THE HARDWARE IMPLEMENTATION OF PRIVATE-KEY BLOCK CIPHERS

MOHSIN RIAZ

The Hardware Implementation of Private-key Block Ciphers

by

©Mohsin Riaz

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

July, 1999

St. John's Newfoundland Canada

Dedication

To my mom, Qaiser, whose selfless love and tenderness for me always make me strive for better in life.

Abstract

The National Institute of Standards and Technology (NIST) in the U.S. has initiated a process to develop a Federal Information Processing Standard (FIPS) for an Advanced Encryption Standard (AES) [1], to become the standard for private-key block encryption. The new encryption algorithm will be based on a 128-bit block size and the key size can be 128, 192, or 256 bits. AES will be a replacement for the Data Encryption Standard (DES) [2] which is based on a 64-bit block size and has a 56-bit key. In this regard, the agency has accepted candidate algorithm nominations for AES.

One of the important evaluation criteria concerns the efficiency of the private-key block cipher from the hardware implementation perspective . RC6 [3] and CAST-256 [4] are among the fifteen candidate algorithms that have been accepted in the first round of the AES development phase. This thesis investigates the efficiency of these two AES candidates from the hardware implementation perspective with Field Programmable Gate Arrays (FPGAs) as the target technology.

Our analysis and synthesis studies of both the ciphers suggest it would be desirable for FPGA implementations to have a simpler cipher design that makes use of simpler operations that not only possess good cryptographic properties, but also make the overall cipher design efficient from the hardware implementation perspective. As a result, the thesis also proposes a new private-key block cipher design that, not only is very efficient as far as its implementation in FPGAs is concerned, but at the same time is secure against the two most potent attacks that have been applied to block ciphers, namely, differential and linear cryptanalysis.

Acknowledgements

I wish to express my gratitude to my mentor and supervisor, Dr. Howard Heys for his valuable time, guidance, and financial support throughout the course of my research. His constant encouragement especially during hard times is not only appreciated, but will always be remembered.

A special thank you to my family and friends, who had always been there for me. To Bala and Sidhu - better friends I could not ask for - a heart-felt appreciation for reminding me what else there is to life.

I also appreciate the timely support provided to me by Mr. Michael Rendell while I was battling along with the CAD tools.

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

M	Original plaintext message
C	Encrypted ciphertext message
F	Round function
r	Total number of rounds of encryption
m	Number of inputs to an S-box
n	Number of outputs of an S-box
α	Number of S-boxes in a round function
K_{r_i}	5-bit round subkey for the i^{th} round
K_{m_i}	32-bit masking key for the i^{th} round
$N(f)$	Nonlinearity of an m-bit boolean function
$N(S)$	Nonlinearity of an S-box
K	Primary key of encryption in bytes
NL	Nonlinearity of S-box function
N_l	Total number of known plaintexts needed for linear cryptanalysis to be successful
N_C	Total number of chosen plaintexts needed for differential cryptanalysis to be successfi
AES	Advanced Encryption Standard
DES	Data Encryption Standard
NIST	National Institute of Standards and Technology
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Arrays
MPGA	Mask Programmable Gate Array
VPN	Virtual Private Network

LAN	Local Area Network
WAN	Wide Area Network
NBS	National Bureau of Standards
SPN	Substitution-Permutation Network
SAC	Strict Avalanche Criterion
XOR	Exclusive-OR
LUT	Lookup Table
SRAM	Static RAM
ASIC	Application Specific Integrated Circuits
CLB	Configurable Logic Block
IOB	Input/Output Block
FHC	Fast Hardware Cipher
CMC	Canadian Microelectronics corporation
CLA	Carry Lookahead Adder
BCLA	Block Carry Lookahead Adder
CRA	Carry Ripple Adder
CSEA	Carry Select Adder
CSA	Carry Save Adder
CPA	Carry Propagate Adder
T_{cpa}	Total delay for the 32-bit carry propagate adder
T_{enc}	Time to achieve one encryption
T_{clk}	One clock period in nanoseconds
τ	One logic level delay in nanoseconds

P_{Ω_r}	Probability of best r-round iterative characteristic
p_i	Probability of the output XOR, given the input XOR in round i
p_β	Probability that the linear approximation holds true
p_{lin}	Probability of one linear or affine function
p_k	Probability that k out of a total of eight S-boxes are nonlinear
p_l	Probability of the best linear expression for an r-round cipher

Chapter 1

Introduction

> Civilization is the progress toward a society of privacy. The savage's whole existence is public, ruled by the laws of his tribe. Civilization is the process of setting man free from men. *Ayn Rand, The Fountainhead (1943)*

In the recent years, there has been a great need for much improved techniques of securely transmitting and storing information. From electronic mail to cellular communications, secure web access to smart cards and electronic commerce, wireless LAN and WAN computer networks to virtual private networks (VPNs) - these and other new information-based applications will have far reaching consequences, affecting the way business is done as well as private communication and social interaction. As this happens, security aspects of communication systems are of growing commercial and public interest. Unfortunately, these aspects have been widely underestimated or ignored in the past. Today, however, there is high demand for expertise and high-quality products in the field of information security and cryptography.

Until recently, encryption products were generally in the form of specialized hardware.

These encryption/decryption devices plugged into the communications line and encrypted all the data going across the line. Although, software encryption is becoming more prevalent today, hardware is still the embodiment of choice for many military and commercial applications. As an industry trend, many companies in North America as well as Europe are developing cryptographic hardware for applications such as secure voice, fax and data networks, secure VPN cryptographic accelerators, protocol sensitive encryption for wide area networks, and DSP voice ciphering.

Speed and security are also important issues that play in the favour of the hardware implementation of encryption devices. Encryption algorithms involve many complex operations on the message or plaintext bits. Often these are not the type of operations that are incorporated into our typical desktop computers. The most widely accepted private-key block cipher, the Data Encryption Standard (DES) [2], introduced in 1977, runs inefficiently on general purpose processors. Although, some cryptographers have tried to shape their algorithms to suit software implementations, specialized hardware such as an encryption chip will likely emerge as the winner in efficiency. Another key factor that favours the hardware implementation of a block cipher is security. An encryption algorithm being run on a generalized computing machine has no physical protection. On the other hand, hardware encryption devices can be securely encapsulated to prevent this. Other factors that suggest a hardware implementation include cost, ease of installation, and lower power consumption.

The National Institute of Standards and Technology (NIST) in the U.S. has initiated a process to develop a Federal Information Processing Standard (FIPS) for an Advanced Encryption Standard (AES) [1]. The new encryption standard is based on a 128-bit block size and a 128, 192, or 256-bit key size. This standard will be a replacement for DES. This

thesis examines the hardware implementation of two private-key block ciphers, RC6 [3] and CAST-256 [4], in Field Programmable Gate Arrays (FPGAs). Both RC6 and CAST-256 are among the fifteen candidate algorithms that have been accepted in the first round of AES development phase. The thesis also proposes a simpler private-key block cipher design that is very efficient in terms of hardware implementation in FPGAs.

1.1 Motivation for the Research

The Data Encryption Standard (DES) [2], a private-key block cipher, is the most widely used cryptosystem in the world. DES was developed by IBM, as a modification of an earlier cryptosystem known as LUCIFER [5]. DES was first published in the Federal Register in 1975. DES was adopted as a standard for "unclassified" applications in 1977 by the National Bureau of Standards (NBS).

DES has been a target of criticism since its inception in 1977. One objection of DES concerns the mystery surrounding the design of its S-boxes, which being the only nonlinear component of the cryptosystem, is vital to its security. However, the most pertinent criticism of DES is that the size of the key, 56 bits, is too small to be really secure. After twenty two years, DES is nearing its demise and is theoretically breakable by two powerful cryptanalytical attacks of differential and linear cryptanalysis [6, 7].

The National Institute of Standards and Technology (NIST) has initiated a process to develop a Federal Information Processing Standard (FIPS) for an Advanced Encryption Standard (AES) [1] specifying an encryption algorithm for the twenty-first century as a replacement of DES. In this regard, the agency has announced a request for candidate algorithm nominations of AES. One of the important evaluation criteria concerns the efficiency

of the private-key block cipher from the hardware implementation perspective. RC6 [3] and CAST-256 [4] are among the fifteen candidate algorithms that have been presented to the first round of the AES development phase. Both ciphers are modifications of earlier generation ciphers (RC5 [8] and CAST-128 [9]) based on smaller (64-bit) block sizes. Like most proposed private-key block ciphers, RC6 and CAST-256 are clearly designed for efficient implementation in software.

This thesis discusses the issues that effect the hardware implementation of the two AES candidates, RC6 and CAST-256, in FPGAs. The two major aspects of speed and hardware complexity associated with the two ciphers are explored and a comparative analysis of the two ciphers in terms of implementation in FPGAs is presented.

As the result of our study of these two ciphers, we also propose a new private-key block cipher, specifically targeted for hardware implementation. This cipher is based on simpler operations that not only possess good cryptographic properties, but also make the overall cipher design efficient for implementation in custom architectures as FPGAs.

1.2 Outline of Thesis

The thesis is organized as follows:

- Chapter 2 presents a literature survey of the previous research that is relevant to our work.
- Chapter 3 examines the different issues pertaining to hardware implementation of cryptographic algorithms in FPGAs.
- Chapter 4 examines the design of RC6 and CAST-256 encryptions and their implementation in target FPGA devices.

- Chapter 5 presents the design of a new private-key block cipher based on simpler cryptographic operations and its implementation in FPGAs. The security of the proposed cipher against linear and differential cryptanalysis is also examined in this chapter.

- Chapter 6 summarizes the results of the thesis and presents certain suggestions for future work.

Chapter 2

Review of Previous Research

Security of information stems from the need for private transmission of both military and public messages. This need is as old as civilization itself. The ancient Spartans, for instance, enciphered their military messages. The first secure communication channels were very simple and their reliability depended on the physical security of messengers. Due to the invention of computer systems and the pervasive intrusion of computer networks, the spectrum of protection issues has been stretched. Many protection issues of modern day computer systems and networks are strictly related to the protection of communication channels. Due to the natural characteristics of any channel, we have a communication medium that is accessible to eavesdroppers, so physical security is meaningless. The only way to enforce security in communication channels is by the application of cryptography.

The term *cryptology* originates from Greek roots meaning " hidden" and "word" and is the umbrella term used to describe the entire field of secret communications. Cryptology further branches into two: cryptography and cryptanalysis. *Cryptography* is the art and science of transforming information into an intermediate form which secures that information

while in storage or in transit. As opposed to *steganography*, which seeks to hide the existence of any message, cryptography seeks to render a message unintelligible even when the message is completely exposed. *Cryptanalysis*, on the other hand is the aspect of cryptology which concerns the strength analysis of a cryptographic system or cryptosystem, and the penetration or breaking of a cryptosystem.

A *cryptosystem* is any system that employs methods of cryptography to encrypt a message. *Encryption* is a process that transforms the original message or information referred to as the *plaintext* into an encrypted message known as the *ciphertext*. This ciphertext is then transmitted over an insecure channel. When this ciphertext reaches the receiver, a reverse transformation process, referred to as *decryption*, is performed to recover the original plaintext from the corresponding ciphertext. This encryption/decryption scheme is also referred to as a *cipher* [10]. Figure 2.1 shows the encryption/decryption process in the context of an insecure communications channel. A block cipher is a function that maps N-bit plaintext blocks to N-bit ciphertext blocks, where N is the block length, which is 64 bits in the case of DES and 128 bits in the case of AES.

In 1948 Shannon [11] proposed two principles that present a sound theoretical basis for cryptosystems with good security, namely *confusion* and *diffusion*. Confusion employs substitution to hide the plaintext and the key. Diffusion spreads the confusion effect across the entire ciphertext, thereby masking any statistical properties of the plaintext.

The field of cryptography is divided into two main branches: *private-key* cryptography and *public-key* cryptography. The two types of cryptosystems are shown in Figure 2.2. In private-key cryptosystems, the same secret key is used both for encryption and decryption. Assuming the algorithm to be secure enough, the security of the cryptosystem is based on

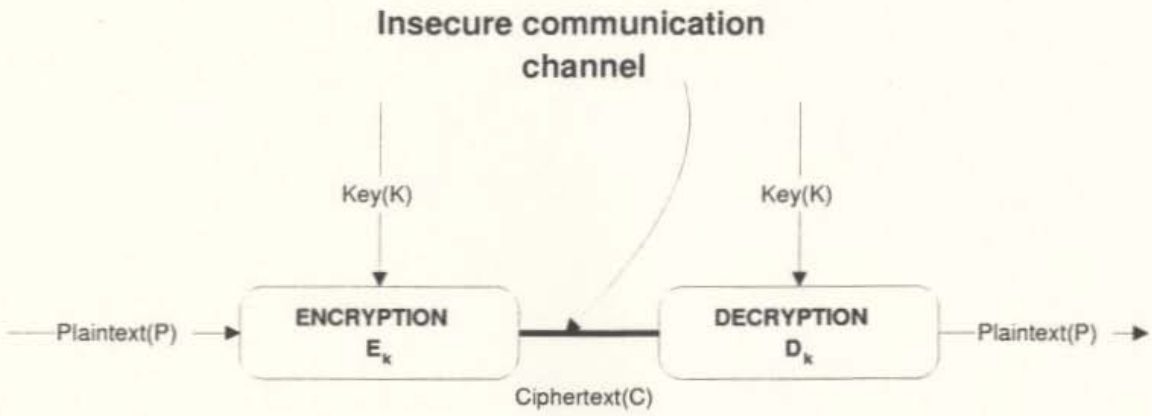

Figure 2.1: A General Cryptosystem

keeping the key secret. In contrast, every user in a public-key cryptosystem possesses two keys. One key is *public* and is known to everyone. The other is *private* and is only known to the person possessing it and no one else! Public-key cryptography has the advantage that a secure channel is not required to exchange keys. However, its disadvantage is that it is orders of magnitude slower to encrypt as compared to private-key cryptosystems. Most cryptosystems use a combination of public and private key cryptography. In a typical scenario, a public key scheme is first used to exchange the secret key that is then used for encrypting or decrypting the messages using a private key encryption algorithm.

2.1 Private Key Block Ciphers

The security of a private key block cipher depends on the communicating parties sharing the same common secret key. If this key is compromised, then the encrypted messages will be easily decrypted using the known key. The cipher is called a *block* cipher because the

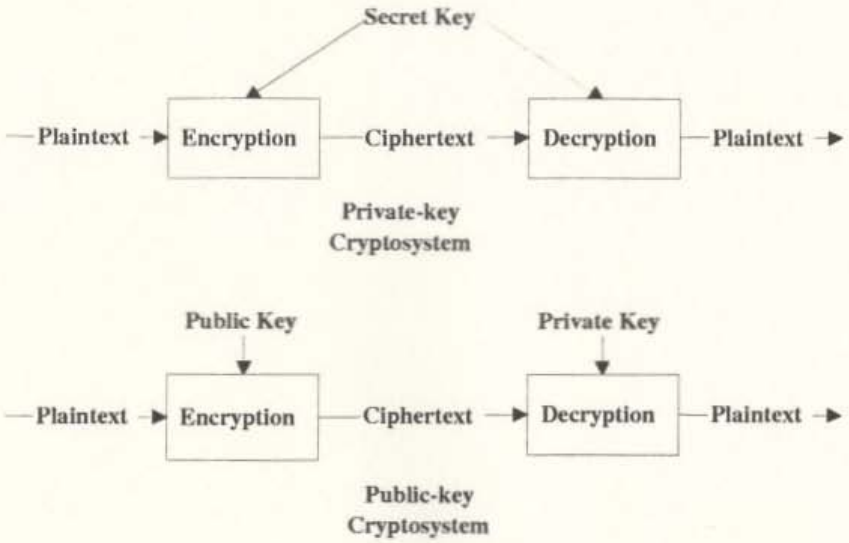

Figure 2.2: Private and Public Key Cryptosystems

plaintext is broken into fixed length blocks before being encrypted.

2.1.1 Architectures

The first practical private key block cipher designs based on Shannon's principles of confusion and diffusion were laid down by Feistel [5] and Feistel, Notz and Smith [12]. These design frameworks are referred to as *Substitution-Permutation Networks* (SPNs) and *Feistel Networks* or *Feistel* ciphers.

The SPN cryptographic network consists of a number of stages or rounds of substitution-permutation layers. Each substitution-permutation layer (SP layer) is made up of several smaller sub-block substitutions (known as S-boxes) followed by a large bit position permutation operation (known as P-box). The former has the effect of Shannon's confusion, while the latter operation implements Shannon's concept of diffusion. A primary key is used to

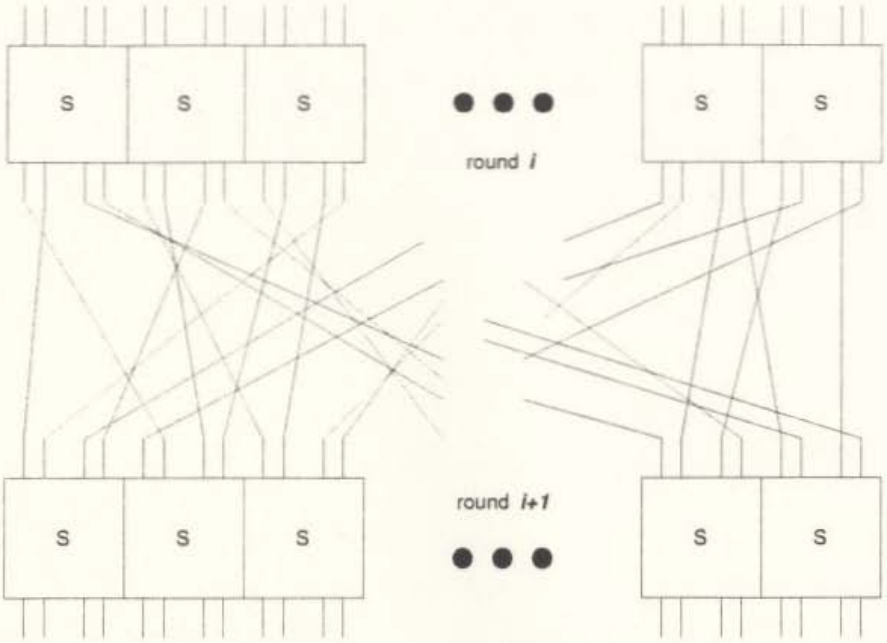

Figure 2.3: A Basic Substitution-Permutation Network

generate all subkeys implemented in each substitution-permutation layer according to a key schedule scheme. An $m \times n$ S-box has a nonlinear mapping from an m bit input to an n bit output pattern. The S-boxes for SPNs, however, must have same number of inputs and outputs. Such S-boxes are also known as *symmetric* S-boxes. Moreover, these mappings should be *bijective*, meaning that they are a one-to-one mapping and are invertible. Invertibility is needed for the purpose of decryption. Two stages of S-boxes in an SP structure based on 4×4 S-boxes are shown in Figure 2.3.

Another type of private key block cipher design is based on a Feistel network architecture proposed by Feistel, Notz, and Smith [12]. In a Feistel architecture, as shown in Figure 2.4, Shannon's mixing transformation can be achieved using S-boxes and permutations inside a round function f. But these operations are performed on only half the block at a time. For

each round, the right half is fed into a round function f whose output is bitwise XORed with the left half. This is followed by an immediate swapping of the two halves. After a total of R rounds, the two halves are concatenated to constitute the ciphertext block. The complete encryption process can be visualized as an iteration of the following operation:

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= f(R_i, K_i) \oplus L_i \end{aligned} \tag{2.1}$$

where R_i and L_i are the right and left halves, respectively, of the block for the i^{th} round. Also, K_i represents the i^{th} round subkey.

Decryption is similar to encryption, with the only exception that the subkeys are used in reverse order. The round function is the most critical component of the cipher design as it introduces an element of randomness to the plaintext. It is basically the structure of the round function that distinguishes between different Feistel ciphers. Feistel ciphers, unlike SPNs, can have asymmetric S-boxes, i.e. $m \neq n$.

Most of the existing commercial cryptographic implementations use DES for their private key algorithms. The Data Encryption Standard, first introduced in 1977 as an encryption standard for unclassified applications is based on a 64-bit block size. The key size is 56 bits. The round function expands the 32-bit input into a 48-bit block using an expansion table, followed by an XOR operation involving a 48-bit subkey generated by the *key schedule* scheme and the 48-bit expanded block. The resultant 48 bits are then fed into eight 6×4 S-boxes. The output of the eight S-boxes goes through a final 32-bit permutation giving the final 32-bit output of the round function.

The ciphers are typically keyed by applying subkey bits (derived for each round by the key schedule) to the S-boxes employing either:

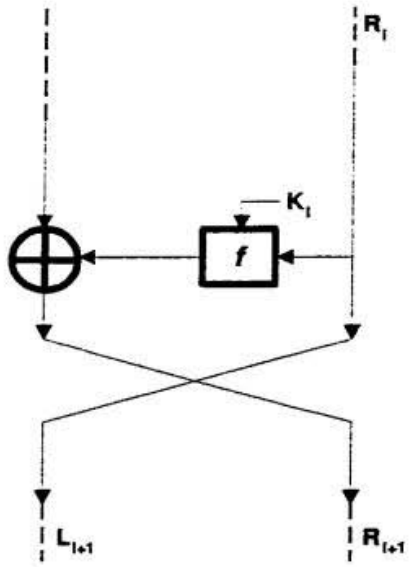

Figure 2.4: A Basic Feistel Structure

(i) selection keying: Here the key bits select the desired mapping for a particular S-box.

(ii) XOR keying: Here the key bits are XORed with the input bits before feeding into the S-box.

2.1.2 Popular Private-key Block Ciphers

Over the years, many private-key block ciphers have been proposed as potential replacements for DES. The structure of these block ciphers may or may not be a Feistel network. An introduction to some of the popular private-key block ciphers is presented here.

Blowfish [13] is an algorithm developed by Bruce Schneier. It is a block cipher with a 64-bit block size and variable length keys (up to 448 bits). It has gained a fair amount of acceptance in a number of applications. No successful attacks are known against it. Blowfish

is used in a number of popular software packages, including Nautilus and PGPfone.

FEAL [14] is a 64-bit block cipher with a 64-bit key. It is basically a Feistel structure. The 8×8 S-boxes in the round function execute XOR additions and byte rotations. The algorithm is well suited for 8-bit microprocessors. However, the down side of this cipher has to do with its resistance against differential cryptanalysis. It has been shown that the algorithm with less than 8 rounds can be easily broken using differential cryptanalysis [15]. The cipher is resistant to this kind of attack only if the number of rounds exceeds 32 [16].

IDEA (International Data Encryption Algorithm) is an algorithm developed at ETH Zurich in Switzerland [17]. It uses a 128 bit key, and it is generally considered to be very secure. The block size is again 64 bits. The algorithm uses a mix of three different groups of operations - bitwise XOR, integer additions and integer multiplications. It has already been around for several years, and no practical attacks on it have been published despite the number of attempts to analyze it. IDEA is patented in the United States and in most of the European countries. The patent is held by Ascom-Tech. Non-commercial use of IDEA is free.

RC5 [8] is a very efficient word-oriented secret-key block cipher. It is a parameterized family of symmetric ciphers. It uses a variable word size, a variable-length secret key and a variable number of encryption rounds. The architecture of this novel symmetric block cipher does not fall into the realms of a typical SPN or Feistel cipher. This algorithm makes use of data-dependent rotations. It also makes use of integer additions, subtractions and bitwise XORs. RC5 has been shown to be very resistant against both linear and differential cryptanalysis [18], although potentially susceptible to timing attacks [19].

CAST-128 [9] is another private-key block cipher that is based on a 64-bit block size.

It uses a 128-bit primary encryption key. The algorithm uses six 8×32 S-boxes. The strength of this algorithm has been shown to lie in the large size of the S-boxes [9]. Lee, Heys, and Tavares [20] showed that the algorithm is resistant to both linear and differential cryptanalysis.

2.1.3 Advanced Encryption Standard (AES)

As mentioned earlier, the National Institute of Standards and Technology (NIST) has unveiled a process to develop a Federal Information Processing Standard (FIPS) for an Advanced Encryption Standard (AES) [1]. The AES represents a specification for a private-key block cipher as a replacement for DES. As a part of the AES process, a number of minimum acceptability requirements have been drafted. These candidate algorithm evaluation criteria include:

- AES shall be a symmetric private-key block cipher.
- The adopted standard shall be publicly defined.
- AES should be suitable both for hardware and software implementations.
- The key length for the AES may be increased as needed.
- Candidate algorithms that meet the above requirements will be judged on the basis of the following factors:
    1. Computational efficiency
    2. Hardware complexity
    3. Encryption speed

4. Software suitability

5. Memory requirements

6. Flexibility

7. Licensing requirements

8. Simplicity

Fifteen candidate algorithms have been presented to the first round of AES development phase. Information on AES candidates can be found in [1]. CAST-256 and RC6 are two of the fifteen AES submissions and are investigated in this thesis. The emphasis of this investigation is on the hardware implementation of these ciphers in FPGAs. Both ciphers are strong candidates because they are modifications of their earlier versions (CAST-128 [9] and RC5 [8]). Like most of these proposed ciphers, CAST-256 and RC6 are designed for efficient implementations in software. But as one of the implementation requirements for an AES candidate, the hardware efficiency of these algorithms has to be thoroughly investigated. Detailed architectures of the CAST-256 and RC6 ciphers are presented in Chapter 4.

