,      6, 61, 53, 45, 37, 29, 21,
        13,  5, 60, 52, 44, 36, 28,     20, 12,  4, 27, 19, 11,  3 };

static unsigned char totrot[16] = {
        1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28 };

static unsigned char pc2[48] = {
        13, 16, 10, 23,  0,  4,  2, 27, 14,  5, 20,  9,
        22, 18, 11,  3, 25,  7, 15,  6, 26, 19, 12,  1,
        40, 51, 30, 36, 46, 54, 29, 39, 50, 44, 32, 47,
        43, 48, 38, 55, 33, 52, 45, 41, 49, 35, 28, 31 };

void deskey(key, edf)   /* Thanks to James Gillogly & Phil Karn! */
unsigned char *key;
short edf;
{
        register int i, j, l, m, n;
        unsigned char pc1m[56], pcr[56];
        unsigned long kn[32];

        for ( j = 0; j < 56; j++ ) {
                l = pc1[j];
                m = l & 07;
                pc1m[j] = (key[l >> 3] & bytebit[m]) ? 1 : 0;
                }
        for( i = 0; i < 16; i++ ) {
                if( edf == DE1 ) m = (15 - i) << 1;
                else m = i << 1;
                n = m + 1;
                kn[m] = kn[n] = 0L;
                for( j = 0; j < 28; j++ ) {
                        l = j + totrot[i];
                        if( l < 28 ) pcr[j] = pc1m[l];
                        else pcr[j] = pc1m[l - 28];
                        }
                for( j = 28; j < 56; j++ ) {
                    l = j + totrot[i];
                    if( l < 56 ) pcr[j] = pc1m[l];
                    else pcr[j] = pc1m[l - 28];
                    }
                for( j = 0; j < 24; j++ ) {
                        if( pcr[pc2[j]] ) kn[m] |= bigbyte[j];
                        if( pcr[pc2[j+24]] ) kn[n] |= bigbyte[j];
                        }
                }
        cookey(kn);
        return;
        }

static void cookey(raw1)
register unsigned long *raw1;
{
        register unsigned long *cook, *raw0;
        unsigned long dough[32];
        register int i;

        cook = dough;
        for( i = 0; i < 16; i++, raw1++ ) {
                raw0 = raw1++;
                *cook     = (*raw0 & 0x00fc0000L) << 6;
                *cook    |= (*raw0 & 0x00000fc0L) << 10;
                *cook    |= (*raw1 & 0x00fc0000L) >> 10;
                *cook++  |= (*raw1 & 0x00000fc0L) >> 6;
                *cook     = (*raw0 & 0x0003f000L) << 12;
                *cook    |= (*raw0 & 0x0000003fL) << 16;
                *cook    |= (*raw1 & 0x0003f000L) >> 4;
                *cook++  |= (*raw1 & 0x0000003fL);
                }
        usekey(dough);
        return;
        }

void cpkey(into)
register unsigned long *into;
{
        register unsigned long *from, *endp;

        from = KnL, endp = &KnL[32];
        while( from < endp ) *into++ = *from++;
        return;
        }

void usekey(from)
register unsigned long *from;
{
        register unsigned long *to, *endp;

        to = KnL, endp = &KnL[32];
        while( to < endp ) *to++ = *from++;
        return;
        }

/*ADDED BY Andrew Cook*/
void desblock(unsigned char *inblock, unsigned char *outblock, int numPlaintexts)
{

        unsigned long work[2];

        int i, j;



        for (j=numPlaintexts; j >= 13 ; j=j -13)
        {

                /*prepare the plaintexts*/
                for(i=0;i<13;i++)
                {
                        scrunch(inblock,work);
                        leftplaintexts[i]=work[0];
                        rightplaintexts[i]=work[1];
                        inblock+=8;
                }

                /*prepare the keys*/
                for(i=0;i<16;i++)
                {
                        oddkeys[i]=KnL[2*i];
                        evenkeys[i]=KnL[2*i+1];
                }

                /*now call fabric function*/
        #pragma CMLN_FUNC_CALL DES() SLICES=(0:4)
                        DES(rightplaintexts, leftplaintexts, evenkeys, oddkeys, SP7, SP5, SP3, SP1,
                /*now put ciphertexts back into a char array*/
                for(i=0; i<13; i++)
                {
                        work[0]=leftciphertexts[i];
                        work[1]=rightciphertexts[i];
                        unscrun(work,outblock);
                        outblock+=8;
                }
        }

        if (j > 0)
        {
                /*there are still < 13 left so prepare the plaintexts*/
                for(i=0;i<j;i++)
                {
                        scrunch(inblock,work);
                        leftplaintexts[i]=work[0];
                        rightplaintexts[i]=work[1];
                        inblock+=8;
                }