2.2 Cryptographic Properties of a Block Cipher

Since its adoption as a standard, DES has been the focus of most of the research in private-key cryptography. Much of the effort had been directed towards cryptanalyzing DES or investigating properties that might improve the overall security of the cipher. In this section, different cryptographic properties that are vital to the security of a block cipher are presented.

2.2.1 Nonlinearity

Nonlinearity is the most crucial feature in the design of private-key block ciphers. For instance, if there exists a linear relationship (on a per bit or per block basis) between the ciphertext output and the plaintext input, the cipher can be easily broken by reducing the cipher to a system of linear equations. These linear equations can be then solved using a small amount of known plaintext- ciphertext pairs. Typically, an S-box is the only nonlinear component of an SPN or a Feistel cipher. As such, the need to design highly nonlinear S-boxes makes the difference between a more or a less secure cipher.

An m-bit affine boolean function g is defined [21] as

$$g(X) = a_0 \oplus a_1 x_1 \oplus .... \oplus a_m x_m \tag{2.2}$$

where $X = [x_1, ..., x_m]$ is the m-bit binary input, $\oplus$ is the bitwise exclusive-or, and $a_i \in \{0, 1\}$, $0 \leq i \leq m$. The *Hamming distance* between two m-bit boolean functions, $f(X)$ and $g(X)$, is defined to be

$$d(f, g) = \#\{X \in \{0, 1\}^m | f(X) \oplus g(X) = 1\} \tag{2.3}$$

where $\#$ is the total number of m-bit binary inputs.

The nonlinearity of an m-bit boolean function f is defined as

$$N(f) = \min_{g \in A} d(f, g) \tag{2.4}$$

where A is the set of all m-bit affine boolean functions. Since an $m \times n$ S-box has n output bits, each of which is an m-bit boolean function, the nonlinearity of the S-box S is defined as the minimum nonlinearity over all non-zero combinations of output bit boolean functions:

$$N(S) = \min_{c_i \in \{0,1\}, \textit{not all } c_i = 0} N(\bigoplus_{i=1}^{n} c_i f_i) \tag{2.5}$$

where f_i is the m-bit boolean function of the i^{th} output bit of the S-box, and $(c_i f_i)(X) = c_i(f_i(X))$ for all X.

Heys and Tavares [21] used random search and filtering against known weaknesses to find highly nonlinear large S-boxes. They have used this technique in constructing S-boxes for SPNs.

2.2.2 Avalanche

Feistel, Notz, and Smith [12] first described the concept of *avalanche* as an important cryptographic property in the design of a block cipher. The avalanche property is satisfied only when, on average, half the output block bits vary when one input bit changes.

2.2.3 Completeness

Completeness was a concept introduced by Kam and Davida [22]. The completeness criterion is satisfied if all output bits depend on all input bits. Kam and Davida proposed a class of permutations in a basic SPN which ensures the completeness of an SPN, provided each S-box is complete. Brown and Seberry [23] found that DES is complete after four to five rounds with a high probability.

2.2.4 Strict Avalanche Criterion

Webster and Tavares [24] used the concepts of completeness and avalanche to come up with a new cryptographic property that concerns not only the individual S-boxes but also complete cryptosystems. This property is known as the *Strict Avalanche Criterion* or *SAC*. This property states that for every input bit, inverting the bit causes each output bit to vary with

a probability of one half over all possible input vectors. Higher order properties of SAC have been presented by Forré [25], Adams [26] and Preneel et al [27].

2.2.5 Information Theory

Many of the contributions to the field of cryptography come from the *information theory* concepts introduced by Shannon [11]. In a cipher that has *perfect secrecy* the plaintext is statistically independent of the ciphertext. This means that even with an unlimited time and computational resources at our disposal, we cannot guess the plaintext, given knowledge of the ciphertext. For a private-key cipher to be perfectly secure, the uncertainty in the key must be at least as large as that of the plaintext.

Dawson and Tavares [28] further investigated the work of Forré [29] in using information theory to design the S-boxes. They proposed minimizing the mutual information between a subset of output bits and any subset of input and/or output bits in the design of S-boxes for SPNs and Feistel ciphers.

2.2.6 Invertibility

An $n \times n$ S-box is said to be invertible if it is a bijective mapping. Adams and Tavares [30] proposed a method of constructing S-boxes such that it satisfies (*i*) bijection, (*ii*) minimum nonlinearity, (*iii*) SAC, and (*iv*) output bit independence by combining $0-1$ balanced boolean functions. However, O'Connor [31] was of the opinion that this technique becomes impractical as n increases.

2.3 Cryptanalysis of Private-key Block Ciphers

The purpose of cryptanalysis is to recover a secret primary key used in a particular cryptosystem. There are three types of general attacks that can be applied against any particular block cipher. These include *ciphertext only*, *known plaintext*, and *chosen plaintext*. In case of a ciphertext only attack, the cryptanalyst possesses the ciphertexts only. A known plaintext attack uses the knowledge of both plaintexts and their corresponding ciphertexts. In the case of chosen plaintext attack, the cryptanalyst can select particular plaintexts, and produces the corresponding ciphertext. This is possible only because he or she has temporary access to the encryption machinery.

The most fundamental way to break a cipher is exhaustive key search, referred to as the *brute force attack*. Two recent and powerful methods that have demonstrated the ability to break modern day block ciphers are *differential* and *linear* cryptanalysis. This section describes these two widely known attacks against private-key block ciphers as well. At the end of this section a new type of attack, called the *timing attack* is presented.

2.3.1 Brute force Attack

A brute force attack, also known as exhaustive key search is a known plaintext attack. In this kind of attack, the cryptanalyst gets hold of a few ciphertexts and their corresponding plaintexts. The next step is to exhaustively search all possible keys by encrypting a known plaintext with each of these keys. When one of the keys generates the correct ciphertext, we very likely have the correct key. We can use a few more ciphertext-plaintext pairs to verify the correctness of the key.

The best line of defense against this type of attack is to increase the key size such that

the attack becomes infeasible. Theoretically speaking, a cipher is broken, if the time and memory resources required by any cryptanalytic attack are less than what is needed for a brute force attack.

2.3.2 Differential Cryptanalysis

Differential cryptanalysis, developed by Biham and Shamir [6], is one of the most potent techniques used to cryptanalyze many private-key block ciphers. SPNs and Feistel ciphers belong to the class of iterated product ciphers and this attack is very much applicable to them.

Differential cryptanalysis is essentially a chosen plaintext attack. Biham and Shamir have successfully attacked DES using this technique and have found it to be more efficient than a brute force attack. This method takes into account ciphertext pairs, whose corresponding plaintexts have a particular difference. In other words, it looks at the XOR difference of two plaintexts and considers the corresponding ciphertext pair. In a particular S-box, if we know the input XOR of a pair, it does not ensure the knowledge of its output XOR. However, there exists a probabilistic relation between the output XORs and every input XOR. Differential cryptanalysis makes use of the highly probable occurrences of sequences of output XOR differences at each round given a particular plaintext XOR difference.

Several methods have been proposed to ensure the immunity of a round function against this type of attack. Several methods have been used to reduce these highly probable occurrences of output XORs in relation to input XORs. For example, this can be done by increasing the output bits of the S-box to some reasonable value [32]. A second approach uses a modular multiplication to mask the input of the S-boxes as a way to replace the XOR

operation in the round function that involves the subkey [33].

2.3.3 Linear Cryptanalysis

Linear cryptanalysis is a known plaintext attack, invented by Matsui [7], uses linear expressions to approximate the action of a block cipher. This attack exploits the statistical linear relations between plaintext, ciphertext and subkey bits. This implies that if we XOR some plaintext bits together, XOR some ciphertext bits together and then finally XOR the result, we end up getting a single bit that is equal to the XOR of some of the key bits with a probability that is significantly different than one-half. This defines a *linear approximation*, which holds with a certain probability. If this probability is different from one half, we can use this fact to construct a linear approximation of the entire algorithm. This is done by concatenating linear approximations of different rounds. Matsui, in his paper presented two algorithms used to derive the subkey bits using a linear approximation. Algorithm 1 is used to recover one subkey bit that is the XOR sum of a subset of subkey bits. The second algorithm, an extension of the first, determines a number of the subkey bits at one time.

DES is highly susceptible to this kind of attack as the S-boxes of DES are not optimized against this attack. When this attack is mounted against a 16-round DES, the cipher is broken with 2^{47} known plaintexts. As the attack greatly relies on the structure of S-boxes, the best way to increase the immunity of SPNs against linear cryptanalysis is to select highly nonlinear S-boxes. Alternate approaches to thwart linear cryptanalysis, involve the use of key-dependent rotations [8] and modular additions and subtractions [13].

2.3.4 Timing Attack

Another type of attack aimed at breaking a private-key block cipher is the *timing attack*. Based on the assumption that accurate timing measurements are available for individual encryptions, this attack employs the methodology of deriving the key bits using timing information from a set of ciphertexts. The timing attack of RC5 as outlined in [19], exploits the fact that a naive implementation of the cipher could result in data-dependent rotations taking a time that is a function of the data. This implies that it is important for the designers to be aware of different cryptographic issues when implementing ciphers like RC5. However, the timing attack can be prevented if a digital hardware implemention ensures that the rotations take constant time. A *barrel shifter* is one piece of digital hardware that can execute any size rotations in one clock cycle.

2.4 Security of RC6 and CAST-256 Encryption Algorithms

As mentioned earlier, CAST-256 and RC6 are among the fifteen candidate algorithms that have been presented to the first round of AES development phase. Although CAST-256 and RC6 are neither SPNs nor Feistel ciphers, their architectures are extensions of the basic Feistel cipher. This section briefly discusses the security of these two AES submissions against linear and differential cryptanalysis as well as against brute force attack.

In AES submission for RC6 [3], several modifications have been made such as the use of four working registers instead of two as in RC5 [8], and the introduction of a quadratic function that uses the primitive operation of integer multiplication. The use of multiplication

operation enhances the diffusion effect, thereby increasing the overall security of the cipher.

It has been conjectured in [3] that the best approach to attack RC6 block cipher is to adopt brute force attack. This is achieved by carrying out an exhaustive search for the user-supplied encryption key. Rivest, Robshaw, Sidney, and Yin [3] have concluded that the work load needed to exhaustively search for the b-byte encryption key or the expanded forty-four 32-bit subkeys (as a part of AES submission) is $min\{2^{8b}, 2^{1048}\}$ operations! As far as the linear and differential cryptanalytic attacks on this cipher are concerned, the data requirements to execute these attacks on RC6 exceed the available data. For instance, considering an 8-round version of RC6 would require more than 2^{76} chosen plaintext pairs for successfully mounting differential cryptanalysis, while it needs more than 2^{60} known plaintexts to apply linear cryptanalysis successfully. Clearly, application of these attacks to the 20-round version of RC6, as presented for AES submission makes these attacks impractical.

The security analysis of CAST-256 [4, 34] reveals that the cipher is resistant to both linear and differential cryptanalysis. The total number of known plaintexts needed for a 48-round linear approximation of CAST-256 is approximately 2^{122}, which is almost equal to the total number of plaintexts available (2^{128}). This implies that linear cryptanalysis is impractical against CAST-256. In case of differential cryptanalysis of CAST-256, we need more than 2^{140} chosen plaintexts which is much greater than the number of plaintexts available for a 128-block size. It therefore appears that CAST-256 is immune to differential cryptanalysis attack too.

2.5 Conclusion

We have introduced the fundamentals of cryptography in the beginning of this chapter, followed by a detailed investigation of private-key block ciphers. Here, we have presented two main architectures of block ciphers. Many popular private-key block ciphers have been described too. In addition, various cryptographic properties that are vital to the design and analysis of S-boxes and ciphers have been discussed. Next, different cryptanalysis techniques as applied to ciphers have been presented. Finally, we have discussed the security of two block ciphers, RC6 and CAST-256, against the different attacks presented.

Chapter 3

Hardware Environments for Cryptographic Applications

Two important communication revolutions have catalyzed the generation of an entire new information-based industry. The first revolution was the interconnection of data networks around the globe culminating in the Internet. The second is the recent availability of inexpensive high speed connections that link users at home or in the office to these networks. This growing trend of internetworking has led to the commercialization of on-line services and electronic commerce. This has resulted in a critical urge for data security.

Most modern day security applications make use of cryptographic hardware as a potent weapon against different security breaches and intrusions. With the demoniac growth of the Internet, the need for privacy, authenticity and anonymity has encouraged cryptography to surface as a viable means of achieving security. There are now a lot of engineering design companies that are shipping out cryptographic hardware for applications as diverse as electronic commerce and banking, secure wireless solutions, smart cards, PCMCIA card

security, certification authority, digital signatures etc. One of the most recent applications of cryptographic hardware has to do with the security of *virtual private networks* or VPNs as they are popularly known. These are cryptographic accelerator modules that act as fast coprocessors providing cryptographic processing at wireline speeds, freeing the router or firewall to perform other critical tasks while eliminating congestion in virtual private networks. Other examples of cryptographic hardware include LAN/WAN encryptors.

3.1 Hardware Encryption vs. Software Encryption

Any encryption algorithm can be implemented in software. But there are several disadvantages inherent to software implementations. These include lower speed, higher cost, and less security. The speed of encryption is basically restricted by the maximum clock frequency of the computing platform, whereas in case of a hardware solution, we can go for an extremely fast implementation such as a full custom ASIC implementation. Moreover, a software-alone solution is vulnerable to viruses, inadvertant erasing, complications from system failures, and hackers.

In contrast, hardware encryption has many advantages over software solutions. Encryption in hardware is faster. A hardware solution is impervious to system failures such as viruses. Hardware implementations can protect against internal and external intruders using two-factor authentication: both the hardware device and a password are necessary to access the root or primary encryption key.

Internal key management and distribution is better taken care of using a hardware encryption solution. Hardware solutions can control access to the root keys so that one can distribute access codes across several individuals who must cooperate to gain access to the

root keys. Moreover, hardware solutions offer scalable security.

3.2 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Array (FPGA) is a general-purpose, multi-level programmable logic device that is customized in the package by the end users. FPGAs are devices whose cores are populated with an array of logic structures of changing granularity and programmable interconnect used to connect them in several different ways. For instance, the logic blocks can be SRAM-based lookup tables (LUTs) or even multiplexers with or without registers, while special purpose routing switch boxes or segmented channels can make up the programmable interconnect.

The structure, size and number of blocks of logic as well as the amount of glue logic or the connectivity of the interconnection differs largely among FPGA architectures. This variation in FPGA architectures is dictated by different programming technologies and different target applications of the parts. This implies that an architectural arrangement that works well with a particular programming technology does not necessarily work with another.

Based on different programming technologies and architectural styles, FPGAs fall into four groups:

- Island-style SRAM-programmed devices.
- Cellular SRAM-programmed devices.
- Channeled, antifuse-programmed devices.
- Array-style EPROM or EEPROM-programmed devices.

SRAM-based island-style FPGAs include Xilinx LCA families. The Xilinx FPGA uses a fairly large logic block with table lookup functionality and two D flip-flops. Xilinx arrays

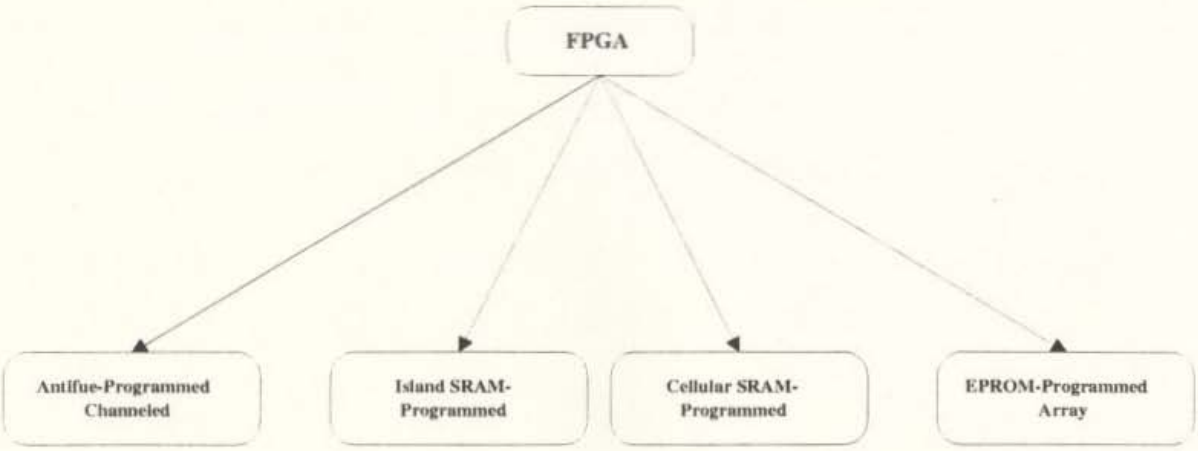

Figure 3.1: A Simple FPGA Taxonomy

have specialized routing blocks. This enables interconnection of a subset of inputs to one another. The AT&T Ocra and Altera Flex, as well as UTFPGA1 [35], also belong to this type of FPGA architecture. Toshiba, Plessey's ERA, Atmel's CLi family, the Algotronix CAL, as well as Triptych [36] FPGAs belong to the cellular-style architecture. Algotronix and CAL reuse some of the logic cells themselves to act as routing resources.

Antifuse-based channeled FPGAs include Actel's ACT1 and ACT2, Quicklogic's pASIC and Crosspoint's CP20K Series FPGA. Actel logic blocks are very small and multiplexer based [37]. Actel arrays use segmented channel resources. EPROM-programmed devices include Altera's MAX 5000 and MAX 7000, AMD's Mach and Xilinx's EPLD, as well as a few others. Altera logic blocks directly support multi-level combinational logic. A simple taxonomy of FPGAs is illustrated in Figure 3.1.

3.2.1 Advantages of FPGAs over MPGAs

In this section, we discuss the advantages of using FPGAs as a hardware implementation solution versus a masked programmed gate array (MPGA) implementation.

Low Tooling Costs: Every design to be implemented in a *mask programmed gate array* (MPGA) requires custom masks to construct the custom wiring patterns. The cost of each mask is several thousand dollars and this cost is then amortized over the total number of units being manufactured. As a consequence, masking charges for designs based on MPGAs are tremendous. In contrast, there is no custom tooling needed for FPGA designs, presenting FPGAs as cost-effective for most logic designs.

Rapid Turnaround: From the completion of the design to the delivery of the finished products, the manufacturing process takes several weeks in case of MPGAs. An FPGA on the other hand can be programmed in minutes by the end user. Faster design turnaround leads to faster product development and shorter time-to-market for new FPGA products. In [38], Reinersten found that in a design environment that caters the needs of a high-tech industry, a delay of six months in product delivery reduces the lifetime profits of a product by thirty-three percent.

Reduced Risk: The advantages of low initial non-recurring engineering charges and rapid turnaround design time implies that a redesign due to an error incurs low expenses and small delays. This encourages rapid prototyping and more aggressive logic design.

Efficient Design Verification: MPGA users have to verify their designs by extensive and elaborate simulation before manufacture, mainly, because of huge non-recurring engineering costs and long manufacturing delays. An MPGA design may include errors due

to inaccuracies or oversimplifications made in the simulation model. This is because there is need for long time simulations. FPGAs do not suffer from this dilemma. FPGA users may choose to use in-circuit verification, instead of simulating large amounts of time.

Low Testing Expenses: There are three types of costs associated with testing MPGA parts: on-chip logic for testing, generating test program and final parts testing when the manufacturing is done. The manufacturer's test program verifies that every FPGA will be functional for all possible designs that may be implemented on it. FPGA users do not worry about writing design-specific tests for their designs. This eliminates the need to build testability into the design. Moreover, since the test program for FPGAs is the same for all designs as opposed to MPGAs, it is reasonable to invest more effort on improving it. This achieves excellent test coverage, providing high quality ICs.

3.2.2 Disadvantages of FPGAs over MPGAs

Field programmable gate arrays (FPGAs) also have some disadvantages. These drawbacks are mainly due to the inherent nature of the technology itself. To begin with, FPGAs suffer from on-chip programming overhead circuitry which is responsible for the programming of a given part. The area occupied by programming overhead cannot be utilized by the end user. This results in low gate density for the FPGA. The programmable switch matrices and interconnects in the FPGAs are larger than their mask-programmed counterparts in MPGAs.

The programmable switches also increase signal delay by adding resistance and capacitance to interconnect paths. As a consequence, FPGAs are larger and slower than equivalent

MPGAs. FPGAs also exhibit some design limitations in relation to implementing configurable computing applications in them. For instance, FPGAs are well suited to algorithms composed of bit-level operations, such as integer arithmetic, but they do not render themselves efficient enough for implementing numeric operations, such as high-precision multiplication. In fact, dedicated multiplier circuits such as those used in microprocessor and DSP chips can be optimized to work more efficiently than those developed for configurable logic blocks available in FPGAs.

The on-chip memory provided by FPGAs for storing intermediate computational results is too little. This implies that most configurable computing applications need some sort of additional external memory. This will slow down the computations. However, researchers and industry people are developing newer and more advanced FPGA architectures that incorporate enough on-chip memory, very fast and efficient arithmetic processing and some special-purpose functional units. A very recent example of such FPGAs are Xilinx's Virtex FPGAs that not only possess great gate densities, but also harness very high speeds. The Virtex family FPGAs have broken density and performance barriers while offering unprecedented system level integration, achieving clock speeds in excess of 150 MHz.

3.3 SRAM-based FPGAs

In this section, we shall focus our discussion on the issues surrounding the SRAM-based FPGAs. This is because we have used these devices as our target technology for realizing the cipher designs in hardware.

SRAM-based FPGAs are the most popular. This is mainly due to their ability to *reconfigure*. Several researchers [39, 40, 41, 42, 43, 44, 36, 35, 45, 46] have all investigated

this class of FPGAs. An SRAM-based FPGA is programmed by downloading configuration memory from an external source. The configuration memory cells control the logic and the interconnect that execute the application function inside an FPGA. The RAM memory is not centralized, but rather distributed among the configurable logic blocks. This type of programming approach has many advantages as well as disadvantages attached to it. One obvious drawback is the *volatile* nature of programming. This means that when power is switched off, the device loses its programming and as such the FPGA is to reprogrammed every time its turned on. However, the disadvantage of volatility furnishes the benefit of *reprogrammability*. This ability makes SRAM-based FPGAs ideal for rapid prototyping. This results in very high quality devices. Since the same CMOS process as used in ASICs is employed to build these type of FPGAs, SRAM-based FPGAs benefit from process improvements driven by semiconductor industry. Finally, because these FPGAs implement logic using static gates, SRAM-based FPGAs have very low power consumption.

3.3.1 Xilinx XC4000 Structure

Xilinx FPGAs possess an array based structure, each chip comprising a two-dimensional array of logic blocks interconnected by horizontal and vertical routing channels. Xilinx introduced the first FPGA series, the XC2000 in 1985, and now offers three more generations: XC3000, XC4000 and XC5000 devices. A very recent device family is the Xilinx's Virtex series which is still undergoing field tests. Of all the device families introduced so far, Xilinx XC4000 is the one that has most proven its worth over the years. This is the most widely used FPGA family in industry. More information can be obtained from Xilinx data books [47]. A detailed description of this device family is presented.

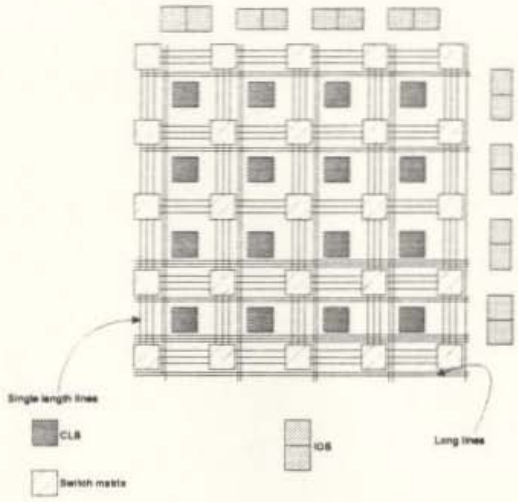

Figure 3.2: A Xilinx XC4000 structure

3.3.1.1 XC4000 Structure

The Xilinx XC4000 FPGA structure is illustrated in Figure 3.2. The logic inside the FPGA is implemented in an array of programmable blocks of logic called *configurable logic blocks* (CLBs). Input to and output from the array are taken care of by the input/output blocks (IOBs) along the edges of the array. The CLBs and the IOBs are interconnected using several types of programmable interconnect architectures. Connections to and from CLBs and IOBs can be programmed and wire segments can be interconnected, to form paths that extend from one box to another using an array of programmable connection blocks called the switch matrices.

3.3.1.2 Programming Technology

As mentioned earlier, Xilinx uses SRAM technology to store the programming information. After the power is applied to the circuit, the program data defining the logic configuration must be loaded into the SRAM. The FPGA itself contains the logic needed to load the information into itself from from a PROM. Once the programming information has been loaded, the device switches from *programming* mode to *operational* mode in which the logic is available. This logic is maintained as long as the device is powered up. As soon as the device is down, it loses it. The ability to reprogram the FPGA permits a new hardware design on the fly.

The SRAM bits control the logic that is implemented in a Xilinx XC4000 device. This is done using three techniques, namely *pass transistor control*, *multiplexer control*, and *table lookup implementation* [48]. Figure 3.3 shows an SRAM cell driving the gate terminal of an n-channel MOS transistor. When such a transistor is used to make or break a bidirectional connection for passing a signal between two wiring segments, it is called a pass transistor. When the SRAM cell contains a zero-bit, the transistor is OFF, the path between the two wiring segments is OPEN, and as such no control signal can be passed. On the other hand, when the SRAM bit contains a one, the transistor is ON, the path between the two wiring segments is CLOSED, permitting the signal to be passed. An XC4000 series FPGA contains tens of thousands of such pass transistors in the interconnection structure.

Figure 3.4 shows an SRAM cell connected to the select input of a 2×1 multiplexer. When the SRAM cell contains a 0, value on the zero input line is passed to the multiplexer output. The structure is used to make selections between two signals.

In the Figure 3.5, we have a lookup table (LUT) built using the SRAM cells. A LUT for

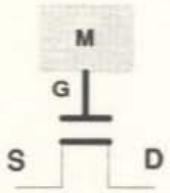

Figure 3.3: Pass Transistor Control Technique

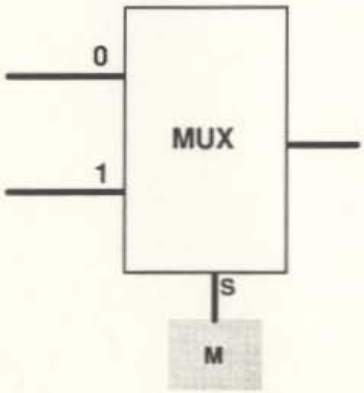

Figure 3.4: Multiplexer Control Technique

a three variable function $F(A, B, C)$ is illustrated. The SRAM cells in the LUT store the actual truth table for the logic function F. This implies that each cell houses the value of the function F for the corresponding minterm [49].

3.3.1.3 Interconnections

Connections between the CLBs as well as between CLBs and IOBs are established using wiring segments. These wiring segments extend in both the horizontal and vertical directions in channels lying between the various blocks. Some of the segments are very long, spanning the entire length or width of the array. Such segments are known as *long lines*. They are

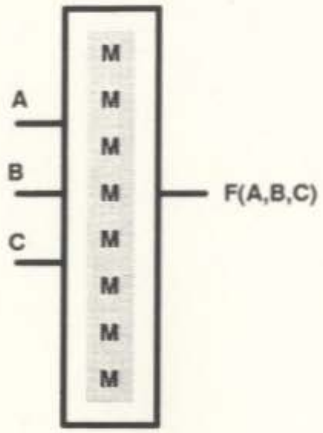

Figure 3.5: A Lookup Table Implementation

basically intended for high fan-out, time-critical signal nets, or the ones that are distributed over long distances [47]. Other segments are long enough to span a single CLB. These can be interconnected using the switch matrices. These are called *single-length lines.* Some of the XC4000 family devices even have *double-length lines* or *quad lines.* The benefit of using longer wire segments is that signal passes through less series resistance in traversing the same distance compared to single-length lines.

The discussion of Xilinx interconnections is incomplete without describing its *switch matrices.* An example of a switch matrix is shown in Figure 3.6. Here we have four segments meeting at a point. At this particular crosspoint, there are six pass transistors: one vertical, one horizontal and four on the diagonals. The horizontal and vertical pass transistors are shown as a plus sign as they intersect each other. The remaining four pass transistors, represented by black bold lines surround these two pass transistors. The connection between two segments is CLOSED for a one stored in the SRAM cell driving the transistor gate. Similarly, the connection is open for a zero stored in the SRAM cell. Thus, the SRAM-

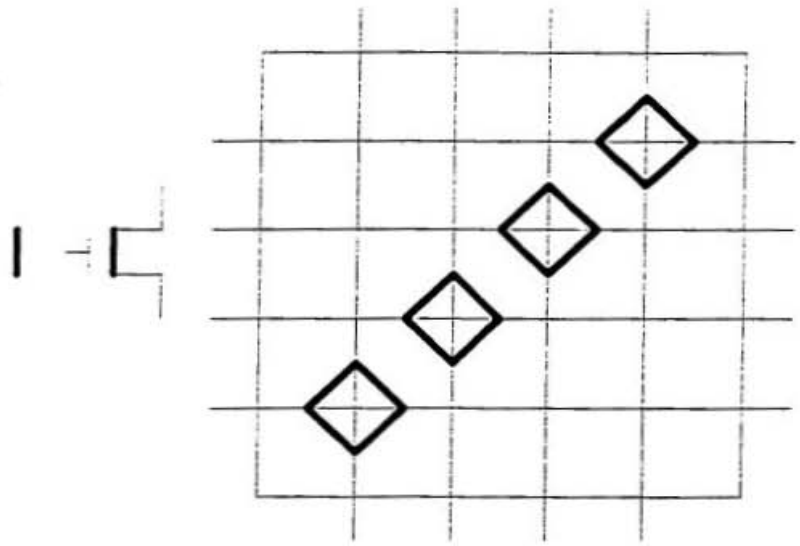

Figure 3.6: A Xilinx XC4000 Switch Matrix

controlled pass transistors lie at selected interconnections between the wiring segments in the routing channels [50].

3.3.1.4 Xilinx Logic

Most of the logic in a Xilinx XC4000 device lies within the CLBs and the IOBs. The structure of the CLB as well as of the IOB is internally programmable. A simplified representation of a Xilinx XC4000 CLB is shown in Figure 3.7 [47].

Here there are thirteen inputs to the CLB, including the clock input. Two flip-flops and their associated logic are also represented by broken lines. The remaining part of the CLB is used to implement the combinational logic. There are three LUTs that implement combinational functions. Two four-input tables implement two functions labeled F́ and Ǵ. A third function generator, H́ can implement any boolean function of its three inputs. Two

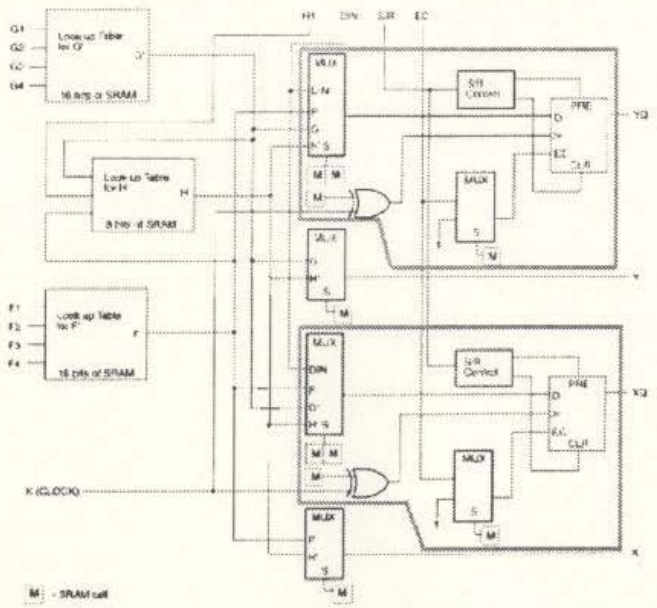

Figure 3.7: Simplified Logic Schematic of a Xilinx CLB

of these inputs can optionally be the F́ and Ǵ outputs of the two function generators. The third input essentially comes from outside the CLB.

A CLB can implement any of the following functions:

- Any function of up to four variables, any additional second function of up to four unrelated variables, plus any third function of up to three unrelated variables.
- Any single function of five variables.
- Any function of four variables along with some functions of six variables.
- Some functions of up to nine variables.

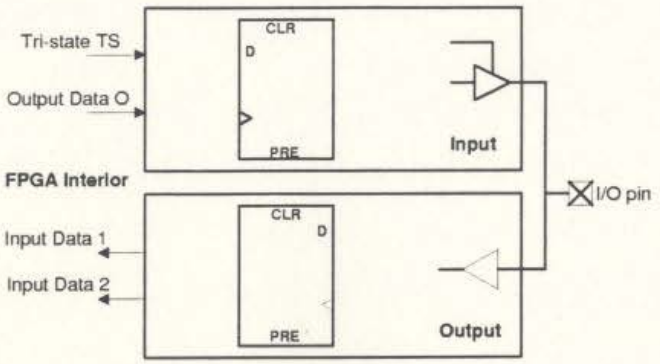

Figure 3.8: Simplified Block Diagram of Xilinx IOB

User-programmable input/output blocks (IOBs) supply the interface between external package pins and the internal logic. A simplified block diagram representation of a Xilinx XC4000 IOB is given in Figure 3.8.

To simplify things, the IOB block is divided into two sections - an input portion and an output part. The output part of the IOB gives the output data from the interior of the FPGA and the I/O pin. Alternatively, it can provide the stored value of the output data from a flip-flop. A tri-state driver on the output allows the I/O pin to be used as an input, an output or the input/output. In the input part, the signal at the I/O pin enters the input buffer. The signal can be fed directly to input data 1 and input data 2. These are two input lines to the interior of the FPGA or one or both of the inputs can be fed by a stored value from the I/O pin signal or its complement.

3.4 Cryptographic Algorithms: FPGAs vs. ASICs

A new development in integrated circuits that offers a hardware implementation choice that is much more flexible than Application Specific Integrated Circuits (ASICs) is the remarkable entry of reconfigurable custom hardware; these large, fast reconfigurable gate arrays are FPGAs. In contrast, ASICs provide only functionality needed for a specific task. A well-designed ASIC chip will support a particular application for which it is designed, but not a slightly modified version of the same application introduced after the ASIC design is completed. Furthermore, even if a modified ASIC can be developed, the original hardware is too highly customized to be reused in successive generations. In contrast, the configuration of an FPGA can be easily reprogrammed to accomodate a design modification.

Field Programmable Gate Arrays (FPGAs) have been chosen as the target technology for realizing the RC6 and CAST-256 cryptographic algorithms in hardware because of a number of reasons. Replacing one cryptographic algorithm with another is a trivial matter in software, but it is not the same in hardware. Moreover, at the same time hardware solutions can offer improved performance in terms of speed, security, and cost. As such the solution to this problem is reconfigurable hardware and FPGAs are the answer. As a matter of fact, FPGAs can be used to build *algorithm agile applications* [51]. The term algorithm agility refers to the fact that the same FPGA can be reprogrammed at run time to support different algorithms. Other key factors that favour the use of FPGAs for hardware implementation of ciphers include *faster turnaround design time*, *scalable security*, and *variable architectural parameters*.

3.5 Conclusion

In this chapter, we have touched different issues that relate to the hardware implementation of cryptographic algorithms in FPGAs. We have compared hardware encryption to encryption in software. Next, we introduced FPGAs as a viable custom architecture implementation choice. Here we have described different FPGA architectures, presented several key advantages and disadvantages of using FPGAs as a target technology. We have focused on SRAM-based FPGAs in general and Xilinx XC4000 family in particular. We have closed our discussion by reasoning why FPGAs have been chosen over ASICs for this particular application.

Chapter 4

Design of RC6 and CAST-256

In this chapter, we will basically focus on the FPGA implementation of two strong AES candidates - RC6 and CAST-256. We will investigate issues relating to the efficiency of the two ciphers from the hardware implementation perspective.

4.1 The RC6 Cipher

RC6 is a symmetric (private-key) block cipher submitted to NIST for consideration as the new AES. RC6 is an evolutionary improvement of RC5 [8]. Modifications have been made to meet the AES requirements, to increase security, and to enhance performance.

RC6-$w/r/b$, a general version of the RC6 cipher [3], operates on units of four w-bit words, with the encryption consisting of a nonnegative number of rounds r, and b representing the length of the encryption key in bytes. The user supplies a *primary* key of b bytes, where $0 \leq b \leq 255$ and, from this key, the key schedule scheme of the RC6-$w/r/b$ algorithm derives $2r + 4$ subkeys, where each subkey is a w-bit word. These $2r + 4$ subkeys are then stored in

the array $S[0,....,2r+3]$. This array of subkeys is used in both encryption and decryption.

Encryption with the RC6 algorithm is described below. RC6 works with four w-bit registers A, B, C, and D which contain the initial input plaintext as well as the output ciphertext at the end of the encryption. The standard *little endian* convention is used for packing the data bytes into the input/output blocks. The encryption block involves the following basic operations on two w-bit words a and b:

$a \oplus b$: bitwise exclusive-or of w-bit words

$a + b$: integer addition modulo 2^w

$a \times b$: integer multiplication modulo 2^w

$a \ll b$: rotate the w-bit word a to the left by the amount given by the least significant $\log_2 w$ bits of b

For the AES implementation of RC6, $w = 32$ and $r = 20$. Each of the four 32-bit registers A, B, C, and D is updated after each round of encryption. The output of the 20-round encryption is also stored in the four registers as the ciphertext. The entire process of encryption in the RC6 algorithm is illustrated in Figure 4.1.

Decryption is similar, but involves reversing the order of the subkeys, replacing left rotations by right rotations and replacing addition with subtraction. Since the AES submission requires the cipher to operate on 32-bit words and there should be twenty rounds of encryption/decryption, the issues related to hardware implementation of RC6-32/20/b version will be discussed in the coming sections. Note that for AES, $b \leq 64$ bytes (i.e. up to 512 bits of key) are allowed as *primary* key. For details on the *key scheduling scheme* refer to [3].

$$B = B + S[0]$$
$$D = D + S[1]$$
for $(i = 1;\ i \leq r\ ;\ i{+}{+})$
{
$$t = (B \times (2B + 1)) \lll \log_2 w$$
$$u = (D \times (2D + 1)) \lll \log_2 w$$
$$A = ((A \oplus t) \lll u) + S[2r]$$
$$C = ((C \oplus u) \lll t) + S[2r + 1]$$
$$(A, B, C, D) = (B, C, D, A)$$
}
$$A = A + S[2r + 2]$$
$$C = C + S[2r + 3]$$

Figure 4.1: Encryption with RC6-$w/r/b$

4.2 The CAST-256 Cipher

CAST-256 [4] is a private-key block cipher that is a generalization of the basic Feistel network [12]. CAST-256 algorithm uses a 128-bit block size and a 256-bit (or less) primary key that is used in the algorithm's key schedule scheme to generate two sets of subkeys, each of which is used per round: a 5-bit subkey K_{r_i} is used as a "rotation key" for round i and a 32-bit subkey K_{m_i} is used as a "masking key" for round i. There are a total of 48 rounds in encryption.

Three different 32-bit round functions are used in CAST-256. Using the same notation as for RC6, these functions are defined as follows:

- Round Function f_1

$$I = ((K_{m_i} + D) \ll K_{r_i})$$

$$O = ((S_1[I_a] \oplus S_2[I_b]) - S_3[I_c]) + S_4[I_d]$$

- Round Function f_2

$$I = ((K_{m_i} \oplus D) \ll K_{r_i})$$

$$O = ((S_1[I_a] - S_2[I_b]) + S_3[I_c]) \oplus S_4[I_d]$$

- Round Function f_3

$$I = ((K_{m_i} - D) \ll K_{r_i})$$

$$O = ((S_1[I_a] + S_2[I_b]) \oplus S_3[I_c]) - S_4[I_d]$$

Here D is the 32-bit data input to the round function, I_a to I_d are the most significant byte through the least significant byte of I, respectively, S_i is the i^{th} substitution box or S-box, and O is the 32-bit output of the round function. Each S-box is a nonlinear mapping of an 8-bit input to a 32-bit output [4]. Moreover, "+" and "-" are addition and subtraction modulo 2^{32} operations, "$\oplus$ " is bitwise exclusive-OR operation, and, finally, "$u \ll v$" is the rotation of u to left by the value indicated by v. The CAST-256 encryption algorithm is illustrated in Figure 4.2.

The plaintext is stored in four 32-bit input registers A, B, C, and D. There are 48 rounds in encryption. The four 32-bit registers are updated after each round of encryption. The output of the 48-round encryption is also stored in the four 32-bit registers A, B, C, and D as the ciphertext. Decryption is identical to encryption except that the masking and round keys derived from the primary key are used in the reverse order. Note that the 256-bit primary key can be generated from smaller user keys as outlined in the CAST-256 algorithm specifications [4]. Details of the *key scheduling scheme* for CAST-256 are also outlined in [4].

for $(i = 0;\ i < 6;\ i ++)$

{

$C = C \oplus f_1(D, K_{r_{4i+1}}, K_{m_{4i+1}})$

$B = B \oplus f_2(C, K_{r_{4i+2}}, K_{m_{4i+2}})$

$A = A \oplus f_3(B, K_{r_{4i+3}}, K_{m_{4i+3}})$

$D = D \oplus f_1(A, K_{r_{4i+4}}, K_{m_{4i+4}})$

}

for $(i = 6;\ i < 12;\ i ++)$

{

$D = D \oplus f_1(A, K_{r_{4i+1}}, K_{m_{4i+1}})$

$A = A \oplus f_3(B, K_{r_{4i+2}}, K_{m_{4i+2}})$

$B = B \oplus f_2(C, K_{r_{4i+3}}, K_{m_{4i+3}})$

$C = C \oplus f_1(D, K_{r_{4i+3}}, K_{m_{4i+3}})$

}

Figure 4.2: Encryption with CAST-256

4.3 Hardware Development Environment

The design cycle and CAD tools used for the hardware implementation of RC6 and CAST-256 algorithms have been provided by Canadian Microelectronics Corporation (CMC) [52]. The entire design process can be divided into the following three stages:

- Generating the VHDL (IEEE 1076) descriptions of the cipher design, employing different architectural options. The functional VHDL simulation of the design is carried out using the Synopsys VSS simulator version 1998.02 to verify the correct operation

of the cryptographic algorithm.

- Gate-level synthesis and logic optimization of the design utilizing Synopsys Design Compiler version 1998.02 to produce a functionally equivalent schematic in hardware.

- Place and route for a specific FPGA device followed by a final verification of the design. The timing simulation data is generated during this design stage which is then used to carry out timing simulation for the final verification of the design.

We chose Xilinx as the FPGA vendor and an XC4000 device family. In particular, we used XC40200XV-9-BG560 as our target device. This particular FPGA has a total of 7056 configurable logic blocks (CLBs), which gives us a baseline with which we can measure FPGA resource consumption. Xilinx Alliance Series version 1.5 is used for place and route.

4.4 Design of RC6

The block diagram representation of the RC6 encryption algorithm realized in hardware is shown in Figure 4.3. The RC6 core basically consists of four components - a 32-bit adder, a 32×32 "partial" integer multiplier (i.e. the product is modulo 2^{32}), a 32-bit bitwise exclusive-or (XOR) and a 32-bit barrel shifter. The control path of the RC6 encryptor consists of state machine/controller unit that controls the various modes of operation of the cipher. Other major components of the design that implement the *glue logic* include shift registers, multiplexers/demultiplexers, serial-in parallel-out (SIPO), parallel-in serial-out (PISO) and parallel-in parallel-out (PIPO) registers. As an example, the SIPO register takes a stream of 32-bit words and converts every four consecutive words to four parallel output words.

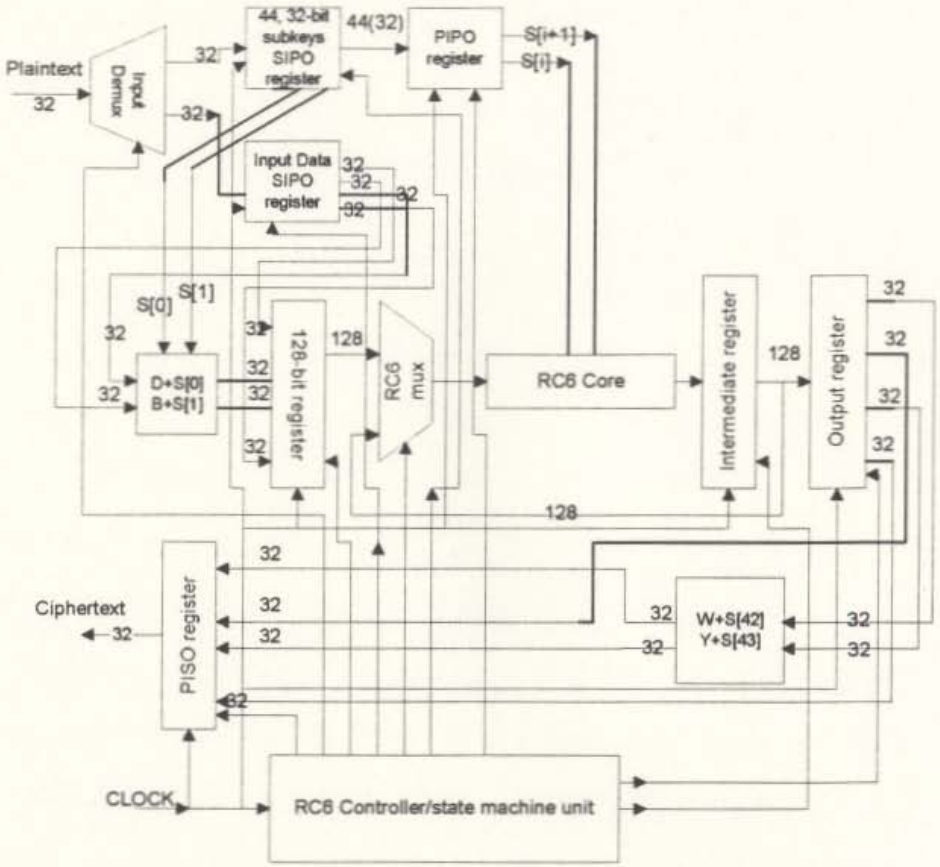

Figure 4.3: Realization of RC6 Encryption in Hardware

The implementation of the RC6 encryption algorithm is based on a *single-stage iterative* architecture. This particular architectural option involves the hardware for one round of encryption. The control path of the encryptor is designed such that data flows through the RC6 core for a total of 20 rounds as required for the AES submission.

4.4.1 RC6 Datapath

The datapath for the RC6 encryption, as illustrated in Figure 4.4, consists of two parallel functional pipes. First there are two initial modulo 2^{32} additions of the two 32-bit data blocks with the two corresponding subkeys. These additions are executed in parallel. The other two 32-bit data blocks are pushed into the two identical functional pipes without any modifications. Each of the two functional pipes, as indicated by the operations within the two dotted traces in Figure 4.4, are comprised of a number of operations. The 32-bit data is first fed into a *quadratic* function $f = 2x^2 + x$ used in the cipher for enhancing the rate of diffusion thereby improving the security of the cipher. The quadratic function f can be implemented using an addition followed by a multiplication operation. Here it should be noted that all these are modulo 2^{32} operations. The 32-bit output of the quadratic function goes through a left circular shift or rotation of 5 bits. This is followed by a 32-bit XOR operation, another left rotation, and another modulo 2^{32} addition with the 32-bit subkey for this particular round. The amount of rotation is determined by the least significant five bits coming from the other path. Note that the two functional pipes are not independent of each other. Before the end of the first round, the four 32-bit modified words are swapped. This is done to increase the nonlinearity of the scheme. Finally, the four 32-bits outputs of the two functional pipes are stored back into the four 32-bit registers. The same process is repeated for a total of twenty rounds. Upon completion of twenty iterations, two of the four 32-bit output blocks undergo final 32-bit additions with the last two subkeys. The other two 32-bit output blocks are passed out without these final additions.

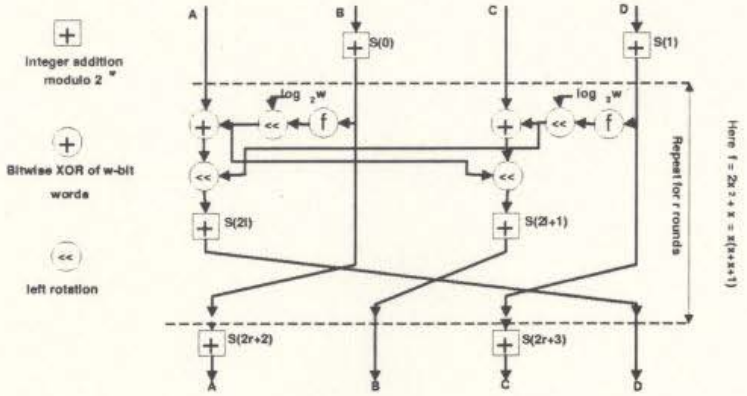

Figure 4.4: Functional Representation of RC6 Datapath

4.4.1.1 Design of 32-bit Barrel Shifter

One of the major concerns in the design of the RC6 core has to do with the data-dependent rotations. We have to look for an implementation that would take constant time for these rotations, irrespective of the size of the rotation. The need for constant time rotations stems from the fact that the RC6 algorithm is vulnerable to the *timing attack* [19] that may ultimately lead to breaking the cipher. This attack exploits the fact that it takes a variable amount of time to encrypt different plaintexts. This vulnerability occurs if the data-dependent rotations take a time that is a function of the data.

The solution to this problem has to do with the way we implement these data-dependent rotations. A barrel shifter is a device that can shift any number of bits in one clock cycle. We have designed a 32-bit barrel shifter at the behavioral level and our implementations in FPGAs reveal the following synthesis results:

- Maximum delay for the data to be shifted = 4.88 ns

- Total number of CLBs used = 369 (5.2 % of the total available CLBs)

Although the total number of configurable logic blocks is much greater than a normal serial shifter, the barrel shifter is much faster and takes only one CLB propagation delay time for any size rotation.