                /*prepare the keys*/
                for(i=0;i<16;i++)
                {
                        oddkeys[i]=KnL[2*i];
                        evenkeys[i]=KnL[2*i+1];
                }

                /*now call fabric function*/
        #pragma CMLN_FUNC_CALL DES() SLICES=(0:4)
                        DES(rightplaintexts, leftplaintexts, evenkeys, oddkeys, SP7, SP5, SP3, SP1,

                /*now put ciphertexts back into a char array*/
                for(i=0; i<j; i++)
                {
                        work[0]=leftciphertexts[i];
                        work[1]=rightciphertexts[i];
                        unscrun(work,outblock);
                        outblock+=8;
                }
        }
}


void des(inblock, outblock)
unsigned char *inblock, *outblock;
{
        unsigned long work[2];

        scrunch(inblock, work);
        desfunc(work, KnL);
        unscrun(work, outblock);
        return;
        }

static void scrunch(outof, into)
register unsigned char *outof;
register unsigned long *into;
{
        *into     = (*outof++ & 0xffL) << 24;
        *into    |= (*outof++ & 0xffL) << 16;
        *into    |= (*outof++ & 0xffL) << 8;
        *into++  |= (*outof++ & 0xffL);
        *into     = (*outof++ & 0xffL) << 24;
        *into    |= (*outof++ & 0xffL) << 16;
        *into    |= (*outof++ & 0xffL) << 8;
        *into    |= (*outof   & 0xffL);
        return;
        }

static void unscrun(outof, into)
register unsigned long *outof;
register unsigned char *into;
{
        *into++ = (*outof >> 24) & 0xffL;
        *into++ = (*outof >> 16) & 0xffL;
        *into++ = (*outof >>  8) & 0xffL;
        *into++ =  *outof++      & 0xffL;
        *into++ = (*outof >> 24) & 0xffL;
        *into++ = (*outof >> 16) & 0xffL;
        *into++ = (*outof >>  8) & 0xffL;
        *into   =  *outof        & 0xffL;
        return;
        }


static void desfunc(block, keys)
register unsigned long *block, *keys;
{
        register unsigned long fval, work, right, leftt;
        register int round;

        leftt = block[0];
        right = block[1];
        work = ((leftt >> 4) ^ right) & 0x0f0f0f0fL;
        right ^= work;
        leftt ^= (work << 4);
        work = ((leftt >> 16) ^ right) & 0x0000ffffL;
        right ^= work;
        leftt ^= (work << 16);
        work = ((right >> 2) ^ leftt) & 0x33333333L;
        leftt ^= work;
        right ^= (work << 2);
        work = ((right >> 8) ^ leftt) & 0x00ff00ffL;
        leftt ^= work;
        right ^= (work << 8);
        right = ((right << 1) | ((right >> 31) & 1L)) & 0xffffffffL;
        work = (leftt ^ right) & 0xaaaaaaaaL;
        leftt ^= work;
        right ^= work;
        leftt = ((leftt << 1) | ((leftt >> 31) & 1L)) & 0xffffffffL;


        for( round = 0; round < 8; round++ ) {
                work  = (right << 28) | (right >> 4);
                work ^= *keys++;
                fval  = SP7[ work              & 0x3fL];
                fval |= SP5[(work >>  8) & 0x3fL];
                fval |= SP3[(work >> 16) & 0x3fL];
                fval |= SP1[(work >> 24) & 0x3fL];
                work  = right ^ *keys++;
                fval |= SP8[ work              & 0x3fL];
                fval |= SP6[(work >>  8) & 0x3fL];
                fval |= SP4[(work >> 16) & 0x3fL];
                fval |= SP2[(work >> 24) & 0x3fL];
                leftt ^= fval;
                work  = (leftt << 28) | (leftt >> 4);
                work ^= *keys++;
                fval  = SP7[ work              & 0x3fL];
                fval |= SP5[(work >>  8) & 0x3fL];
                fval |= SP3[(work >> 16) & 0x3fL];
                fval |= SP1[(work >> 24) & 0x3fL];
                work  = leftt ^ *keys++;
                fval |= SP8[ work              & 0x3fL];
                fval |= SP6[(work >>  8) & 0x3fL];
                fval |= SP4[(work >> 16) & 0x3fL];
                fval |= SP2[(work >> 24) & 0x3fL];
                right ^= fval;
        }

        right = (right << 31) | (right >> 1);
        work = (leftt ^ right) & 0xaaaaaaaaL;
        leftt ^= work;
        right ^= work;
        leftt = (leftt << 31) | (leftt >> 1);
        work = ((leftt >> 8) ^ right) & 0x00ff00ffL;
        right ^= work;
        leftt ^= (work << 8);
        work = ((leftt >> 2) ^ right) & 0x33333333L;
        right ^= work;
        leftt ^= (work << 2);
        work = ((right >> 16) ^ leftt) & 0x0000ffffL;
        leftt ^= work;
        right ^= (work << 16);
        work = ((right >> 4) ^ leftt) & 0x0f0f0f0fL;
        leftt ^= work;
        right ^= (work << 4);
        *block++ = right;
        *block = leftt;
        return;
}

#ifdef D2_DES

void des2key(hexkey, mode)                      /* stomps on Kn3 too */
unsigned char *hexkey;                          /* unsigned char[16] */
short mode;
{
        short revmod;

        revmod = (mode == EN0) ? DE1 : EN0;
        deskey(&hexkey[8], revmod);
        cpkey(KnR);
        deskey(hexkey, mode);
        cpkey(Kn3);                                             /* Kn3 = KnL */
        return;
        }

void Ddes(from, into)
unsigned char *from, *into;                     /* unsigned char[8] */
{
        unsigned long work[2];

        scrunch(from, work);
        desfunc(work, KnL);
        desfunc(work, KnR);
        desfunc(work, Kn3);
        unscrun(work, into);
        return;
        }

void D2des(from, into)
unsigned char *from;                            /* unsigned char[16] */
unsigned char *into;                            /* unsigned char[16] */
{
        unsigned long *right, *l1, swap;
        unsigned long leftt[2], bufR[2];

        right = bufR;
        l1 = &leftt[1];
        scrunch(from, leftt);
        scrunch(&from[8], right);
        desfunc(leftt, KnL);
```

```
	desfunc(right, KnL);
	swap = *l1;
	*l1 = *right;
	*right = swap;
	desfunc(leftt, KnR);
	desfunc(right, KnR);
	swap = *l1;
	*l1 = *right;
	*right = swap;
	desfunc(leftt, Kn3);
	desfunc(right, Kn3);
	unscrun(leftt, into);
	unscrun(right, &into[8]);
	return;
	}

void makekey(aptr, kptr)
register char *aptr;				/* NULL-terminated   */
register unsigned char *kptr;		/* unsigned char[8] */
{
	register unsigned char *store;
	register int first, i;
	unsigned long savek[96];

	cpDkey(savek);
	des2key(Df_Key, EN0);
	for( i = 0; i < 8; i++ ) kptr[i] = Df_Key[i];
	first = 1;
	while( (*aptr != '\0') || first ) {
		store = kptr;
		for( i = 0; i < 8 && (*aptr != '\0'); i++ ) {
			*store++ ^= *aptr & 0x7f;
			*aptr++ = '\0';
			}
		Ddes(kptr, kptr);
		first = 0;
		}
	useDkey(savek);
	return;
	}

void make2key(aptr, kptr)
register char *aptr;				/* NULL-terminated    */
register unsigned char *kptr;		/* unsigned char[16] */
{
	register unsigned char *store;
	register int first, i;
	unsigned long savek[96];

	cpDkey(savek);
	des2key(Df_Key, EN0);
	for( i = 0; i < 16; i++ ) kptr[i] = Df_Key[i];
	first = 1;
	while( (*aptr != '\0') || first ) {
		store = kptr;
		for( i = 0; i < 16 && (*aptr != '\0'); i++ ) {
			*store++ ^= *aptr & 0x7f;
			*aptr++ = '\0';
			}
		D2des(kptr, kptr);
		first = 0;
		}
	useDkey(savek);
	return;
	}

#ifndef D3_DES	/* D2_DES only */
#ifdef	D2_DES	/* iff D2_DES! */
```

```
void cp2key(into)
register unsigned long *into;     /* unsigned long[64] */
{
        register unsigned long *from, *endp;

        cpkey(into);
        into = &into[32];
        from = KnR, endp = &KnR[32];
        while( from < endp ) *into++ = *from++;
        return;
        }

void use2key(from)                                    /* stomps on Kn3 too */
register unsigned long *from;     /* unsigned long[64] */
{
        register unsigned long *to, *endp;

        usekey(from);
        from = &from[32];
        to = KnR, endp = &KnR[32];
        while( to < endp ) *to++ = *from++;
        cpkey(Kn3);                                   /* Kn3 = KnL */
        return;
        }