4.4.1.2 Design of 32-bit Adder

The implementation of a fast, low complexity 32-bit adder involves the consideration of a number of design choices.

We first explored a 32-bit *carry ripple adder* (CRA) implementation in an FPGA. A 32-bit CRA is made up of 32 stages of full adders, with the carry out of the preceding stage feeding in as the carry in to the following one. When implemented in an FPGA, the synthesis results are as follows:

- Maximum delay = 173.21 ns
- Total number of CLBs used = 32 (0.45 % of available FPGA resources)

The reason for this large delay is the way the 32-bit CRA is constructed as shown in Figure 4.5. It involves 32 CLB delays because the full adder at stage i has to wait for a possible carry from stage $i-1$, which in turn has to wait for a possible carry from stage $i-2$, and so on.

A *carry lookahead adder* (CLA) is the fastest of all adders as it has a maximum delay of four logic levels, irrespective of the adder width [53]. The components of this timing delay are one logic level for the carry propagate, carry generate, and partial sum signals, two delays for the carry signals, and one more delay for assimilating the carries and the partial sums.

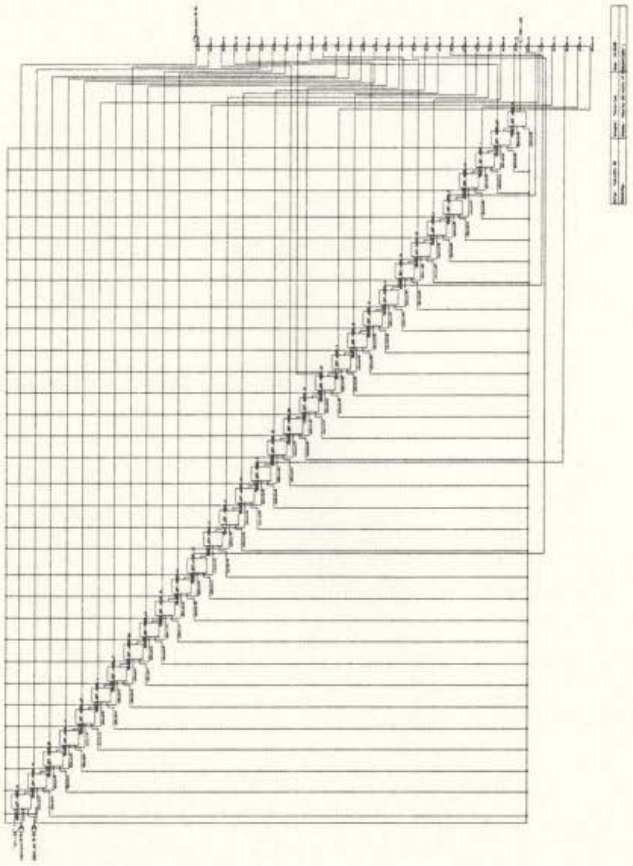

Figure 4.5: Configurable Logic Block Schematic of the 32-bit Carry Ripple Adder

However, it is not practically feasible to implement a CLA that adds numbers greater than eight bits because of the very high complexity and the limitations that arise from potentially high fan-in and fan-out requirements. As a consequence, implementing a *pure* 32-bit CLA is not a likely option.

Another way to design a practical CLA is to alter the basic design principle of the *ripple carry lookahead adder* (RCLA) that uses 4-bit CLA blocks with an inter-block ripple. However, our synthesis studies in relation to implementations in FPGAs reveal that a much

more efficient implementation is to design a *hierarchical carry lookahead*, also known as *block carry lookahead adder* (BCLA) [54]. In this particular design approach, we ripple the carries within the 4-bit adders, and generate carries between the 4-bit adder blocks using CLAs. This particular adder implementation results in the following numbers:

- Maximum delay = 70 ns
- Total number of CLBs used = 60 (0.85 % of the available CLBs)

The maximum delay associated with this implementation is reduced by a factor 2.45 over the 32-bit CRA, but at the same time the hardware complexity is increased by a factor of 1.875. Other 32-bit adder design choices that have been investigated include a pure 32-bit *carry select adder* (CSEA) as well as a 32-bit CSEA that incorporates a 4-bit CRA as the basic unit. A carry select adder is an adder organization that introduces redundant hardware to make the carry calculations go faster [55].

The adder finally selected for our design is a *hybrid* of CLA and CSEA. Our synthesis studies using FPGAs reveal that such a design is preferred on the basis of its speed. This particular design uses a modular architecture, wherein a 4-bit pure CLA is used to form an 8-bit CSEA and an 8-bit CSEA is then used to develop a 16-bit CSEA and so on. Our FPGA implementation of this design yields the following results:

- Maximum delay = 39.32 ns
- Total number of CLBs used = 197 (2.79 % of the available FPGA resources)

These results suggest that the *hybrid* approach is very fast (an improvement by a factor of 4.4 over the CRA implementation). But since there is always a trade off between speed

and hardware complexity, this design achieves this high speed at the expense of increasing hardware complexity.

4.4.1.3 Design of 32-bit XOR

A digital hardware component is needed to perform the 32-bit bitwise exclusive-or operation. The timing and FPGA resource reports for the synthesized 32-bit XOR unit are as follows:

- Maximum delay = 4.88 ns
- Total number of CLBs used = 16 (0.2 % of FPGA CLB resources)

These results suggest the cost effectiveness of this simple operation in the context of implementing an encryption algorithm in an FPGA.

4.4.1.4 Design of 32×32 "Partial" Integer Multiplier

The design of an efficient multiplier has been the corner stone of the RC6 core implementation in FPGAs. In particular, we need a 32×32 "partial" integer multiplier to compute integer multiplication modulo 2^{32}. This implies that we need to obtain only the least significant 32 bits of the 64-bit product. Our initial behavioral level implementation of the multiplier was very discouraging. The synthesis results for the implementation of the partial multiplier in the target FPGA furnished the following numbers:

- Maximum delay = 294 ns
- Total number of CLBs used = 551 (7.8 % of the available CLBs)

This low speed, less efficient FPGA implementation of the multiplier forced us to investigate a structural design option for the multiplier. In this respect, we have considered

a number of multiplier designs and their implementation in FPGAs. The different multiplier architectures considered from the FPGA implementation perspective include a *serial* multiplier [56], *ripple carry array* multipliers [57], *row adder tree* designs [58], *parallel array* multipliers [59], *lookup table (LUT)* multipliers [60], and *Wallace tree* multipliers [61].

The serial multiplier is the one that uses a serial adder for computing the partial sums and therefore produces the product at a rate of one bit per multiplier-cycle. The operational time for this type of multiplier is $3n\tau$ per cycle and $3n^2\tau$ for the entire multiplication process, where, τ represents delay of one logic level and n is the operand size in bits. Clearly, implementation of this sort of a multiplier design is extremely slow in FPGAs.

A ripple carry array multiplier (also known as *row ripple* architecture) is an unrolled embodiment of the classic shift-add multiplication algorithm. This sort of design has a maximum delay of the order $2n\tau$, if one ignores the routing delays. Implementations in FPGAs of this particular structure suggest that it does not make efficient use of the logic available inside the target FPGA and is found to be slower than many other implementations.

A variant of the ripple carry array multiplier is the row adder tree multiplier that uses an optimized form of row ripple. Basically, the gate count is same as for row ripple one, but it improves on the delay by arranging the row adders in tree. But it has been found that routing such a design in an FPGA is very cumbersome and the design seems to be workable in certain FPGAs only [62].

In principle, we can evaluate any finite function by using a lookup table (LUT) that is addressed with the arguments for the evaluation and whose output is the result of the evaluation. In theory, this furnishes the fastest possible implementation, as no actual arithmetic is needed. In the case of multiplication, however, the use of a single lookup table is not prac-

tical for any but the smallest operands. This is because the table size grows exponentially. LUT multipliers are simply a block of memory containing a complete multiplication table of all possible input combinations. The large table sizes needed for even modest input widths make these impractical for implementations in FPGAs, especially with their limited on-chip memory. For instance, a single table for 32-bit $\times$ 32-bit integer multiplication would have a size of 2^{64} words $\times$ 64-bit, which is simply out of question.

A better multiplier design approach is the use of parallel array multipliers. Here the term *parallel multiplier* refers to any multiplier that employs two or more adders in the *adder* section. This implies that we can have multiple additions in one cycle. In this particular design, we first have n^2 AND gates operating in parallel, generating the logical-AND of bit i of the multiplier and bit j of the multiplicand, called the *multiplicand-multiples.* This is followed by a number of logic layers of *carry save adders* [63] to add different multiplicand-multiples along with different carries that are generated. The last stage uses an n-bit *carry propagate adder* (CPA) to sum the bits coming out from the penultimate stage. The operational time is made up of τ, where τ represents on logic level delay, for generating the n^2 multiplicand-multiples, 3τ for each carry save adder, and the delay associated with the n-bit CPA (T_{cpa}). This final n-bit adder can be any of the n-bit modulo 2^{32} adders discussed earlier. Here it should be noted that each carry save adder stage is implemented using full adders as well as half adders.

Varieties of parallel-array multipliers, for operand sizes ranging from four bits to sixteen bits, have been commercially available for many years. However, for larger operands, high fan-out requirements and costs are generally prohibitive. One of the major contributors to the total delay associated with the parallel-array multipliers has to do with the way the

carry save adders are arranged in each row of the array. Taken to its logical conclusion, the approach of having as much concurrency as possible in the operation of the carry save adder, yields a class of multipliers known as *Wallace tree multipliers* or *simultaneous multipliers* [61].

The basic design principles for these kind of multipliers is explained as follows. First, all n multiplicand-multiples for n-bit $\times$ n-bit multiplication are generated concurrently as n^2 ANDs. Then a number of addition steps follow. Assume $n = 3l_0 + m_0$, where $0 \leq m_0 \leq 2$. In the first addition step, each of the l_0 triplets is reduced in a carry save adder to two sets of outputs. At this point we have $2l_0 + m_0 = 3l_1 + m_1$, where $0 \leq m_1 \leq 2$, outputs. In the second addition step, each of l_1 triplets is once again reduced in a carry save adder to two outputs. This process is repeated until only two outputs are left and these are then fed in a carry propagate adder (CPA). As each step reduces the number of outputs by a factor of 3/2, the complete process of reduction to two outputs takes $\lceil \log_{1.5} n/2 \rceil$ steps.

A Wallace tree is an implementation choice that is designed for minimum propagation delay. A Wallace tree approach rearranges the wiring so that the partial product bits with the longest delays are wired closer to the root of the tree. This changes the delay characteristics from $O(n^2)$ to $O(n \log(n))$ without increasing the hardware in comparison to parallel array approach. So, finally we find this approach to be quite suitable for implementation in FPGAs.

An important consideration at this point is the requirement of producing only the least significant 32-bits of the 64-bit product for our *partial* multiplier. As a consequence, we could easily exploit this concurrency (i.e. the parallel adder columns) that comes with the Wallace tree design. A high-level organization of Wallace tree multiplier for our 32 $\times$ 32 *partial integer* multiplier is shown in Figure 4.6. Here it should be noted that the adder blocks labelled as CSA in Figure 4.6 represent the carry save adders. Basically, the carry

save adders can be implemented using full adders with carry inputs. A carry save adder essentially takes in three inputs and generates two outputs, the carry-out C and the sum bit S. In other words, after each stage of addition, the carry save adder reduces the number of inputs going into the next stage by one.

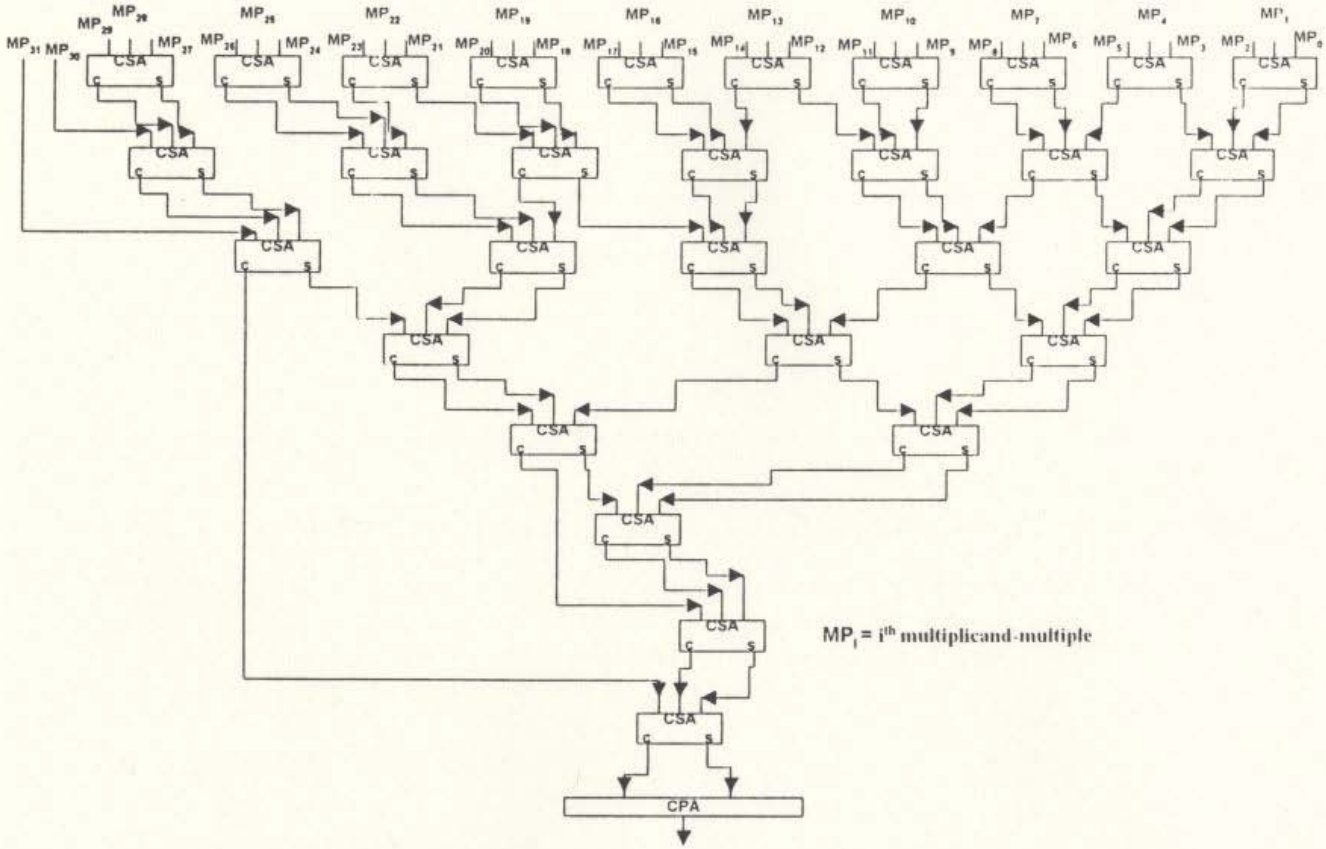

Figure 4.6: A High Level Organization of Wallace Tree Multiplier

The extra hardware needed to compute the most significant 32 bits of the 64-bit product is removed. In other words, we are using the hardware that is only required to produce the partial products used in generating the least significant 32 bits of the 64-bit product. The total delay constitutes one delay associated with the hardware producing the multiplicand-

multiples, eight full adder delays plus a final delay for the 32-bit CPA. The final 32-bit CPA is implemented using the *hybrid adder*. Our synthesis of this multiplier design in a target FPGA furnished the following numbers:

- Maximum delay = 79 ns (an improvement by a factor of 3.8 over the original synthesized behavioral description)
- Total number of CLBs used = 930 CLBs (13 % of the available CLBs)

An interesting observation has to do with the maximum delay number of 79 ns for the FPGA implementation of the multiplier. Almost half of the total delay (i.e. 39 ns) is contributed by the final stage 32-bit adder.

4.4.2 RC6 Control Path Design

The control path for the RC6 encryptor consists of two major functional units besides some glue logic. These are a *global state machine* unit and a *data flow controller* used for ensuring the proper operation of various components during the encryption process.

4.4.2.1 RC6 Global State Machine

The design of a synchronous finite state machine is vital to the proper operation of the RC6 encryptor, as the cipher cycles through a number of modes of operation. These include *reset* mode, *key-download* mode, *plaintext data-download* mode, *idle mode*, and the *data-encrypt* mode. A component interface representation of the RC6 encryption is illustrated in Figure 4.7. Here it should be noted that the RC6 encryptor design essentially encrypts the data only. However, decryption can be accomplished with minor modifications to the

original design. The cipher has three asynchronous inputs (RESET-CHIP, KD, and DD), a 32-bit data-in, a 32-bit data-out, a clock input and finally a status flag output. The status flag is used to indicate the mode of operation of the cipher.

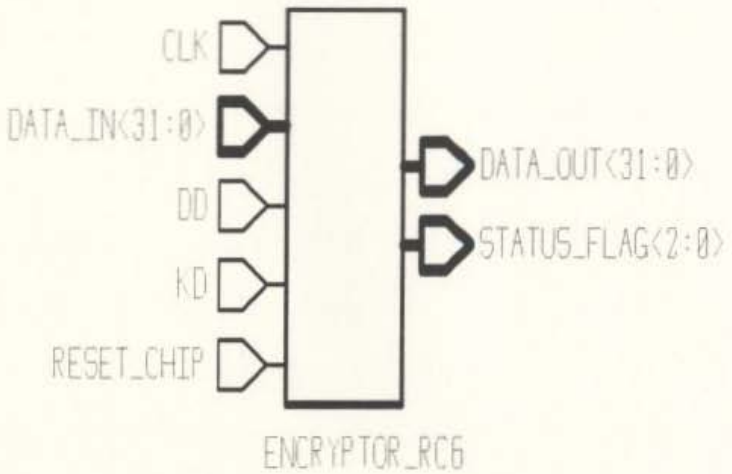

Figure 4.7: A Component Interface of RC6 Encryptor

When the cipher is powered up, it is in the reset mode, with all the registers and flipflops being initialized. Now, in order to start the normal operation of the cipher, the *reset-chip* input is disabled, and the cipher enters into the key-download mode. During the key-download mode, the forty four 32-bit subkeys are downloaded into the cipher *key storage* unit. Once all forty four subkeys are downloaded into the encryptor (indicated by the status flag), then depending upon the choice of the user-select inputs KD and DD, the cipher can either proceed forward to start downloading the plaintext or it can go into the idle mode. The idle mode is a feature incorporated into the design of the state machine so that

we can have enough flexibility among the key-download, data-download, and data-encrypt stages. During the normal operation of the cipher, once the keys have been downloaded, the cipher enters into the plaintext data-download mode. Once the 128-bit plaintext is downloaded, the cipher enters the data-encrypt mode. During the data encrypt mode, the plaintext undergoes twenty rounds of encryption, finally coming out of the cipher as the 128-bit ciphertext (i.e. as 128-bit encrypted data). From then onwards, the cipher can synchronously download the 128-bit data and encrypt it. The component interface of the RC6 global state machine is shown in Figure 4.8. The design of the RC6 state machine is based on a synchronous finite state machine with asynchronous inputs. The RC6 state machine has a number of synchronous inputs such as DONE-4, DONE-44, DONE-OUT, COUNT-20. These inputs to the state machine come from different counters used during the execution of different modes of operation of the cipher. As well, we have a number of reset and enable outputs that feed into the datapath of the cipher and make sure that different registers and multiplexers/demultiplexers are enabled or disabled at the appropriate time. A sample VHDL code for the RC6 state machine design is included in the Appendix A.

Essentially, the RC6 state machine is based on *Moore finite state machine* approach [64]. This finite state machine model is characterized by the fact that the outputs are identified solely with the present state of the device. Our global state machine has been designed keeping in mind that our main intent is to investigate the efficiency of the encryption algorithms from the hardware implementation perspective. As a result, the state machine design can not be regarded as a very robust one. This is because the design does not fully account for error conditions. Obviously, one can spend more resources to develop a more robust and sophisticated state machine for the design, but that has not been the focus of our attention.

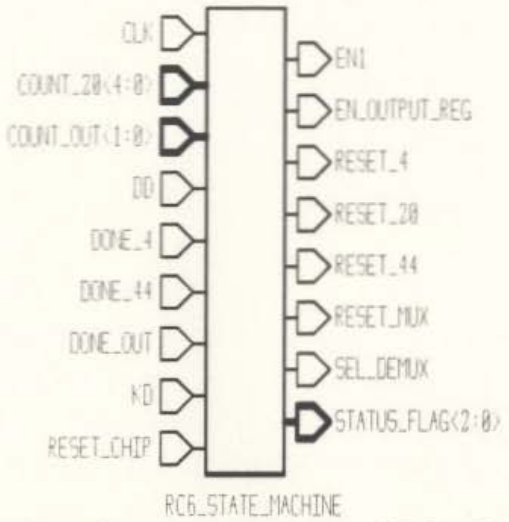

Figure 4.8: A Component Interface Representation of RC6 Global State Machine

4.4.2.2 RC6 Data Flow Controller

A simple controller has been incorporated in the design of the RC6 encryptor. The controller is needed to enable and disable a number of registers and multiplexers during the data encryption mode of the cipher. In other words, it is used to regulate the flow of data through the RC6 core and update the various registers after each round of encryption, thereby allowing a feedback connection from the output of the RC6 core to its input. As soon as the global state machine control unit forces the cipher to go into data-encrypt mode, the flow controller starts operating until the cipher exits this mode, once the data has been encrypted.

4.4.3 Key Storage for RC6

One of the major concerns in the design of the RC6 cipher has to do with storage of the forty-four 32-bit subkeys that are to be used during the data-encrypt mode of the cipher. Our design of the RC6 cipher assumes that the subkeys for encryption are being generated outside the FPGA using the *key schedule scheme* for the cipher. The details of RC6 cipher key schedule scheme are outlined in [3].

During the *key-download* mode of the RC6 cipher, the forty-four 32-bit subkeys are being downloaded into the *key storage unit* inside the cipher. Thus the forty four clock cycles needed to store the subkeys inside the cipher constitute the *key-setup time*. Our implementation of the key storage unit involves using a combination of serial-in parallel-out and parallel-in parallel-out registers. The former is used during the storage of forty four 32-bit words. During the data-encrypt mode, the forty four subkeys are being fed into the RC6 core using the parallel-in parallel-out register. This implementation of the key storage unit consumes a total of 704 configurable logic blocks (CLBs). This implies that the key storage unit takes up 10% of the available FPGA resources.

4.4.4 Simulation and Synthesis Results

This section presents the findings of our simulation and synthesis studies for the complete design of the RC6 cipher. As explained earlier, the cipher operates in five different modes - reset mode, key-download mode, idle mode, data-download mode, and finally the data-encrypt mode.

As the simulation runs for the design are very extensive, only the portions that concern each mode of global state machine operation are illustrated in Appendix B. When the cipher

is powered up, it is in the reset mode as indicated by the status flag ('7', decimal equivalent of "111"). Next, when the RESET-CHIP input to the cipher is disabled, the cipher moves into the key-download mode. During this mode of operation (status flag = '1', decimal equivalent of "001"), the forty four 32-bit subkeys are being downloaded. Once the forty four subkeys are downloaded, the status flag changes to a new value ('2', decimal equivalent of "010") and then the cipher enters into the idle mode as indicated by status flag ('0', decimal equivalent of "000"). Now depending upon the user-controlled KD and DD inputs, the cipher can move into any other mode of operation. As seen from the simulation figure, the cipher then enters the data-download mode (KD = '0', DD = '1' and status flag = '3'). During this mode of operation, the 128-bit data is downloaded into the cipher in four clock cycles as the cipher has 32-bit I/O buses. Once the 128-bit plaintext is downloaded (status flag = '4'), the cipher then enters the data-encrypt mode (status flag = '5'). After 20 rounds of encryption, the ciphertext data is latched out in four clock cycles (status flag = '6'). Unless directed otherwise, the cipher then goes back to download the data and encrypt it in a synchronous fashion.

As a verification of the design functionality, we adopt a bottom-up approach by testing each individual component and subcomponent thoroughly until we verify the correct operation of the entire design. Each component is also synthesized and a postsynthesis simulation is carried out to make sure that we generate the right hardware. In addition, we use test vectors such as previously encrypted plaintext-ciphertext pairs and encryption subkeys to verify the correct operation of the cipher. For instance, a plaintext B18C555EB18C555E2AA5D2AB2AA5D2AB (a 128-bit plaintext block represented in hexadecimal form) is loaded and encrypted with the forty-four 32-bit subkeys (FFEFFFFF...)

stored inside the cipher. The resulting ciphertext is C749B1640A9DB0EA12579B32F94CC59D.