#endif  /* iff D2_DES */
#else   /* D3_DES too */

static void D3des(unsigned char *, unsigned char *);

void des3key(hexkey, mode)
unsigned char *hexkey;                        /* unsigned char[24] */
short mode;
{
        unsigned char *first, *third;
        short revmod;

        if( mode == EN0 ) {
                revmod = DE1;
                first = hexkey;
                third = &hexkey[16];
                }
        else {
                revmod = EN0;
                first = &hexkey[16];
                third = hexkey;
                }
        deskey(&hexkey[8], revmod);
        cpkey(KnR);
        deskey(third, mode);
        cpkey(Kn3);
        deskey(first, mode);
        return;
        }

void cp3key(into)
register unsigned long *into;     /* unsigned long[96] */
{
        register unsigned long *from, *endp;

        cpkey(into);
        into = &into[32];
        from = KnR, endp = &KnR[32];
        while( from < endp ) *into++ = *from++;
        from = Kn3, endp = &Kn3[32];
        while( from < endp ) *into++ = *from++;
        return;
        }

void use3key(from)
register unsigned long *from;    /* unsigned long[96] */
{
        register unsigned long *to, *endp;

        usekey(from);
        from = &from[32];
        to = KnR, endp = &KnR[32];
        while( to < endp ) *to++ = *from++;
        to = Kn3, endp = &Kn3[32];
        while( to < endp ) *to++ = *from++;
        return;
        }

static void D3des(from, into)    /* amateur theatrics */
unsigned char *from;                    /* unsigned char[24] */
unsigned char *into;                    /* unsigned char[24] */
{
        unsigned long swap, leftt[2], middl[2], right[2];

        scrunch(from, leftt);
        scrunch(&from[8], middl);
        scrunch(&from[16], right);
        desfunc(leftt, KnL);
        desfunc(middl, KnL);
        desfunc(right, KnL);
        swap = leftt[1];
        leftt[1] = middl[0];
        middl[0] = swap;
        swap = middl[1];
        middl[1] = right[0];
        right[0] = swap;
        desfunc(leftt, KnR);
        desfunc(middl, KnR);
        desfunc(right, KnR);
        swap = leftt[1];
        leftt[1] = middl[0];
        middl[0] = swap;
        swap = middl[1];
        middl[1] = right[0];
        right[0] = swap;
        desfunc(leftt, Kn3);
        desfunc(middl, Kn3);
        desfunc(right, Kn3);
        unscrun(leftt, into);
        unscrun(middl, &into[8]);
        unscrun(right, &into[16]);
        return;
        }

void make3key(aptr, kptr)
register char *aptr;                         /* NULL-terminated    */
register unsigned char *kptr;           /* unsigned char[24] */
{
        register unsigned char *store;
        register int first, i;
        unsigned long savek[96];

        cp3key(savek);
        des3key(Df_Key, EN0);
        for( i = 0; i < 24; i++ ) kptr[i] = Df_Key[i];
        first = 1;
        while( (*aptr != '\0') || first ) {
                store = kptr;
                for( i = 0; i < 24 && (*aptr != '\0'); i++ ) {
                        *store++ ^= *aptr & 0x7f;
                        *aptr++ = '\0';
                        }
                D3des(kptr, kptr);
                first = 0;
                }
        use3key(savek);
        return;
        }

#endif  /* D3_DES */
#endif  /* D2_DES */

/* Validation sets:
 *
 * Single-length key, single-length plaintext -
 * Key    : 0123 4567 89ab cdef
 * Plain  : 0123 4567 89ab cde7
 * Cipher : c957 4425 6a5e d31d
 *
 * Double-length key, single-length plaintext -
 * Key    : 0123 4567 89ab cdef fedc ba98 7654 3210
 * Plain  : 0123 4567 89ab cde7
 * Cipher : 7f1d 0a77 826b 8aff
 *
 * Double-length key, double-length plaintext -
 * Key    : 0123 4567 89ab cdef fedc ba98 7654 3210
 * Plain  : 0123 4567 89ab cdef 0123 4567 89ab cdff
 * Cipher : 27a0 8440 406a df60 278f 47cf 42d6 15d7
 *
 * Triple-length key, single-length plaintext -
 * Key    : 0123 4567 89ab cdef fedc ba98 7654 3210 89ab cdef 0123 4567
 * Plain  : 0123 4567 89ab cde7
 * Cipher : de0b 7c06 ae5e 0ed5
 *
 * Triple-length key, double-length plaintext -
 * Key    : 0123 4567 89ab cdef fedc ba98 7654 3210 89ab cdef 0123 4567
 * Plain  : 0123 4567 89ab cdef 0123 4567 89ab cdff
 * Cipher : ad0d 1b30 ac17 cf07 0ed1 1c63 81e4 4de5
 *
 * d3des V5.0a rwo 9208.07 18:44 Graven Imagery
 **********************************************************************/
```