These simulation results have been obtained after carrying out both functional simulation and the timing simulation after the actual place and route of the RC6 design in the target Xilinx XC40200 FPGA device. Here it should be noted that we are assuming that the subkeys for encryption are being generated outside the FPGA using the key schedule scheme for the cipher and that they are being downloaded into the key storage unit. Neglecting the *key-setup* time, a single encryption time T_{encr} is given as follows:

$$\begin{aligned} T_{encr} &= T_{data-in} + 20T_{clk} + T_{data-out} \\ \Rightarrow T_{encr} &= 4T_{clk} + 20T_{clk} + 4T_{clk} \\ \Rightarrow T_{encr} &= 28T_{clk} \end{aligned} \tag{4.1}$$

Here it should be noted that T_{clk} represents the minimum clock period and defines the maximum combinational path delay of the design. Moreover, since there are 32-bit I/O buses, as such it will take four clock cycles to download the 128-bit plaintext data and an equal number of clock cycles to latch out the data onto the 32-bit output bus. Our synthesis process results in the following numbers:

- Minimum clock period = 146 ns
- Maximum clock frequency = 6.85 MHz
- Time for a one encryption = T_{encr} = 4088 ns
- Rate of encryption = 31.3 Mbps

The hardware needed for the RC6 encryption is 4944 CLBs for the RC6 core plus 704 CLBs for key storage unit and another 64 CLBs for storing the 128-bit input data. Besides

this, there is also some data flow and control logic overhead. Thus the total FPGA resources required is about 6450 CLBs, which implies that the design takes up 91% of the available CLBs in the target device.

4.5 Design of CAST-256

CAST-256 is the other AES candidate that has been implemented in FPGAs in order to investigate its efficiency from the hardware implementation perspective. The cipher as realized in hardware is illustrated in Figure 4.9.

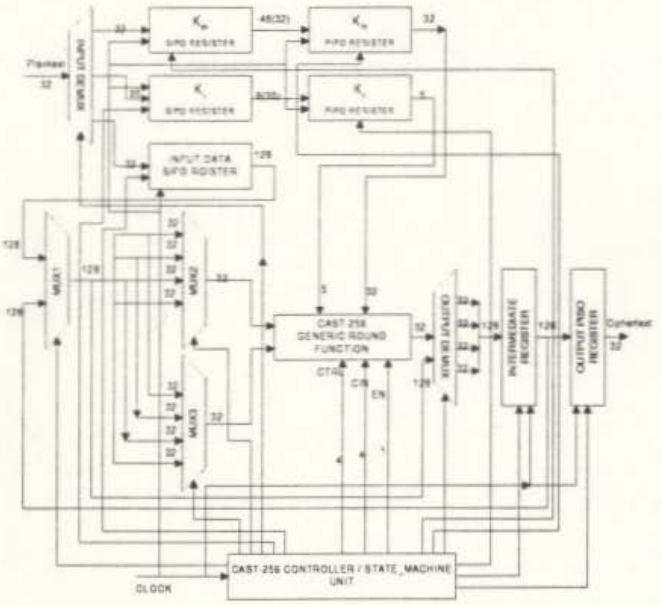

Figure 4.9: CAST-256 Encryption in Hardware

The design of the CAST-256 cipher is based on *single-stage iterative* architecture. As mentioned earlier, this design approach involves generating the hardware for one round of encryption and then designing a control path that allows a feedback connection from the output of the single round hardware to its input. As a consequence, we cycle the original plaintext input through the single stage hardware for the required number of rounds (or iterations), finally latching the encrypted data out at the end of the required number of encryption rounds.

4.5.1 CAST-256 Datapath

The CAST-256 datapath basically consists of a *generic round function*, a number of multiplexers, demultiplexers, a feedback register, a feedback multiplexer and a final output register. The generic round function module realizes any of the three round functions f_1, f_2, or f_3 depending on the particular round in progress.

4.5.1.1 Generic Round Function

The generic round function module consists of four 32-bit *add/subtract/exclusive-or* units, a separate 32-bit XOR module, a 32-bit barrel shifter and four 8×32 S-boxes, namely, S_1, S_2, S_3, and S_4. The generic round function module is shown in Figure 4.10.

The generic round function module receives two 32-bit inputs from the two 4×1 multiplexers, in addition to inputs from the *masking key storage* and *rotation key storage* units. The generic round function module also receives control inputs from the control unit for the cipher. The generic round function has only one 32-bit output, because any one of the four 32-bit blocks A, B, C, or D is modified at the end of each round of encryption. The 32-bit

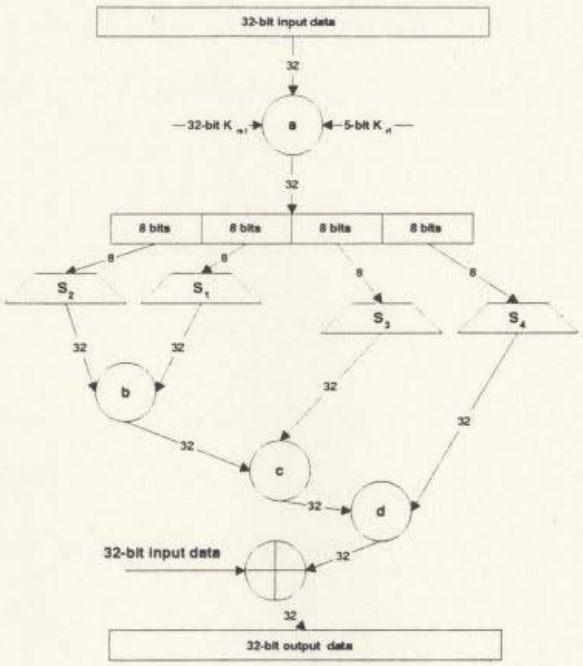

Figure 4.10: The Generic Round Function Module

input which is to be modified by the generic round function module as well as additional 32-bit input that is fed into the separate XOR unit as a final process inside the module are being selected by the two 4×1 multiplexers outside the round function module. These selections are dictated by the control signal that emanate from the control unit of the cipher.

4.5.1.2 The S-Box Design

As illustrated in the block diagram representation of the generic round function module in Figure 4.10, the 32-bit output coming out of the 32-bit barrel shifter is split into four 8-bit vectors used as four sets of 8-bit address lines for the four 8×32 S-boxes. The S-boxes are implemented as *lookup tables* (LUTs). The hardware complexity of the cipher is primarily

due to the complex structures of the S-boxes. The two-hundred and fifty-six 32-bit values stored in each of these S-boxes are tabulated in [4].

Our synthesis results for the 8×32 S-boxes are presented here:

- Total number of CLBs used for each S-box = 411
- Maximum delay of each S-box = 62 ns.

The shere size of the four S-boxes and the associated delay contributes to the hardware complexity and low speed of the generic round function module, which requires a total of 3037 CLBs (43% of the available FPGA resources), and has a maximum combinational path delay of 202 ns.

4.5.2 CAST-256 Control Path Design

The CAST-256 control path consists of a global state machine unit with asynchronous inputs, a data flow controller and some glue logic as is needed to integrate the control logic with the cipher datapath.

4.5.2.1 CAST-256 Global State Machine

As in the case of RC6, here too the state machine unit is based on a *Moore finite state machine* model [64]. The component interface of CAST-256 state machine is presented in Figure 4.11. The state machine unit design has a number of asynchronous inputs (RESET-CHIP, KD, and DD) in addition to other inputs. The state machine is a synchronous one. The CAST-256 state machine is different from the RC6 counterpart in the sense that the former incorporates one more *key download* state. As mentioned earlier, in the case of

CAST-256 we have two sets of keys - forty eight 5-bit *rotation keys* and forty eight more 32-bit *masking keys*. Hence, we are concerned with an additional *rotation-key-download* state. The six cipher modes - *reset*, *masking-key-download*, *rotation-key-download*, *idle*, *plaintext data-download*, and *data-encrypt*.

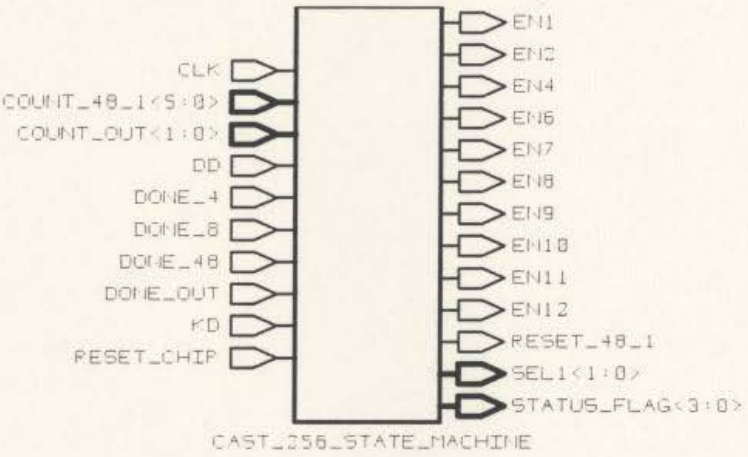

Figure 4.11: CAST-256 State Machine Unit

4.5.2.2 CAST-256 Data Flow Controller

The design of the data flow controller is of prime importance to the proper operation of the cipher, as it is responsible for regulating the flow of data through the CAST-256 core. A component interface for the controller is shown in Figure 4.12. The controller receives the counter output that is needed to stimulate certain control signals emanating from the control path and feeding into the datapath. The controller provides a synchronous control over the generic round function module. It also provides control inputs for the feedback multiplexer as well as for the two 4×1 multiplexers. The feedback multiplexer is used to loop back the data after each round of encryption. The other two multiplexers decide which 32-bit

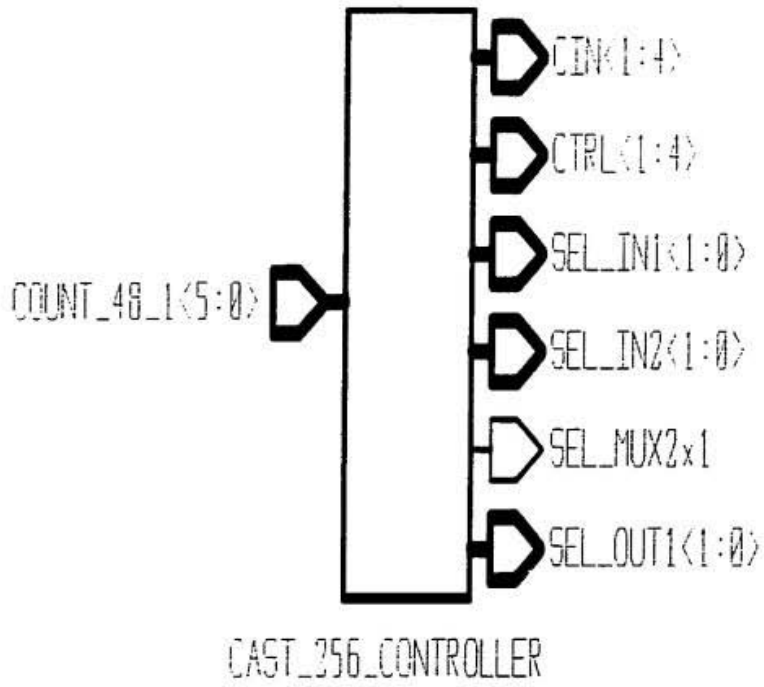

Figure 4.12: CAST-256 Data Flow Controller

input data is to be modified in each round. The controller also supplies four control inputs to the generic round function that decide which operation - addition, subtraction, XOR, or rotation operation - is to be executed for operations a, b, c, and d in the round function.

4.5.3 The Key Storage Unit

The key storage unit for the CAST-256 cipher is comprised of a masking keys storage part and a rotation keys storage portion. There are forty eight 32-bit masking keys stored inside the cipher to be fed to the generic round function. This is accomplished by designing a serial-in parallel-out (SIPO) register, in conjunction with a parallel-in parallel-out (PIPO)

register. We have a similar arrangement for storing the forty eight 5-bit rotation keys . Here, it should be noted that the subkeys for encryption are assumed to be generated outside the FPGA using the key schedule scheme for CAST-256 [4]. The keys are then downloaded into the key storage units during the *masking-key-download* and *round-key-download* modes. This implementation of storing the keys inside the FPGA uses around 1000 CLBs.

4.5.4 Simulation and Synthesis Results

This section presents the findings of our simulation and synthesis studies for the complete design of the CAST-256 cipher. As opposed to RC6, CAST-256 operates in six different modes - reset mode, masking-key-download mode, rotation-key-download mode, idle mode, data-download mode, and finally the data-encrypt mode.

As the simulation runs for the design are very extensive, so only the portions that concern each mode of global state machine operation are illustrated in Appendix C. When the cipher is powered up, it is in the reset mode as indicated by the status flag ('15', decimal equivalent of "1111"). Next, when the RESET-CHIP input to the cipher is disabled, the cipher moves into the masking-key-download mode. During this mode of operation (status flag = '1', decimal equivalent of "0001"), the forty four 32-bit masking subkeys are being downloaded. Once the forty four subkeys are downloaded, the status flag changes to a new value ('2', decimal equivalent of "0010") and then the cipher enters into the rotation-key-download mode as indicated by the status flag ('3', decimal equivalent of "0011"). Once the forty eight 5-bit round keys are downloaded (status flag = '4'), the cipher then enters the idle mode as indicated by status flag ('0').

Now depending upon the user-controlled KD and DD inputs, the cipher can move into

any other mode of operation. As seen from the simulation figure, the cipher then enters the data-download mode (KD = '0', DD = '1' and status flag = '5'). During this mode of operation, the 128-bit data is downloaded into the cipher in four clock cycles as the cipher has 32-bit I/O buses. Once the 128-bit plaintext is downloaded (status flag = '6'), the cipher then enters the data-encrypt mode (status flag = '7'). After 48 rounds of encryption, the ciphertext data is latched out in four clock cycles (status flag = '8'). Unless directed otherwise, the cipher then goes back to download the data and encrypt it in a synchronous fashion.

As a verification of the design functionality, we first encrypt a plaintext with a certain key and then later on use the same ciphertext as input to the cipher and recover the original plaintext. For the correct operation of the CAST-256 cipher, both the encryption and the following decryption should result in the original plaintext as long as the subkeys are fed in the reverse order. For instance, a plaintext E2AAFC11E2AAFC11E2AAFC11E2AAFC11 (a 128-bit plaintext block represented in hexadecimal form) is loaded and encrypted with the forty-eight 32-bit masking subkeys (FFEFFFFF...) and forty-eight 5-bit round subkeys stored inside the cipher. The resulting ciphertext is 3BA95C83135B95DFC54D1C13297FC027. At a later time, the ciphertext 3BA95C83135B95DFC54D1C13297FC027 is fed in as input to the cipher with the subkeys applied in the reverse order to recover the original plaintext E2AAFC11E2AAFC11E2A.... We also carry out presynthesis and postsynthesis testing of each individual component in a bottom-up manner to verify the correct operation of our design. Simulation results have been obtained after carrying out both functional simulation and the timing simulation after actually place and route of the CAST-256 design in the target Xilinx XC40200 FPGA device. Here it should be noted that we are assuming that the

subkeys for encryption are being generated outside the FPGA using the key schedule scheme for the cipher and that they are being downloaded into the key storage unit. Neglecting the *key-setup* time, a single encryption time T_{encr} is given as follows:

$$\begin{aligned} T_{encr} &= T_{data-in} + 48T_{clk} + T_{data-out} \\ \Rightarrow T_{encr} &= 4T_{clk} + 48T_{clk} + 4T_{clk} \\ \Rightarrow T_{encr} &= 56T_{clk} \end{aligned} \tag{4.2}$$

Here it should be noted that T_{clk} represents the minimum clock period and defines the maximum combinational path delay of the design. Moreover, since there are 32-bit I/O buses, as such it will take four clock cycles to download the 128-bit plaintext data and an equal number of clock cycles to latch out the data onto the 32-bit output bus. Our synthesis process results in the following numbers:

- Minimum clock period = 198 ns
- Maximum clock frequency = 5.05 MHz
- Time for a one encryption = T_{encr} = 11088 ns
- Rate of encryption = 11.54 Mbps

The time for one encryption and the corresponding encrypted data rates apply to a 48-round CAST-256 cipher. However, if we had half the number of rounds for CAST-256, i.e. twenty four rounds in all, the encryption speed of the cipher is almost doubled, as the constant 48 in the expression for T_{encr} changes to 24 and this leads to a data rate of 20.2 Mbps. There are no known cryptanalytic attacks that have been applied to a 24-round version of the cipher.

The hardware needed for the CAST-256 encryption is 3037 CLBs for the generic round function plus 768 CLBs for masking keys storage and another 120 CLBs for storing the rotation keys. Besides this, another 64 CLBs are required for storing the 128-bit input data. The data flow and control logic overhead amounts to around 1000 CLBs. Thus the total FPGA resources required is about 5050 CLBs, which implies that the design takes up 72% of the available CLBs in the target device.

4.6 Comparison of RC6 and CAST-256 Ciphers

Simulation and synthesis studies for the two ciphers suggest that neither RC6 nor CAST-256 is well suited for implementation in the targetted Xilinx FPGA. The hardware complexity is high and the encryption speed is low, particularly compared to similar implementations of DES [51].

Our simulation and synthesis studies reveal that multiplication in particular and addition to some extent are major bottlenecks as far as speed of encryption in RC6 cipher is concerned. This has also to do with custom architecture of the FPGAs. However, a faster implementation of RC6 cipher can only be achieved at the expense of increasingly large hardware complexity, which implies the use of a high-end FPGA device. Moreover, it appears that implementation of RC6 in the targetted FPGA using *pipelining* is found to be impractical from a hardware complexity viewpoint. This is because of the large number of CLBs (in excess of 6000 CLBs) that are needed for implementing just one round of encryption hardware. So, if we want to even pipeline say two rounds, we will be needing twice the hardware and will need to consider very high density devices.

CAST-256 encryption in FPGAs is found to be slower than what we can achieve with

the RC6 cipher. At the same time, the hardware complexity associated with CAST-256 cipher is roughly of the same order as RC6. This is because the advantage of not having a multiplication operation is being offset by the use of four large S-boxes.

4.6.1 Some Recent Modifications

Lately in our research, we have made a number of improvements in the design of some of the key cipher components that have improved the ovreall speed of these ciphers to some extent, as well as reduced some of the hardware that was previously in use. But, these modifications still do not mark a significant improvement over the speed and hardware complexity for these ciphers.

One major modification has to do with the storage of the encryption key, and the implementation of the S-boxes. We have now replaced the LUT implementation of the S-boxes with a much faster and low complexity RAM units. Another significant modification involves improving the speed of the multiplier by replacing the current 32-bit adder designs with a new kind of synthesized adders provided by the LOGIBLOX toolbox available with the new FPGA design tools. These are called " Relationally Placed Macros" or RPMs. These adders not only make use of the fast carry logic available within the Xilinx CLBs, but at the same time most of the logic is aligned in parallel to reduce the delays. This 32-bit adder implementation reduces the hardware by a factor of ten over the previously employed adder implementations. The speed of an RPM based 32-bit adder is around 20 ns as opposed to 39 ns for the hybrid adder design. However, it should be noted that these adder results are only fine for the SRAM-based Xilinx FPGAs, and for the technology independent implementation, the hybrid design appears to be the best approach.

Some recent modifications in the the design of control path for both the ciphers have resulted in removing the $T_{data-in} = 4T_{clk}$ overhead, so that during the last stages of encryption, data for next encryption is already available, thus saving us four clock cycles. This has increased the encryption data rate for our RC6 cipher implementation from 31 Mbps to around 37 Mbps. Similarly, the encryption data rate for CAST-256 has been improved to a value of 12.5 Mbps for 48-round implementation and 24 Mbps for a 24-round implementation.

If we compare the encryption speeds of these hardware implementations with the corresponding implementations in software on 200 MHz Pentium and Pentium Pro platforms, we find that rate of encryption for RC6 is around 100 Mbps and that for CAST-256 is 38.8 Mbps [65]. However, in order to attain very high speeds in hardware, one can go for a full custom ASIC implementation. These results also point out the fact that most of these AES candidate algorithms have an element of bias for software implementation. As such there is a need to look for a private-key block cipher that is very efficient in terms of hardware implementation.

4.7 Conclusion

In this chapter, we have presented the FPGA implementation of two AES candidates - RC6 and CAST-256 encryption algorithms. We have first explained the two encryption algorithms in detail. Next we have found it necessary to talk about the hardware development environment that has been used to implement the two ciphers. This is followed by a detailed investigation of the design of RC6 cipher. Here, the design of RC6 datapath as well as its control path is presented. The FPGA implementation of the cipher is carried out and simulation and synthesis results are presented. A similar investigation of CAST-256 cipher

design is presented, followed by simulation and synthesis results. Finally, the hardware complexities of the two ciphers are compared and certain conclusions are derived.

Chapter 5

A New Private-key Block Cipher Design

5.1 The Proposed Cipher

As mentioned earlier, the FPGA industry has become one of the fastest growing segments of today's industrial world. With the continuous enhancements in the FPGA technology in terms of increasing gate density and faster clock speeds, applications like reconfigurable computing, rapid prototyping, as well as *algorithm agile applications* are ideal for FPGA implementations.

However, the FPGA implementation of RC6 and CAST-256 encryption algorithms has brought forward a very important aspect of implementing the private-key block ciphers in hardware; the hardware complexities associated with these cipher designs are significant enough to discourage the possibility of going for programmable logic devices such as FPGAs. As a consequence, we propose a much simpler cipher design that makes use of simpler

operations that not only possess good *cryptographic* properties, but also make the overall cipher design efficient from the hardware implementation perspective. This approach may also encourage us to effectively pipeline multiple rounds of encryption and thereby increase the cipher speed for FPGA implementations in particular and hardware implementations in general.

The proposed cipher, which we shall refer to as *Fast Hardware Cipher* or FHC is a private-key 128-bit block cipher that is a generalization of basic Feistel network, and it incorporates the same data flow scheme as used in CAST-256. However, there are a number of key differences between the proposed cipher and CAST-256. The proposed cipher has a much simpler generic round function than the one used for CAST-256. Complex operations such as additions, subtractions, and data dependent rotations are removed. These have been replaced by simple XOR operations. As well, the mere size and structure of the S-boxes used in CAST-256 cipher was a major contributor towards the overall hardware complexity and low speed for the cipher. The *round function* in the proposed cipher makes use of eight 4×32 S-boxes instead of four 8×32 S-boxes. Hence, the size of each S-box is reduced from 256×32 bits to 16×32 bit.

The "penalty" for these simplifications is that the total number of encryption rounds is increased from 48 in CAST-256 to 96 in FHC. The reason for doing so has to do with the security of the cipher, as will be explained later in this chapter. A *key schedule scheme* must used to generate the 32-bit *masking* subkeys K_{mi}, each of which is used per round. There are no rotation subkeys.

The round function used in FHC is defined as follows:

- Round Function f

for $(i = 0;\ i < 12;\ i++)$

{

$C = C \oplus f(D, K_{m_{4i+1}})$

$B = B \oplus f(C, K_{m_{4i+2}})$

$A = A \oplus f(B, K_{m_{4i+3}})$

$D = D \oplus f(A, K_{m_{4i+4}})$

}

for $(i = 12;\ i < 24;\ i++)$

{

$D = D \oplus f(A, K_{m_{4i+1}})$

$A = A \oplus f(B, K_{m_{4i+2}})$

$B = B \oplus f(C, K_{m_{4i+3}})$

$C = C \oplus f(D, K_{m_{4i+4}})$

}

Figure 5.1: Encryption with Fast Hardware Cipher (FHC)

$$I = K_{m_i} \oplus D$$

$$O = (S_1[I_a] \oplus S_2[I_b] \oplus S_3[I_c] \oplus S_4[I_d] \oplus S_5[I_e] \oplus S_6[I_f] \oplus S_7[I_g] \oplus S_8[I_h])$$

where D is the 32-bit data input to the round function, I_a through I_h are the most significant nibble through the least significant nibble of I, respectively, S_i is the i^{th} substitution box, and O is the 32-bit output of the round function. Each S-box is a nonlinear mapping of a 4-bit input to a 32-bit output. In our implementation, these S-boxes have been randomly generated. Moreover, "$\oplus$" is bitwise exclusive-OR operation.

The proposed encryption algorithm is illustrated in Figure 5.1. The plaintext is stored in

four 32-bit registers, A, B, C, and D. In each round of encryption, a 32-bit masking key is used. The output of the 96-round encryption is also stored in the four 32-bit registers. The design of a suitable key scheduling algorithm is not addressed in this thesis. As in the case of the previous two cipher implementations, here too, the subkeys for encryption are generated outside the FPGA and downloaded into the cipher during the key-download mode.

5.2 FPGA Implementation of the Proposed Cipher

A block diagram of the proposed cipher as realized in digital hardware is presented in Figure 5.2. The cipher is constructed as a single-stage iterative structure. This implies that we have the hardware developed for one round of encryption and the control part of the cipher is responsible for iteratively passing the data through the encryption core.

5.2.1 Datapath

As seen from Figure 5.2, the datapath for the proposed cipher encapsulates the encryption core implemented as a round function unit. A round function consists of a multiple exclusive-OR operations and S-box substitutions. When the 32-bit data enters the round function unit, it first goes through a bitwise XOR with the 32-bit masking key for that particular round. The output of this operation is then divided into eight 4-bit vectors used as address lines for the eight 4×32 S-boxes. The 32-bit outputs of all eight S-boxes are then XORed to yield a 32-bit result. These final 32 bits are then XORed with appropriate 32-bit values of A, B, C, or D. This 32-bit output of the round function unit is then swapped along with the other three 32-bit values and then looped back to the input of the round function. This continues for a total of 96 rounds. The swapping of the 32-bit words and the feedback connection is

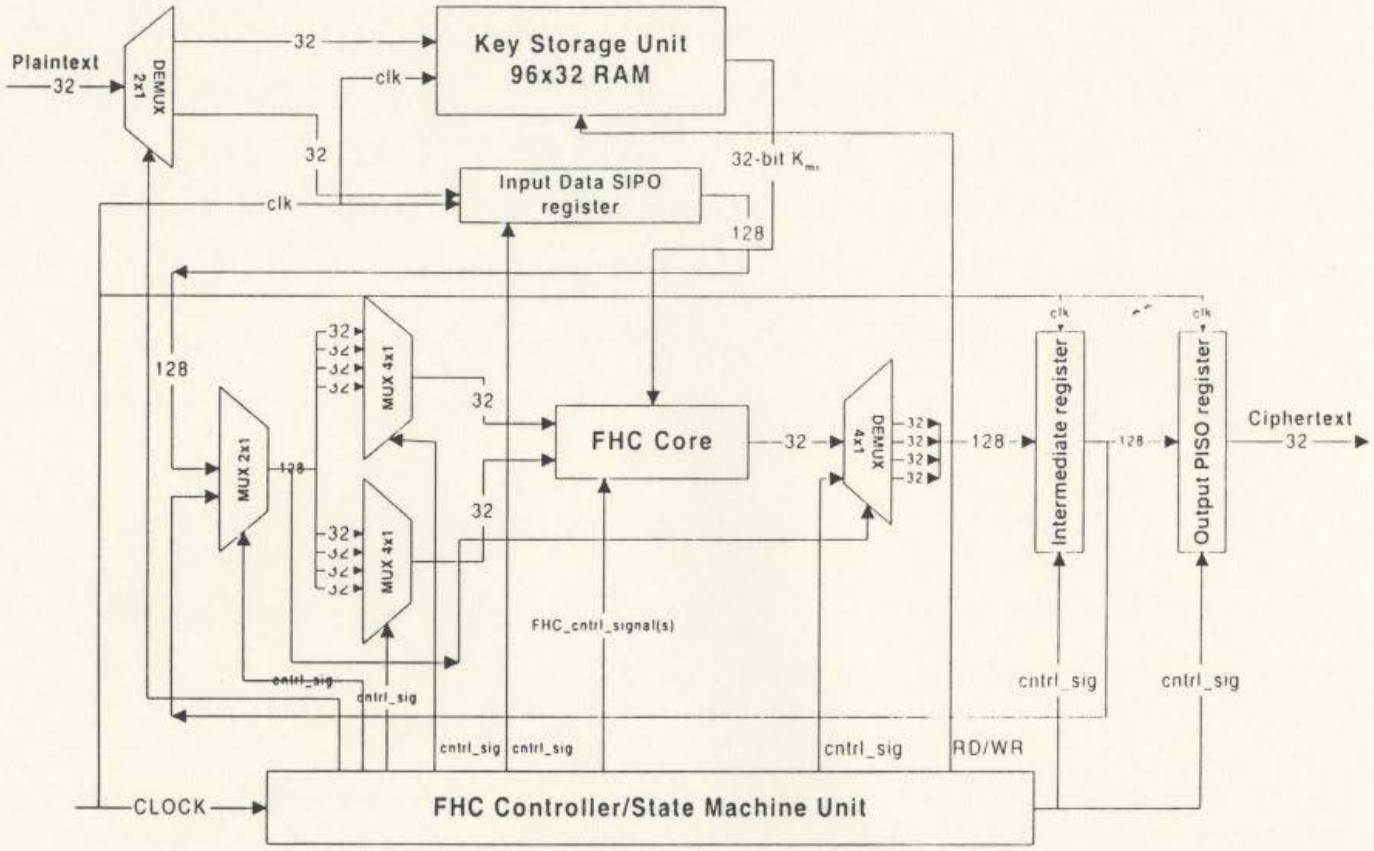

Figure 5.2: Realization of Fast Hardware Cipher in Hardware

achieved using a number of multiplexers, demultiplexers and a feedback register.

The simple structure for the round function lends itself nicely to an FPGA implementation. This is mainly because of employing simpler cryptographic operations such as bitwise XOR. The size of the S-boxes used in this design is reduced from the ones that were used to implement the CAST-256 cipher. This has resulted in saving a lot of hardware as well as reducing the critical path delay considerably. Synthesis of the round function for the proposed cipher yields a maximum combinational path delay of just 30 ns as opposed to 202 ns for the CAST-256 generic round function implementation. The total number of CLBs used for implementing this round function is just 192, which is a considerable improvement

over the CAST-256 generic round function that requires 3037 CLBs. This is mainly because now the total number of CLBs needed for the eight 4×32 S-boxes is $8 \times 16 = 128$ as opposed to about $4 \times 411 = 1644$. This considerable reduction in the hardware is also because the S-boxes for the proposed cipher have been implemented as 16×32 RAMs, instead of lookup table (LUT) implementations based on using the register bits in the CLBs. As such the RAM implementation is still a LUT - just more efficient storage.

5.2.2 Control Path Design

The FHC control path is not much different from the one employed for CAST-256, except for the total number of encryption rounds. The control path comprises of a synchronous state machine unit with asynchronous inputs, a data flow controller and some glue logic.

The state machine for the proposed cipher is no different from the one used for the FPGA implementation of CAST-256, except that in the case of FHC, we have only one *key-download* state. The data flow controller is also based on the same design as used for CAST-256 (see section 4.5.2.2).

5.2.3 The Key Storage Unit

One of the major enhancements to the implementaion of the proposed cipher has to do with the way the encryption keys are being stored inside the target FPGA. Previously, key storage module implementations for RC6 and CAST-256 employed the lookup tables (LUT) approach, but our synthesis studies for FPGA implementations suggest that table lookup implementations do not scale efficiently in custom architecture FPGAs, mainly because of the restricted nature of the CLBs.

On the other hand, implementing the key storage unit as RAMs proves to be very efficient, as the target device makes full use of the on-chip RAM available to it. For the proposed cipher, we need to store ninety six 32-bit encryption subkeys. These subkeys are to be read later on during the *data-encrypt* mode. Our implementation of the key storage unit for the proposed cipher gives the following numbers:

- Maximum combinational path delay = 26.593 ns
- Total number of CLBs used = 99

5.2.4 Simulation and Synthesis Results

The design of the complete proposed cipher has been simulated, synthesized and we place-and-routed in a particular FPGA. The relatively small hardware complexity associated with this design makes it possible for us to use a medium size FPGA device. We have used XILINX XC4036EX FPGA for implementing this design. The target device has been used in the field for quite some time now and has proven to be very popular. XC4036 has a total of 1296 CLBs.

As the simulation runs for the design are very extensive, only portions of simulation that concern each of the five cipher modes are shown (see Appendix D). These include the reset mode, key-download mode, idle mode, data-download mode, and the data-encrypt mode. The key-download mode differs from the one for RC6 because the former takes 96 clock cycles as opposed to 44 clock cycles for the latter. The manner in which the cipher cycles through different modes of operation is the same as described in section 4.4.4.

Neglecting the key setup time, one single encryption time T_{encr} is given as follows:

$$T_{encr} = 96T_{clk} + T_{data-out}$$

$$\Rightarrow T_{encr} = 96T_{clk} + 4T_{clk}$$

$$\Rightarrow T_{encr} = 100T_{clk} \quad (5.1)$$

Here it should be noted that T_{clk} represents the minimum clock period and defines the maximum combinational path delay of the design. Moreover, since there are 32-bit I/O buses, as such it will take four clock cycles to download the 128-bit plaintext data and an equal number of clock cycles to latch out the data onto the 32-bit output bus. However, in this particular implementation, we have made a modification so that during the time data is being encrypted, the next 128-bit plaintext is already available. This has saved us four clock cycles for downloading the input data. Our synthesis process results in the following numbers:

- Minimum clock period = 30 ns
- Maximum clock frequency = 33.33 MHz
- Time for a one encryption = T_{encr} = 3000 ns
- Rate of encryption = 42.67 Mbps

The time for one encryption and the corresponding encrypted data rates apply to a 96-round FHC implementation.

The hardware needed for the FHC encryption is 192 CLBs for the generic round function plus 99 CLBs for "masking" keys storage . Besides this, another 64 CLBs for storing the 128-bit input data and 66 CLBs for each of the eight S-boxes. The data flow and control logic overhead amounts to under 300 CLBs. Thus the total FPGA resources required is about 750 CLBs, which implies that the design takes up 57% of the available CLBs in the

target device, but only 11% of the available CLBs in the device targetted for the original RC6 and CAST-256 designs!

5.3 Security Analysis

In proposing any new cipher, one of the key design elements concerns the security of the proposed cipher. A successful cipher should resist all proposed cryptanalysis techniques, such as *linear* and *differential* cryptanalysis, and at the same time exhibit the potential of surviving brute force attacks in future when computing power is increased. Moreover, there should exist clear mathematical techniques to analyze and determine the cryptographic strength of the cipher in question.

In this section, we analyze the security of our proposed cipher against two very potent cryptanalytical attacks, namely *linear* and *differential* cryptanalysis. An $m \times n$ S-box is a $2^m \times n$ lookup table. In SPNs and most Feistel ciphers, S-boxes are critically important to security, since they are the key components of nonlinearity in the algorithm. As the size of the lookup table increases exponentially with increase of the value m, m should be chosen to be small. On the other hand, the value of n can be selected to be large as the size of lookup table increases linearly with n.

5.3.1 Selecting Nonlinear Round Functions

For the proposed cipher, we have randomly selected eight 4×32 S-boxes. These S-boxes have been constructed using random number generators. It is therefore important to consider the nonlinearity contributed to the cipher by randomly selected S-boxes. The concept of nonlinearity and m-bit affine functions have been defined in Section 2.2.1. It follows that

out of all 2^{2^m} possible m-bit functions, there are 2^{m+1} affine functions. Since there are thirty two 4-bit linear or affine functions, the probability of randomly generating a 4-bit linear or affine function is given as $p_{lin} = (2^5/2^{16}) = 2^{-11}$.

Next consider the problem of selecting $k = 0, 1, 2, ..., 8$ perfectly linear S-box approximations, wherein we consider the approximation that is an XOR of some subset of output bits of a round function and is the XOR of corresponding bits in the outputs of S-boxes. Now the probability of k out of a total of eight S-boxes being nonlinear for a particular subset of output bits, P_k, is given as a binomial distribution:

$$P_k = \begin{pmatrix} 8 \\ k \end{pmatrix} (1 - p_{lin})^k (p_{lin})^{8-k} \tag{5.2}$$

Table 5.1 lists values of P_k for $k = 0, 1, 2, ..., 8$. This implies, for example, the probability of randomly selecting all eight S-boxes contributions to the round approximations as being linear is 2^{-88} for a particular subset of round function output bits. We shall use these results in our following discussion of the security of the cipher with respect to linear cryptanalysis.

P_k	2^{-88}	2^{-76}	2^{-59}	2^{-46}	2^{-38}	2^{-28}	2^{-18}	2^{-8}	0.996
k	0	1	2	3	4	5	6	7	8

Table 5.1: Probabilities of selecting k nonlinear S-boxes

5.3.2 Linear Cryptanalysis

Linear Cryptanalysis [7] attempts to find a linear approximation of a cipher only derived from plaintext, ciphertext, and key terms. A general linear approximation of a cipher is derived by combining a number of linear approximations of the S-boxes of different rounds

so that the intermediate terms are canceled. Basically, the attack makes use of any high probability occurrences of linear expressions of input, output, and round keys in the round function of an iterated cipher structure. As such the basic principle of linear cryptanalysis is to determine a linear approximation of the type:

$$P_{j_1} \oplus P_{j_2} \oplus .... \oplus P_{j_a} \oplus C_{k_1} \oplus C_{k_2} \oplus ..... \oplus C_{k_b} = K_{l_1} \oplus K_{l_2} \oplus .... \oplus K_{l_c} \tag{5.3}$$

where $j_1, j_2, ...j_a, k_1, k_2, ...k_b$, and $l_1, l_2, ...l_c$ denote bit positions of the plaintext P, ciphertext C, and key K, respectively. In [34], the probability of satisfying the best linear expression for r-round cipher is bounded as follows:

$$\left|p_l - \frac{1}{2}\right| \leq 2^{\alpha-1} . \left|p_\beta - \frac{1}{2}\right|^\alpha \tag{5.4}$$

where p_l represents the probability that the linear expression 5.3 holds, p_β represents the probability of the best linear approximation of any S-box. Also, α is the number of S-boxes involved in the linear approximation. Ideally $p_\beta = 0.5 \Rightarrow p_l = 0.5$.

It has been shown in [34] that an S-box linear approximation has a probability p_β where

$$\left|p_\beta - \frac{1}{2}\right| = \frac{2^{m-1} - NL}{2^m} \tag{5.5}$$

where m represents the total number of input bits to the S-box, and NL is the nonlinearity of the S-box function used in the linear approximation. Here $NL = 0$ implies a linear function.

A linear cryptanalytical attack typically employs a number of linear approximations of the rounds to develop an overall linear expression involving subsets of plaintext and ciphertext bits. This makes it possible then to derive one key bit, which in turn is given as the exclusive-or of a number of round key bits as in given in equation 5.3. As a result, it has

been shown in [7] that the number of known plaintexts required, with a success rate of 97.7%, is approximately

$$N_l = \left| p_l - \frac{1}{2} \right|^{-2} \tag{5.6}$$

In order to analyze the strength of the proposed cipher against this attack, we adopt a very pessimistic approach by making worst-case assumptions. From Table 5.1, it is clear that the probability of selecting all eight S-boxes to be linear (2^{-88}) for a particular set of output bits is highly remote. As such we can very safely reject the possibility of all eight S-boxes being perfectly linear in the linear approximation of a round.

Let us consider the scenario where we have seven linear S-boxes and one nonlinear S-box. Once again adopting a very conservative approach, assume $NL = 1$. Substituting this value for NL in equation 5.5, we get a $\left| p_\beta - \frac{1}{2} \right| = \frac{7}{16}$. What follows next is an example of how to construct the best linear approximation for the round function and then extending it to the entire cipher.

Let us assume that the best linear approximation for the round function in round 1 is as follows:

$$\begin{aligned} Round\,1 : X_{3^1} &= P_{99} \oplus K_{3^1} \\ X_{3^1} &= Y_{2^1} \oplus Y_{3^1} \\ K_{3^1} &= P_{99} \oplus Z_2 \oplus Z_3 \end{aligned} \tag{5.7}$$

where equation 5.7 holds with $p_\beta = \frac{7}{16}$. Here X_{ij} and Y_{ij} represent the i^{th} input bit to the j^{th} S-box and i^{th} output bit to S-box S_j, respectively, and K_{ij} represents the i^{th} bit of j^{th}

round key. Z_k is the k^{th} output bit of the round function. Similarly, scenarios for the best linear approximations for second, third, and fourth rounds are given below. All these best linear approximations are assumed to hold with $p_\beta = \frac{7}{16}$.

$$\begin{aligned} \textit{Round}\,2 : Z_2 \oplus Z_3 \oplus K_{2^2} \oplus K_{3^2} &= X_{12^1} \oplus X_{10^1} \\ X_{2^2} \oplus X_{3^2} &= Y_{12^1} \oplus Y_{10^1} \\ Z_2 \oplus Z_3 \oplus K_{2^2} \oplus K_{3^2} &= Z_{12} \oplus Z_{10} \end{aligned} \tag{5.8}$$

$$\begin{aligned} \textit{Round}\,3 : Z_{10} \oplus Z_{12} \oplus K_{10^3} \oplus K_{12^3} &= X_{10^3} \oplus X_{12^3} \\ X_{10^3} \oplus X_{12^3} &= Y_{2^3} \\ Y_{2^3} &= Z_2 \\ Z_{10} \oplus Z_{12} \oplus K_{10^3} \oplus K_{12^3} &= Z_2 \end{aligned} \tag{5.9}$$

$$\begin{aligned} \textit{Round}\,4 : X_{2^1} &= Z_2 \oplus K_{2^4} \\ X_{2^1} &= Y_{3^1} \oplus Y_{5^1} \oplus Y_{7^1} \\ Y_{3^1} \oplus Y_{5^1} \oplus Y_{7^1} &= Z_3 \oplus Z_5 \oplus Z_7 \\ Z_3 \oplus Z_5 \oplus Z_7 &= C_{99} \oplus C_{101} \oplus C_{103} \\ Z_2 \oplus K_{2^4} &= C_{99} \oplus C_{101} \oplus C_{103} \end{aligned} \tag{5.10}$$

Finally, the best linear approximation for a 4-round cipher is as follows:

$$P_{99} \oplus C_{99} \oplus C_{101} \oplus C_{103} = K_{3^1} \oplus K_{2^2} \oplus K_{3^2} \oplus K_{10^3} \oplus K_{12^3} \oplus K_{3^4} \tag{5.11}$$

Now substituting the values for a 4-round linear approximation in equations 5.4 and 5.6, we find that the total number of known plaintexts needed to guess the right hand side of

equation 5.3 with a 97.7 % success rate is only 12. However, if we extend this attack to complete 96 rounds of the cipher, then we need at least 2^{39} known plaintexts for 96 rounds. Although this number is too small to claim 96 rounds for the cipher to be secure, however, the very low probability of selecting seven linear S-boxes (see Table 5.1) implies that being able to represent seven out of eight S-boxes in each round as linear in a linear approximation of a round function is extremely unlikely.

Table 5.2 lists the results for 4-round, 48-round, and 96-round linear approximations for different values of NL. We find that for all S-boxes functions used in the approximation with nonlinearities greater than 4, the attack becomes impractical for a 48-round cipher. The values listed in Table 5.2 also imply that for a 96-round cipher, the attack is not successful for $NL > 2$. In fact, for value of $NL = 3$, the number of known plaintexts required simply exceeds the total number of plaintexts (i.e. 2^{128}).

NL_{min}	$N_l(r = 4)$	$N_l(r = 48)$	$N_l(r = 96)$
1	$> 2^3$	2^{20}	$> 2^{39}$
2	$> 2^5$	2^{41}	$> 2^{82}$
3	$> 2^7$	2^{67}	$> 2^{132}$
4	2^{10}	2^{98}	$> 2^{194}$
5	$> 2^{13}$	2^{138}	$> 2^{274}$
6	2^{18}	2^{194}	–
7	2^{26}	2^{290}	–

Table 5.2: Linear Cryptanalysis Results for Different values of NL

We have so far focused on the upper region of Table 5.1, and have found that even by

adopting very conservative values of $NL = 1$, the cipher seems to be pretty secure against the linear attack. Next, we concentrate on the lower region of Table 5.1, wherein the probabilities of selecting 3 or 4 linear S-boxes are relatively not that remote.

Our analysis so far involves only one linear S-box approximation per round function, the probability of this occurring is extremely small, $p_1 = 2^{-76}$. Now we relax our assumptions a bit to assume that there are four out of eight S-boxes functions which are perfectly linear in our linear approximation of a round function and the probability of this happening being still a very unlikely, $p_4 = 2^{-38}$, for randomly selected S-boxes. This implies that we are to construct a linear approximation per round that takes into account four nonlinear S-boxes approximations and then concatenating the linear approximation for each round to form a 4-round linear approximation. Finally, we extend it to 48 and 96 rounds.

Assuming a $NL = 1$ for four S-boxes functions, say S_1, S_2, S_3, and S_4. This implies $\left|p_\beta - \frac{1}{2}\right| = \frac{7}{16}$. Our analysis for this scenario yields the following results:

- Total number of known plaintexts required for a 4-round attack is greater than 2^8
- Total number of known plaintexts required for a 48-round attack is approximately 2^{76}
- Total number of known plaintexts required for a 96-round attack is approximately 2^{150}

Similarly, we apply the same attack on the proposed cipher with the condition that there are three perfectly linear S-box approximations selected with a greater probability of $p_\beta = 2^{-28}$. This implies that we have to consider the effect of five nonlinear S-box approximations while constructing the linear approximation per round. Once again assuming the lowest value of nonlinearity ($NL = 1$), which corresponds to weakly nonlinear approximation, we get the following results:

- Total number of known plaintexts required for a 4-round attack is greater than 2^9
- Total number of known plaintexts required for a 48-round attack is approximately 2^{94}
- Total number of known plaintexts required for a 96-round attack is approximately 2^{187}

Our analysis of the proposed cipher against linear cryptanalysis under some very pessimistic assumptions suggest:

- The probability of a linear approximation involving five or more linear approximations of S-box functions is negligibly small.
- For a 96 round cipher with three or four linear S-boxes, the total number of known plaintexts required exceeds the the total number of available plaintexts. This renders the linear attack unsuccessful against the cipher.
- Finally, the known plaintexts requirement makes the attack increasingly impractical if we select S-boxes in a way such that fewer than three S-boxes are perfectly linear among them, although such cases are likely to occur.

Here it should be noted that, for our analysis, we had a very conservative approach and used worst-case bounds on the values of nonlinearity, because the probabilities of lower values of NL are low. It is conceivable that a linear approximation can be constructed that involves only one round function every four rounds, but this can be avoided if the S-boxes are selected so that the S-box functions are balanced (i.e. there are an equal number of ones and zeroes). We can safely conclude that the 96-round proposed cipher implementation is secure against linear cryptanalysis.

5.3.3 Differential Cryptanalysis

In this section, we briefly present some of the studies and results in relation to the proposed cipher's resistance to differential cryptanalysis.

Differential cryptanalysis [6] is basically a *chosen plaintext* attack. This method takes into account ciphertext pairs, whose corresponding plaintexts have a particular difference. In other words, it looks at the XOR difference of two plaintexts and compares that to the corresponding ciphertext pair. In a particular S-box, if we know the input XOR of a pair, it does not ensure the knowledge of its output XOR. However, there exists a probabilistic relation between the output XORs and every input XOR. Differential cryptanalysis makes use of the highly probable occurrences of sequences of output XOR differences at each round given a particular plaintext XOR difference.

A block cipher can be proved to resist differential cryptanalysis if it can be shown that high probability differentials do not exist. In a secure cipher, this probability should approach 2^{-N}, where N represents the block size. In the case of the proposed cipher, $N = 128$. In actual practice, its very hard to derive the probability of any practical differential. As a consequence, we search for highly probable r-round iterative characteristics. The probabilities of the most likely characteristics can be estimated and used as a measure of the cipher's resistance to differential cryptanalysis.

In [66], it has been shown that the best 4-round iterative characteristic for a round function having 4×32 S-boxes as used in the design of the proposed cipher has a differential probability of 2^{-12}. In [34], the probability of best r-round iterated characteristic is given as follows:

$$p_{\Omega_r} = \prod_{i=1}^{\frac{r}{4}} p_i \tag{5.12}$$

where p_i is the probability of the output XOR given the input XOR in round i. It has been shown in [34] that the number of chosen plaintexts is $N_C \simeq \frac{1}{p_{\Omega_r}}$ for an appropriate value of r (usually less than the number of rounds). Applying a similar approach for our proposed cipher, p_{Ω_r} is given as follows:

$$p_{\Omega_r} \leq (2^{-12})^{r/4} \tag{5.13}$$

In particular, an 88-round characteristic (used to mount a potential differential attack against the 96 round cipher) has a probability less than or equal to 2^{-264}. As a consequence, the number of chosen plaintexts needed for this attack would be in excess of 2^{264} for a 96-round implementation. These results suggest the proposed cipher appears to be quite immune to this kind of attack.

5.4 Conclusion

In this chapter, we propose a new private-key block cipher. We have discussed the design and implementation of the proposed cipher in FPGAs and outlined our results. We have also investigated the security of the cipher against two very popular and effective cryptanalytical attacks, namely, linear and differential cryptanalysis. Our analysis suggests that the proposed cipher appears to be quite secure against both the attacks.

Chapter 6

Conclusions and Future Work

The AES process, which commenced in 1997 by the National Institute of Standards and Technology (NIST) is a major unfolding in the field of private-key cryptography. As a consequence of this concerted effort, we will select a new block cipher as an eventual replacement for DES, which is nearing the end of its useful life. Interestingly, this event has come at the advent of the new millennium. CAST-256 and RC6 are among the stronger candidates to qualify as AES.

In this thesis, we have presented the hardware implementation of these two candidate encryption algorithms, bringing forth some very interesting observations and results. These conclusions constitute a framework for designing private-key block ciphers that are targetted for a hardware environment such as the FPGAs. As a consequence of our research, we have also proposed a new private-key block cipher which is very conducive for hardware implementations, especially for custom architectures such as the Field Programmable Gate Arrays (FPGAs).

6.1 Summary of the Thesis

In this thesis, we have investigated the issues relating to the hardware implementation of private-key block ciphers. In particular, we have selected field programmable gate arrays (FPGAs) as our target environment for implementing these ciphers in hardware. Two key factors have motivated us to go for FPGA implementations. Firstly, FPGAs possess this very attractive feature of reprogrammibility that has forced many applications to migrate from ASICs to the domain of FPGAs. The other key factors that play in the favour of FPGAs include rapid prototyping, that leads to faster design turnaround times, scalable architectures, and variable architectural parameters.

We have first presented the design of the RC6 cipher and its implementation in FPGAs. The cipher design is based on a single-stage hardware iterative architecture. This design approach requires the hardware for only one round of encryption, and the control machinery ensures that the data cycles through the hardware after each round of encryption. Our simulation and synthesis studies suggest that the inclusion of multiplication operation in the quadratic function of the RC6 core is a major bottleneck as far as the cipher encryption speed is concerned. However, a faster implementation of RC6 can only be achieved at the expense of increasingly large hardware complexity, which implies the use of a high end FPGA device. Implementation of RC6 in the target FPGA device using *pipelining* is impractical from a hardware complexity viewpoint.

CAST-256 encryption in FPGAs is found to be even slower than what we can achieve with RC6. At the same time, the hardware complexity of CAST-256 is roughly of the same order as RC6. This is because the advantage of not having a multiplication operation is offset by using four 8×32 S-boxes.

As a consequence of investigating the FPGA implementations of these two private-key block ciphers, we have proposed a much simpler cipher design that makes use of simpler cryptographic operations that not only possess good cryptographic properties, but also makes the overall cipher design efficient from the hardware implementation perspective. The new proposed cipher uses smaller S-boxes and does not incorporate any arithmetic operations such as addition or multiplication. This cipher design uses only 750 CLBs as opposed to 5050 CLBs for CAST-256, which is a reduction in hardware complexity by a factor of around 7. Also, the speed of the proposed cipher is improved by a factor of 3.5 over the CAST-256 implementation, if we are looking for a 96 round implementation of the proposed cipher.

One key observation has been made with regards to the hardware complexity of the proposed cipher. We have found that although we have been successful in bringing down the hardware complexity to a mere 192 CLBs for the round function, the hardware associated with the control circuitry as well as the key storage unit proves to be the overhead. This puts a lower bound on the hardware complexity that can be achieved. Another performance limiting factor is the maximum clock frequency of the target device. Most of the current FPGAs have a frequency ceiling of 50 MHz. However, with the introduction of the one million gate, 200 MHz FPGA chips, one can achieve very high data rates in the near future.

In this thesis, we have also investigated the security of the proposed cipher against linear and differential cryptanalysis. Our analysis suggests that the new cipher appears to be resistant to both kind of attacks.

6.2 Suggestions for Future Work

Our FPGA implementation of the proposed cipher suggests that since the hardware associated with its design is very small as compared to what we have come to know about RC6 and CAST-256, we can pipeline the new cipher to further increase the data encryption rate. It would be possible to pipeline four stages of the proposed cipher, giving us a speedup of about four over the present implementation. This implies that we can be looking for data encryption rates in excess of 150 Mbps for a 96-round implementation. This will be achieved at the expense of under 3000 CLBs and will therefore comfortably fit in the FPGA targetted for RC6 and CAST-256 ciphers.

Throughout the course of this research, we have not considered the design of key schedule scheme for any of the ciphers in question. This issue can also be investigated in the future. Generating the encryption keys on the fly is another approach that can be explored. A secure and efficient key schedule algorithm should be proposed for the new cipher as well. Yet another future direction would be to investigate the hardware implementation of other AES candidates such as MARS, RIJNDAEL, and TWOFISH [1]. Further research in this dynamic area is strongly encouraged.

Bibliography

[1] "http://csrs.nist.gov/encryption/aes/aes-home.htm." NIST Advanced Encryption Standard (AES) Development Effort Web Site.

[2] "National Bureau of Standards - Data Encryption Standard." FIPS Publication 46, 1977.

[3] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher." available at web site, " http://theory.lcs.mit.edu/rivest/rc6.pdf".

[4] C. Adams, "The CAST-256 Encryption Algorithm." available at web site, " http://www.entrust.com/resour ces/pdf/cast256.pdf".

[5] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, vol. 288(5), pp. 15–23, May 1973.

[6] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Journal of Cryptology*, vol. 4, no. 1, pp. 3–72, 1991.

[7] M. Matsui, "Linear Cryptanalysis Method for DES Cipher," in *Proceedings of EUROCRYPT'93*, pp. 386–397, Springer-Verlag, 1993.

[8] R. L. Rivest, "The RC5 Encryption Algorithm," in *Proceedings of Fast Software Encryption - 2nd International Workshop*, (Leuven, Belgium), pp. 86–96, Springer-Verlag, 1995.

[9] C. Adams, "Constructing Symmetric Ciphers Using the CAST Design Procedure," *Designs, Codes, and Cryptography*, vol. 12, no. 3, pp. 283–316, 1997.

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

[11] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell Systems Technical Journal*, vol. 28, pp. 656–715, 1949.

[12] W. A. Notz, H. Feistel, and J. L. Smith, "Some Cryptographic Techniques for Machine-to-Machine Data Communications," in *Proceedings of the IEEE*, vol. 63(11), pp. 1545–1554, November 1975.

[13] B. Schneier, "The Blowfish Encryption Algorithm," in *Proceedings of the Cambridge Security Workshop on Fast Software Encryption*, pp. 191–204, December 1993.

[14] A. Shimizu and S. Miyaguchi, "Fast Data Encipherment Algorithm FEAL," in *Proceedings of EUROCRYPT'87*, pp. 267–278, Springer-Verlag, 1987.

[15] E. Beham and A. Shamir, "Differential Cryptanalysis of FEAL and N-Hash," in *Proceedings of EUROCRYPT'91*, pp. 1–16, Springer-Verlag, 1991.

[16] K. Ohta and K. Aoki, "Linear Cryptanalysis of Fast Data Encipherment Algorithm," in *Proceedings of CRYPTO'94*, pp. 12–16, Springer-Verlag, 1994.

[17] X. Lai, J. L. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," in *Proceedings of EUROCRYPT'91*, pp. 17–38, 1991.

[18] B. S. K. Jr. and Y. L. Yin, "On Differential and Linear Cryptanalysis of the RC5 Encryption Algorithm," in *Proceedings of CRYPTO'95*, pp. 171–184, Springer-Verlag, 1995.

[19] H. M. Heys, "A Timing Attack on RC5," in *Proceedings of SAC'98*, (Kingston, Ont.), August 1998.

[20] J. Lee, H. M. Heys, and S. E. Tavares, "Resistance of a CAST-Like Encryption Algorithm to Linear and Differential Cryptanalysis," *Designs, Codes, and Cryptography*, vol. 12, no. 3, pp. 267–282, 1997.

[21] H. M. Heys and S. E. Tavares, "Substitution-Permutation Networks Resistant to Differential and Linear Cryptanalysis," *Journal of Cryptology*, vol. 9, pp. 1–19, 1996.

[22] J. B. Kam and G. I. Davida, " A Structured Design of Substitution-Permutation Encryption Networks," *IEEE Transactions on Computers*, vol. 28, no. 10, pp. 747–753, 1979.

[23] L. Brown and J. R. Seberry, "On the Design of Permutation P in DES Type Cryptosystems," in *Proceedings of EUROCRYPT'89*, pp. 696–705, 1989.

[24] A. F. Webster and S. E. Tavares, "On the Design of S-Boxes," in *Proceedings of CRYPTO'85*, pp. 523–534, Springer-Verlag, 1985.

[25] R. Forré, "The Strict Avalanche Criterion: Special Properties of Boolean Functions and an Extended Definition," in *Proceedings of CRYPTO'88*, pp. 450–468, Springer-Verlag, 1990.

[26] C. M. Adams, "A Formal and Practical Design Procedure for Substitution-Permutation Network Cryptosystems." PhD thesis, Queen's University at Kingston, Kingston, Ont., 1990.

[27] B. Preneel, W. V. Leekwijck, L. V. Linden, R. Goevarts, and J. Vanderwalle, "Propagation of Boolean Functions," in *Proceedings of EUROCRYPT'90*, pp. 161–173, Springer-Verlag, 1991.

[28] M. H. Dawson and S. E. Tavares, "An Expanded Set of S-Box Design Criteria Based on Information Theory and its Relation to Differential-Like Attacks," in *Proceedings of EUROCRYPT'91*, pp. 352–367, Springer-Verlag, 1991.

[29] R. Forré, "Methods and Instruments for Designing S-Boxes," *Journal of Cryptology*, vol. 2, no. 3, pp. 115–130, 1990.

[30] C. Adams and S. E. Tavares, "The Structured Design of Cryptographically Good S-Boxes," *Journal of Cryptology*, vol. 3, no. 1, pp. 24–41, 1990.

[31] L. O'Connor, "An Analysis of Product Ciphers Based on the Properties of Boolean Functions" PhD thesis, University of Waterloo, Waterloo, Ont., 1992.

[32] C. M. Adams and S. E. Tavares, "Designing S-Boxes for Ciphers Resistant to Differential Cryptanalysis," in *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, (Rome, Italy), pp. 181–190 , 1993.

[33] C. M. Adams, "Designing DES-Like Ciphers with Guaranteed Resistance to Differential and Linear Attacks," in *Proceedings of SAC'95*, (Carleton University, Ottawa, Ont.), pp. 133–144, 1995.

[34] C. Adams, H. M. Heys, S. E. Tavares, and M. Wiener, "An Analysis of CAST-256 Cipher," in *Proceedings of CCECE'99*, pp. 361–366, May 1999.

[35] P. Chow, S. O. Seo, D. Au, T. Choy, B. Fallah, D. Lewis, C. Li, and J. Rose, "A 1.2 μ CMOS FPGA using Cascaded Logic Blocks," in *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, 1991.

[36] C. Ebeling, G. Borriello, S. A. Hauck, D. Song, and E. A. Walkup, "TRYPTYCH: A New FPGA Architecture," in *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, 1991.

[37] "http://www.actel.com." , Actel web site.

[38] D. G. Reinertsen, "Whodunit? The Search for the New-product Killers," *Electronic Business*, July 1983.

[39] W. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A User Programmable Reconfigurable Gate Array," in *IEEE 1986 Custom Integrated Circuits Conference*, 1986.

[40] H. C. Hsieh, K. Duong, J. Y. Ja, R. Kanazawa, L. T. Ngo, L. G. Tinkey, W. S. Carter, and R. H. Freeman, "A Second Generation User Programmable Gate Array," in *IEEE 1987 Custom Integrated Circuits Conference*, 1987.

[41] H. C. Hsieh, K. Duong, J. Y. Ja, R. Kanazawa, L. T. . Ngo, L. G. Tinkey, W. S. Carter, and R. H. Freeman, "A 9000-Gate User Programmable Gate Array," in *IEEE 1988 Custom Integrated Circuits Conference*, 1988.

[42] T. Kean, "Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation." PhD thesis, University of Edinburgh,, 1989.

[43] F. Furtek, G. Stone, and I. Jones, "Labyrinth: A Homogeneous Computational Medium," in *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

[44] K. Kawana, H. Keida, M. Sakamoto, K. Shibata, and I. Moriyama, "An Efficient Logic Block Interconnect Architecture for User Reprogrammable Gate Array," in *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

[45] S. Hauck, G. Borriello, S. Burns, and C. Ebeling, "Montage: An FPGA for Synchronous and Asynchronous Circuits," in *Proceedings of the 2nd International Workshop on Field Programmable Logic and Applications*, 1992.

[46] R. Cliff, B. Ahanin, L. T. Cope, F. Heile, R. Ho, J. Huang, C. Lytle, S. Mashruwala, B. Pedersen, R. Raman, S. Reddy, V. Singhal, C. K. Sung, K. Veenstra, and A. Gupta, "A Dual Granularity and Globally Interconnected Architecture for a Programmable Logic Device," in *IEEE 1993 Custom Integrated Circuits Conference*, 1993.

[47] "http://www.xilinx.com", Xilinx web site.

[48] J. Rose, E. Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field Programmable Gate Arrays," in *Proceedings of the IEEE*, vol. 81(7), pp. 1013–1029, July 1993.

[49] S. Singh et al., "The Effect of Logic Block Architecture on FPGA Performance," *IEEE J. Solid-State Circuits*, vol. 27, no. 3, pp. 281–287, 1990.

[50] J. S. Rose and S. Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays," *IEEE J. Solid-State Circuits*, vol. 26, no. 3, pp. 227–282, 1991.

[51] J.-P. Kaps and C. Paar, "Fast DES Implementation for FPGAs and its Application to a Universal Key-Search Machine," in *Proceedings of SAC'98*, (Kingston, Ont.), August 1998.

[52] "http://www.cmc.ca." , CMC web site.

[53] Garcia, O.N., H. Glass, and S. C. Haimes, "An Approximate and Empirical Study of the Distribution of the Adder Inputs and Maximum Carry Length Propagation," in *Proceedings of* 4^{th} *IEEE Symposium on Computer Arithmetic*, pp. 97–103, 1978.

[54] Doran and R. W., "Variants on an Improved Carry Lookahead Adder," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1110–1113, 1988.

[55] O. Bedrijj, "Carry Select Adder," *IRE Transactions on Electronic Computers*, vol. EC-11, pp. 340–346, 1960.

[56] L. Dadda, "Some Schemes for Fast Serial Input Multipliers," in *Proceedings of* 6^{th} *IEEE Symposium on Computer Arithmetic*, pp. 52–59, 1983.

[57] D. P. Agarwal, "Optimum Array-Like Structures for High-Speed Arithmetic," in *Proceedings of* 3^{rd} *IEEE Symposium on Computer Arithmetic* , pp. 208–219, 1975.

[58] L. Ciminiera and A. Serra, "Fast Iterative Multiplying Array," in *Proceedings of* 6^{th} *IEEE Symposium on Computer Arithmetic* , pp. 60–66, 1983.

[59] Baugh, C. R., and B. A. Wooly, " A Two's Complement Parallel Array Multiplier," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, 1973.

[60] Brubaker, T. A., and J. C. Becker, "Multiplication Using Logarithms Implemented with ROM," *IEEE transactions on Computers*, vol. C-24, no. 8, pp. 761–765, 1975.

[61] Doran and R. W., "A Suggestion for a Fast Multiplier," *IEEE Transactions on Computers*, vol. EC-13, pp. 14–17, 1964.

[62] "http://users.ids.net/ randraka/multipli.htm.", Multiplication in FPGAs.

[63] J. B. Gosling, "Design of Large High-Speed Binary Multiplier Units," in *Proceedings of the IEE*, vol. 118(3), pp. 499–505, 1971.

[64] S. H. Unger, *Asynchronous Sequential Switching Circuits.* Wiley (Interscience Division), 1969.

[65] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Performance Comparison of the AES Submissions" available at web site, "http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm".

[66] X. Zhu, "A New Class of Unbalanced CAST Ciphers and Its Security Analysis" M.Eng. thesis, Memorial University of Newfoundland, St. John's, NF, Canada, 1997.

Appendix A

A VHDL Description of RC6 Global State Machine

```
**********************************************************************************
-- This is a VHDL description of the Global state machine for the RC6 encryptor
-- at the behavioral level. The state machine is based on the Moore FSM model.
-- The global state machine is a synchronous one with asynchronous inputs.
**********************************************************************************

library IEEE;
use IEEE.std_logic_1164.all;
use Work.RC6_types.all;
use IEEE.std_logic_arith.all;

-- Entity Declaration
```

```
entity RC6_STATE_MACHINE is
        port(RESET_CHIP,DONE_44,DONE_4,DONE_OUT,KD,DD,CLK: in std_logic;
             COUNT_20 : in std_logic_vector(4 downto 0);
             COUNT_OUT : in std_logic_vector(1 downto 0);
             EN1,SEL_DEMUX,RESET_44,RESET_4,RESET_20 : out std_logic;
                    EN_OUTPUT_REG,RESET_MUX : out std_logic;
                    STATUS_FLAG : out std_logic_vector(2 downto 0));
end RC6_STATE_MACHINE;

-- Behavioral architecture for the entity
architecture BEHAVIORAL of RC6_STATE_MACHINE is
        type STATE is (RESET, KEY_DOWNLOAD,IDLE,DATA_DOWNLOAD,DATA_ENCRYPT);
-- Signal declarations
        signal CURRENT : STATE := RESET;
        signal COUNT_OUT_INT : INTEGER range 0 to 3;
begin
  COUNT_OUT_INT <= conv_integer(unsigned(COUNT_OUT));
  process
  begin
   case CURRENT is
-- The encryptor is in RESET state
   when RESET =>
    EN1 <= '1';
```

```
    RESET_44 <= '1';
    RESET_4 <= '1';
    RESET_20 <= '1';
    RESET_MUX <= '1';
    EN_OUTPUT_REG <= '1';
    STATUS_FLAG <= "111";
    if(RESET_CHIP = '0') then
      CURRENT <= KEY_DOWNLOAD;
    elsif(RESET_CHIP = '1') then
      CURRENT <= RESET;
    end if;

-- The encryptor is downloading the key
   when KEY_DOWNLOAD =>
    EN1 <= '0';SEL_DEMUX <= '0';RESET_44 <= '0';
    RESET_4 <= '1';RESET_20 <= '1';
    RESET_MUX <= '1';
    EN_OUTPUT_REG <='1'; STATUS_FLAG <= "001";
    if(DONE_44 = '0') then
     STATUS_FLAG <= "010";
     CURRENT <= IDLE;
     RESET_44 <='1';
    elsif(RESET_CHIP = '1') then
```

```
      CURRENT <= RESET;
     end if;

-- The encryptor is in the idle state.
    when IDLE =>
     EN1 <= '1';
     RESET_44 <= '1';
     RESET_4 <= '1';
     RESET_20 <= '1';
     RESET_MUX <= '1';
     EN_OUTPUT_REG <= '1';
     STATUS_FLAG <= "000";
     if(KD = '0' and DD = '1' and RESET_CHIP = '0') then
      CURRENT <= KEY_DOWNLOAD;
     elsif(KD = '1' and DD = '0' and RESET_CHIP = '0') then
      CURRENT <= DATA_DOWNLOAD;
     elsif((KD = DD) and RESET_CHIP = '0') then
      CURRENT <= IDLE;
     elsif(RESET_CHIP = '1') then
      CURRENT <= RESET;
     end if;

-- The encryptor is downloading the data
```

```
when DATA_DOWNLOAD =>
 EN1 <= '0';SEL_DEMUX <= '1';RESET_44 <= '1';RESET_4 <= '0';
 RESET_20 <= '1';
 RESET_MUX <= '1';
 EN_OUTPUT_REG <= '1';STATUS_FLAG <= "011";
 if(DONE_4 = '0') then
  STATUS_FLAG <= "100";
  CURRENT <= DATA_ENCRYPT;
  RESET_4 <= '1';
 elsif((KD = DD) and RESET_CHIP = '0') then
  CURRENT <= IDLE;
 elsif(RESET_CHIP = '1') then
  CURRENT <= RESET;
 end if;

-- The encryptor is encrypting the data
 when DATA_ENCRYPT =>
 EN1 <= '1';RESET_44 <='1';RESET_4 <= '1';
 RESET_MUX <= '0';RESET_20 <= '0';
 STATUS_FLAG <= "101";
 if(COUNT_20 = "10101") then
  EN_OUTPUT_REG <= '0';
  if COUNT_OUT_INT >= 0 then
```

```
    wait until CLK = '1';
    STATUS_FLAG <= "110";
    wait until CLK = '1';
    STATUS_FLAG <= "110";
    wait until CLK = '1';
    STATUS_FLAG <= "110";
    wait until CLK = '1';
    STATUS_FLAG <= "110";
  end if;
  if(DONE_OUT = '0') then
    EN_OUTPUT_REG <= '1';
  end if;
    if(DONE_OUT = '0' and (KD = DD) and RESET_CHIP = '0') then
       CURRENT <= IDLE;
    elsif(RESET_CHIP = '1') then CURRENT <= RESET;
    elsif(DONE_OUT = '0' and KD = '1' and DD = '0' and RESET_CHIP = '0') then
       CURRENT <= DATA_DOWNLOAD;
    end if;
 elsif(RESET_CHIP = '1') then
  CURRENT <= RESET;
 end if;
end case;
wait until CLK = '1';
```

```
  end process;
end BEHAVIORAL;
```

Appendix B

Gate-Level Simulation of RC6 Cipher

This appendix shows the gate-level simulation results for the design of RC6 cipher. The entire simulation is divided into nine segments, with each segment illustrating a particular mode of operation of the cipher. The simulation figures illustrate all the five modes - data-download mode, keys-download mode, reset mode, idle mode, and the data-encrypt mode. The simulation also shows the state of the three asynchronous signals, namely, RESET-CHIP, KD, and DD.

During the reset-mode, all the control inputs to the datapath are disabled and as such no ciphertext appears at the ouptut of the cipher as illustrated in the simulation figures. During the key-download mode, forty four 32-bit subkeys are downloaded into the cipher. These subkeys are to be used for encryption during the data-encrypt mode. During the data-download mode, the 128-bit plaintext block is downloaded into the cipher. Finally when the global state machine is in the data-encrypt mode, the 128-bit plaintext is encrypted synchronously until finally at the end of the required number of rounds of encryption, the 128-bit ciphertext is available at the 32-bit output bus of the cipher. The idle mode is used

to provide more flexibilty to the RC6 global state machine.

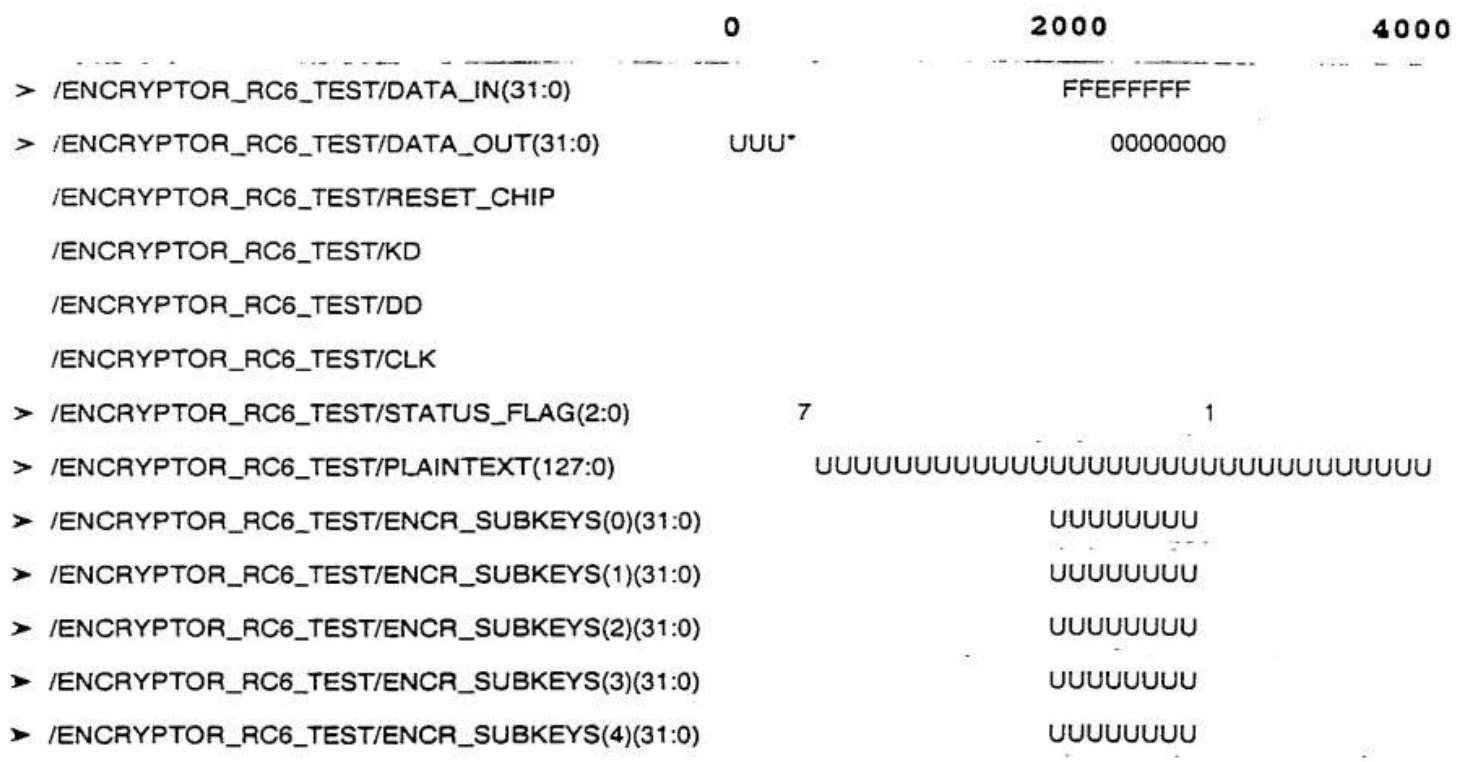

Figure B.1: Gate-Level Simulation of RC6 Cipher Design

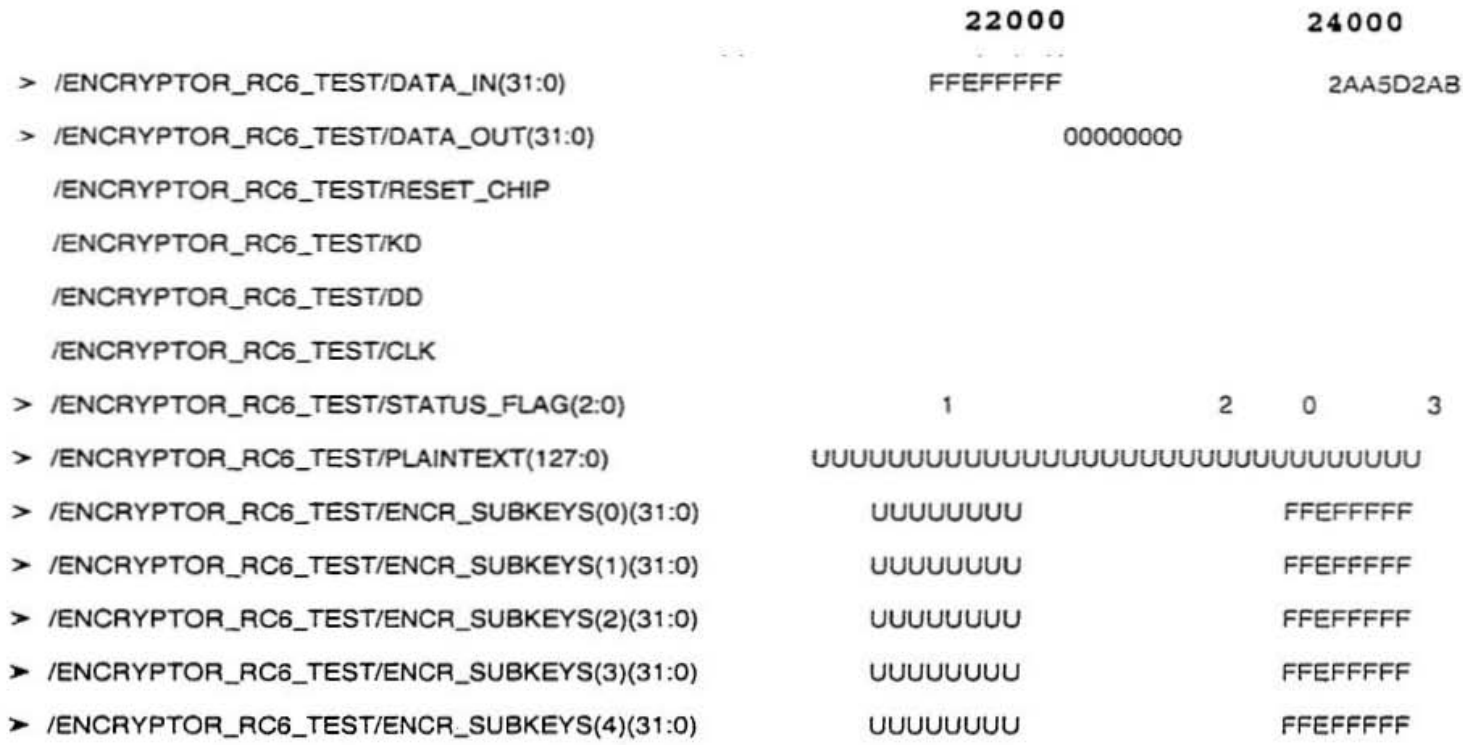

Figure B.2: Gate-Level Simulation of RC6 Cipher Design Cont'd

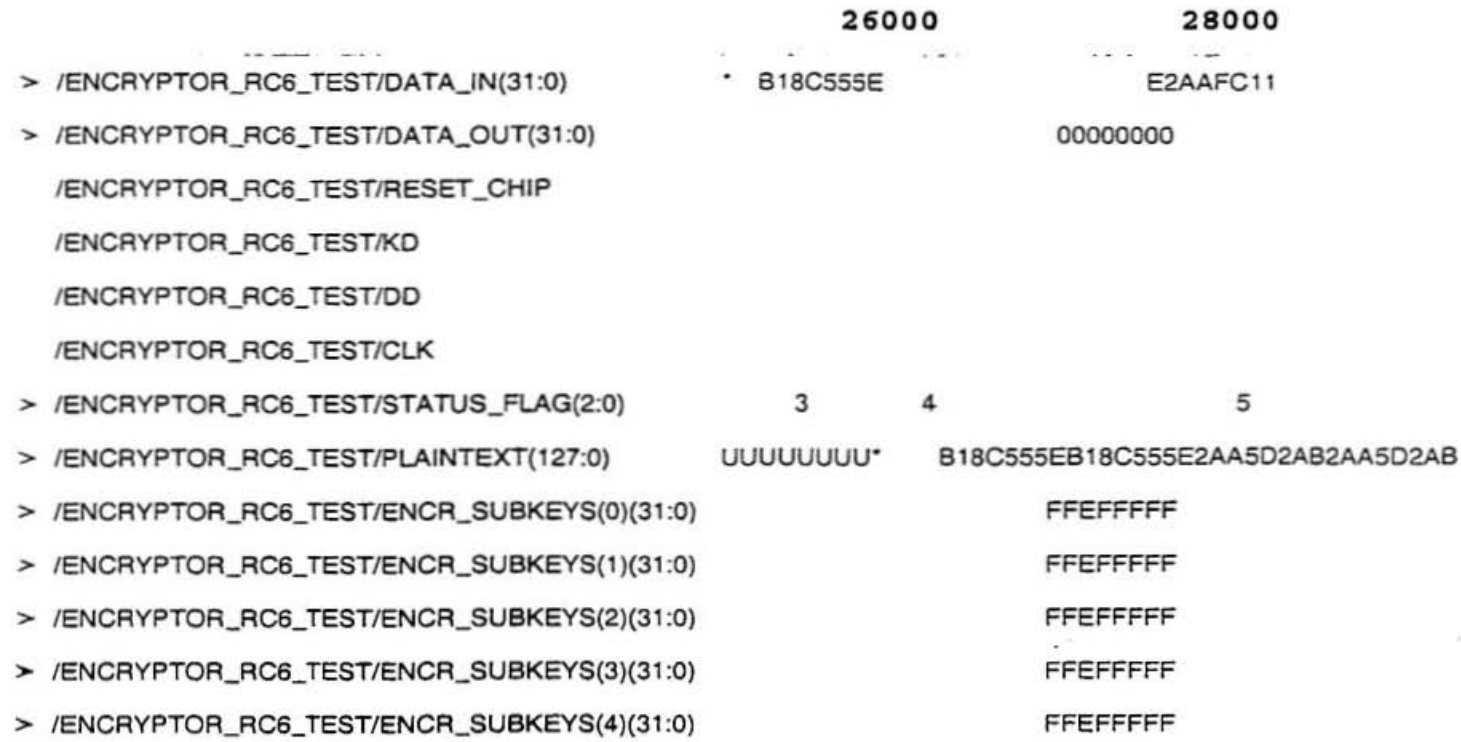

Figure B.3: Gate-Level Simulation of RC6 Cipher Design Cont'd

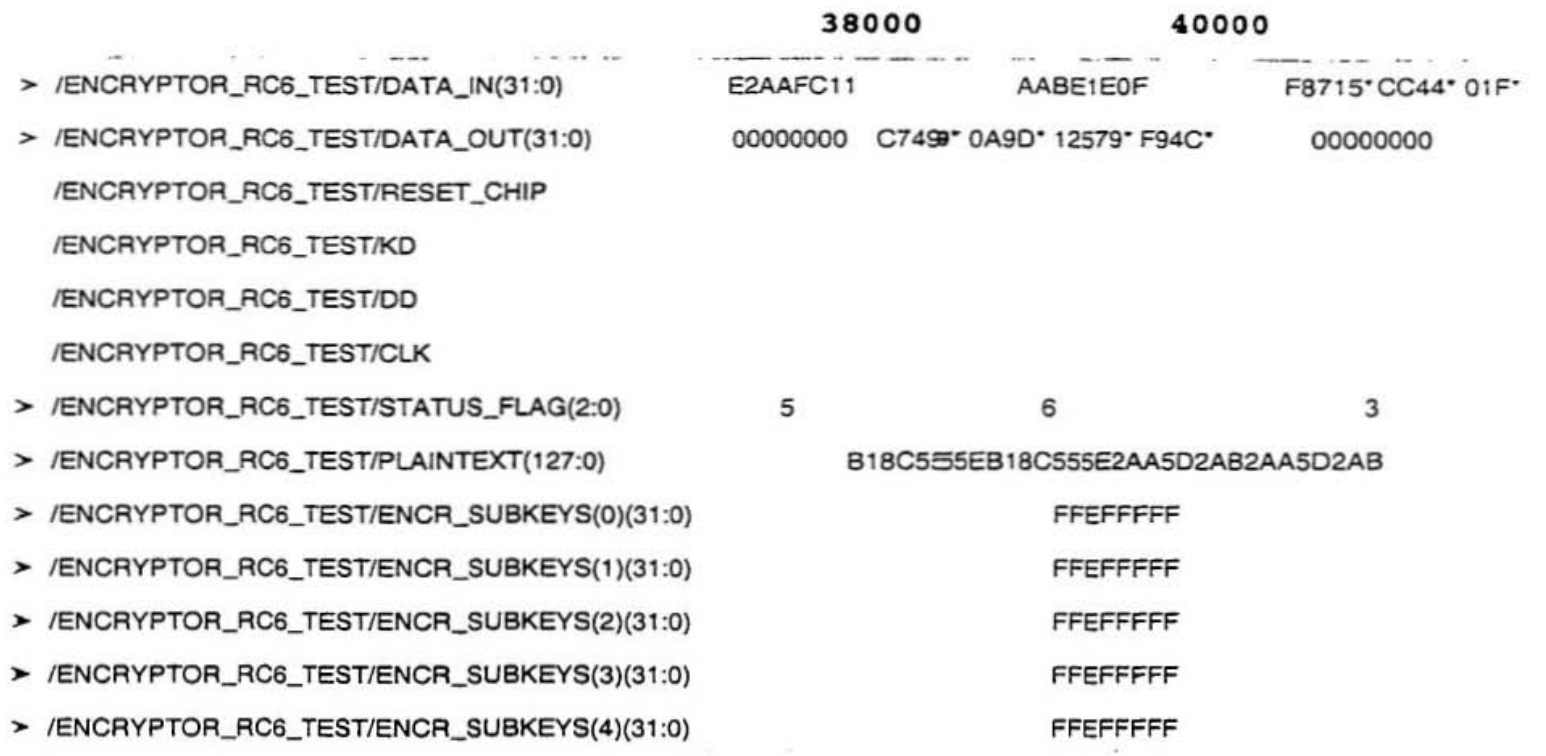

Figure B.4: Gate-Level Simulation of RC6 Cipher Design Cont'd

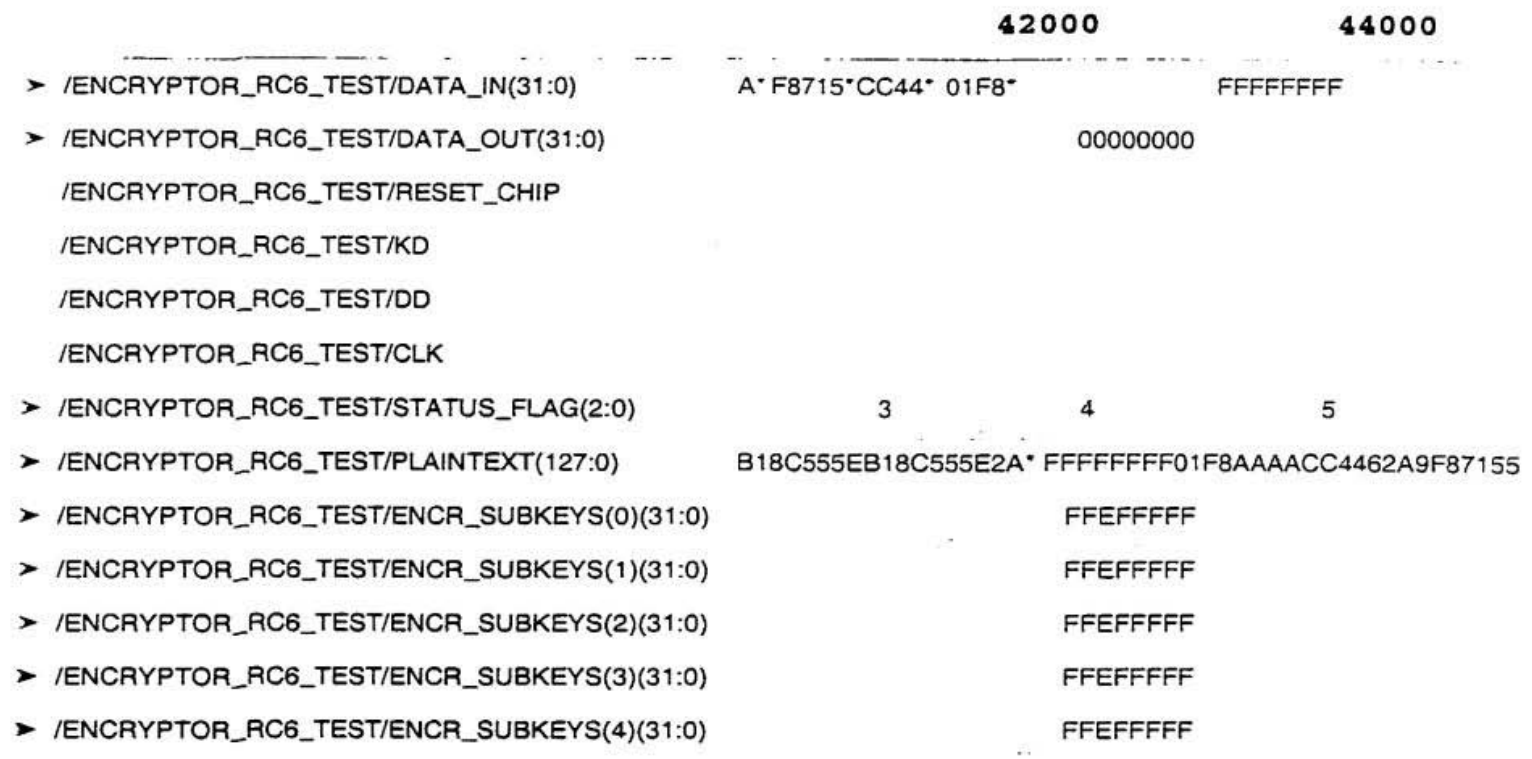

Figure B.5: Gate-Level Simulation of RC6 Cipher Design Cont'd

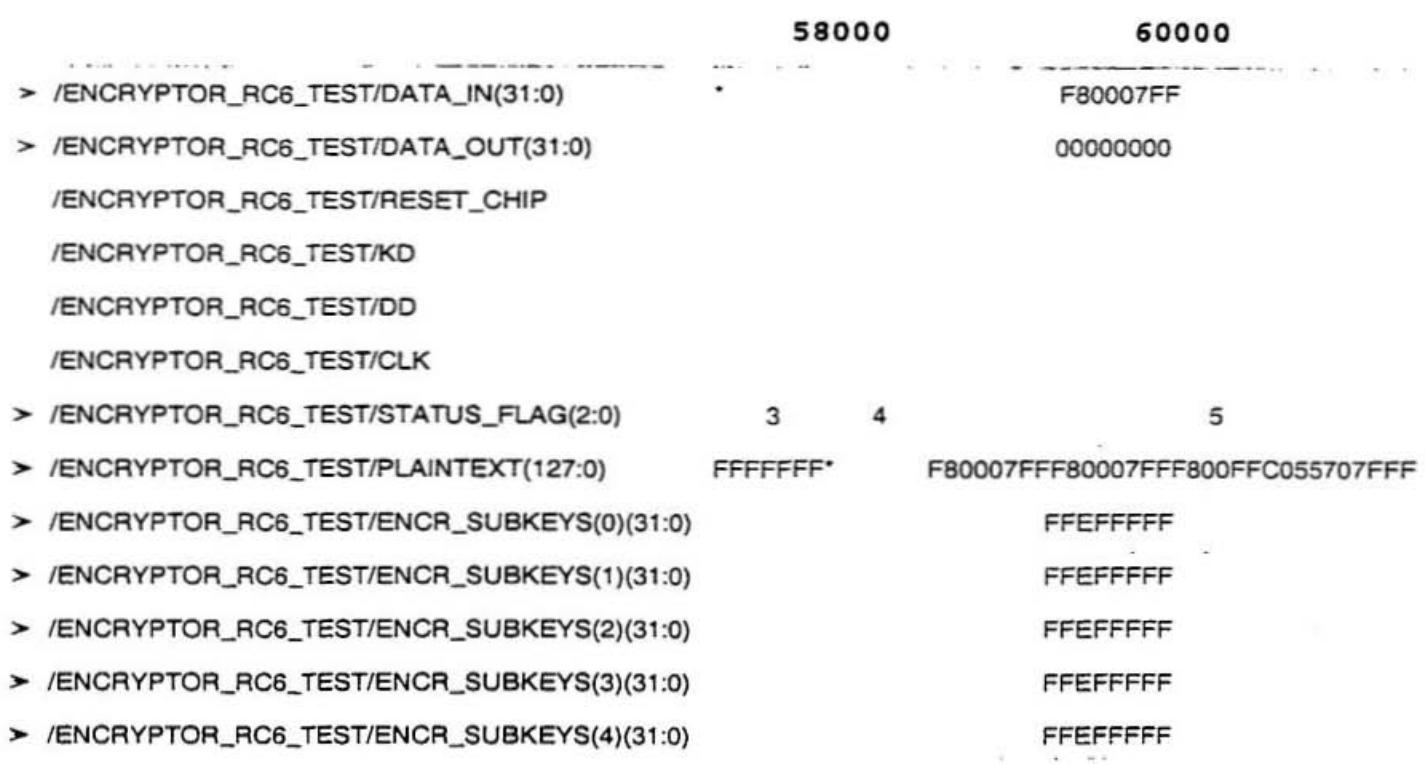

Figure B.6: Gate-Level Simulation of RC6 Cipher Design Cont'd

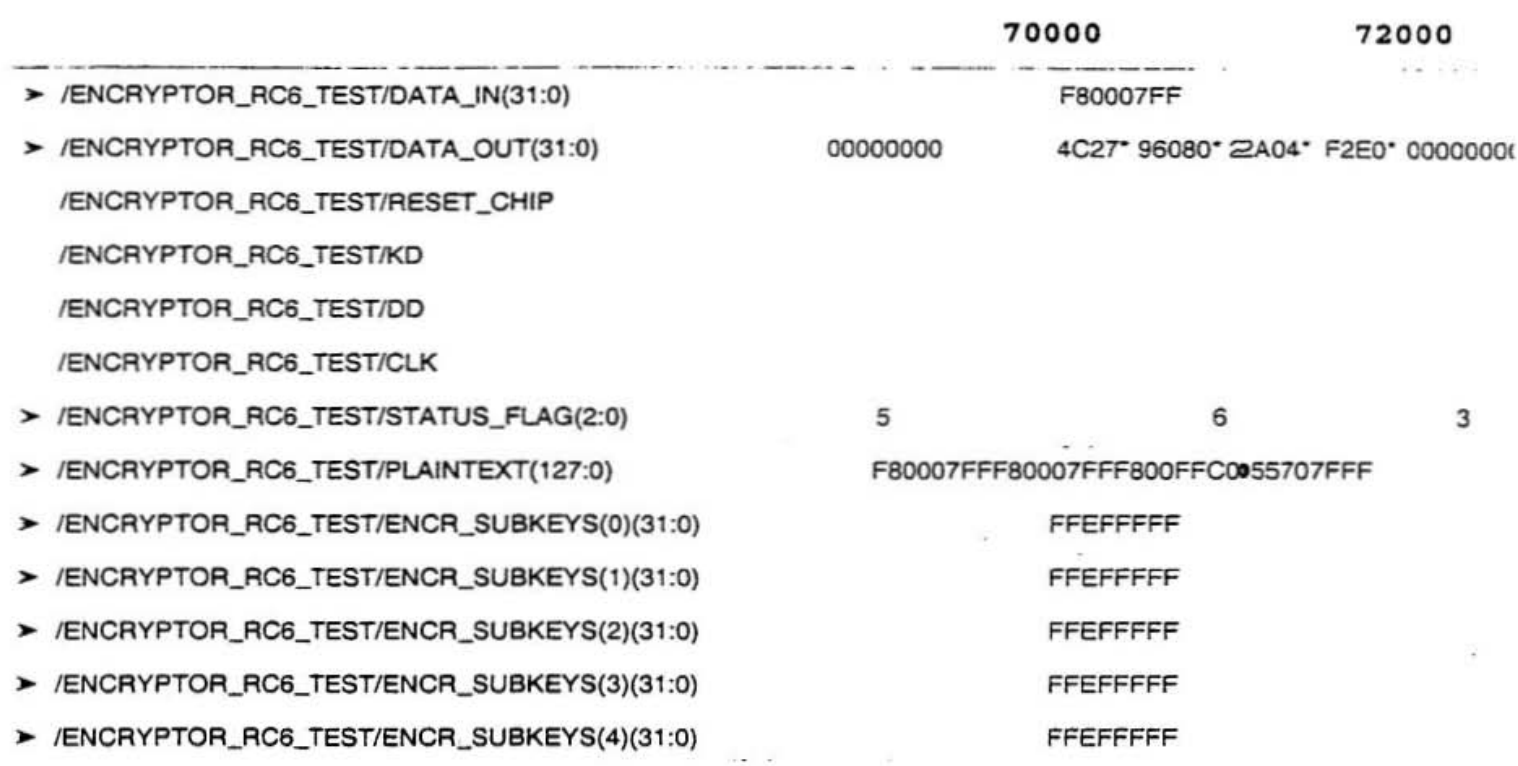

Figure B.7: Gate-Level Simulation of RC6 Cipher Design Cont'd

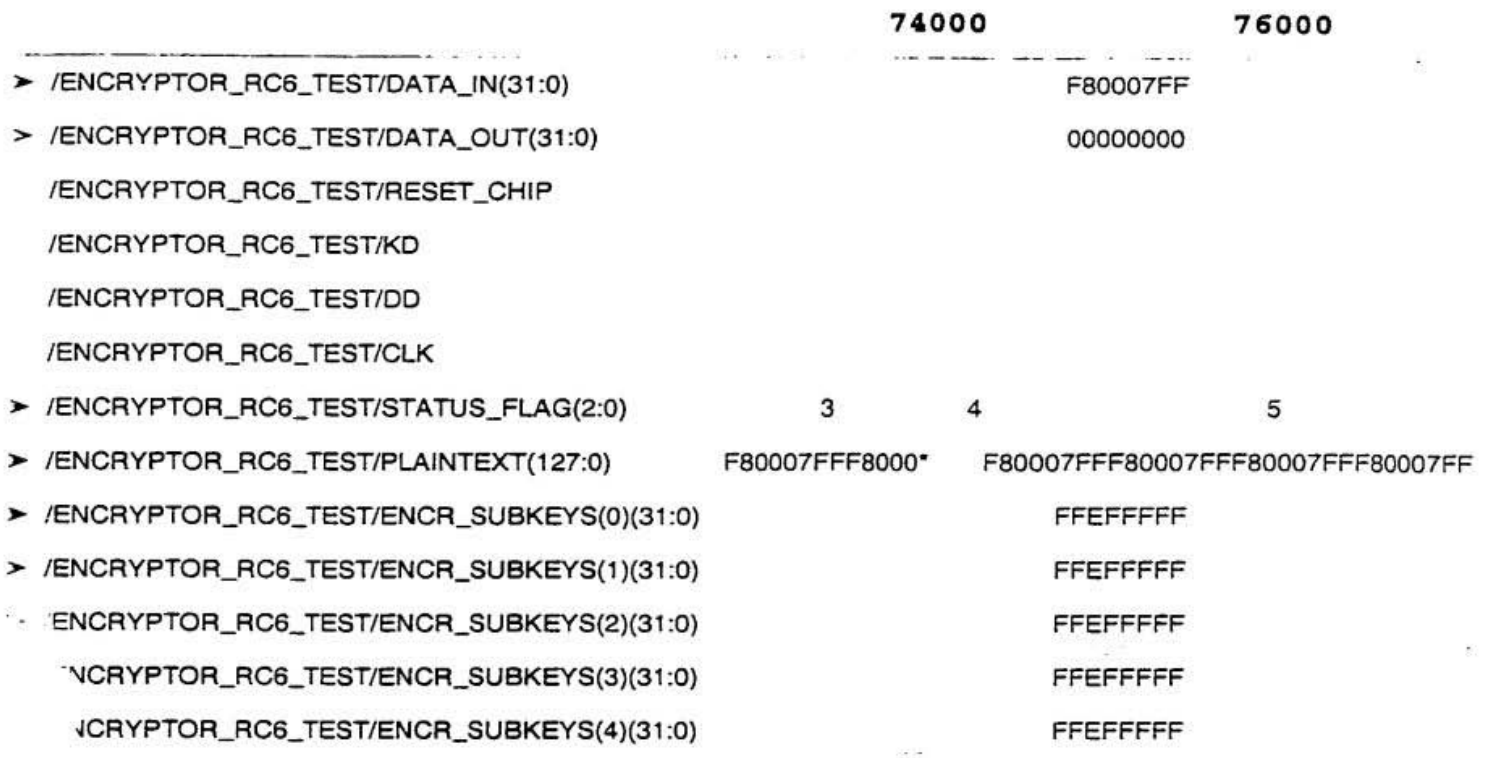

Figure B.8: Gate-Level Simulation of RC6 Cipher Design Cont'd

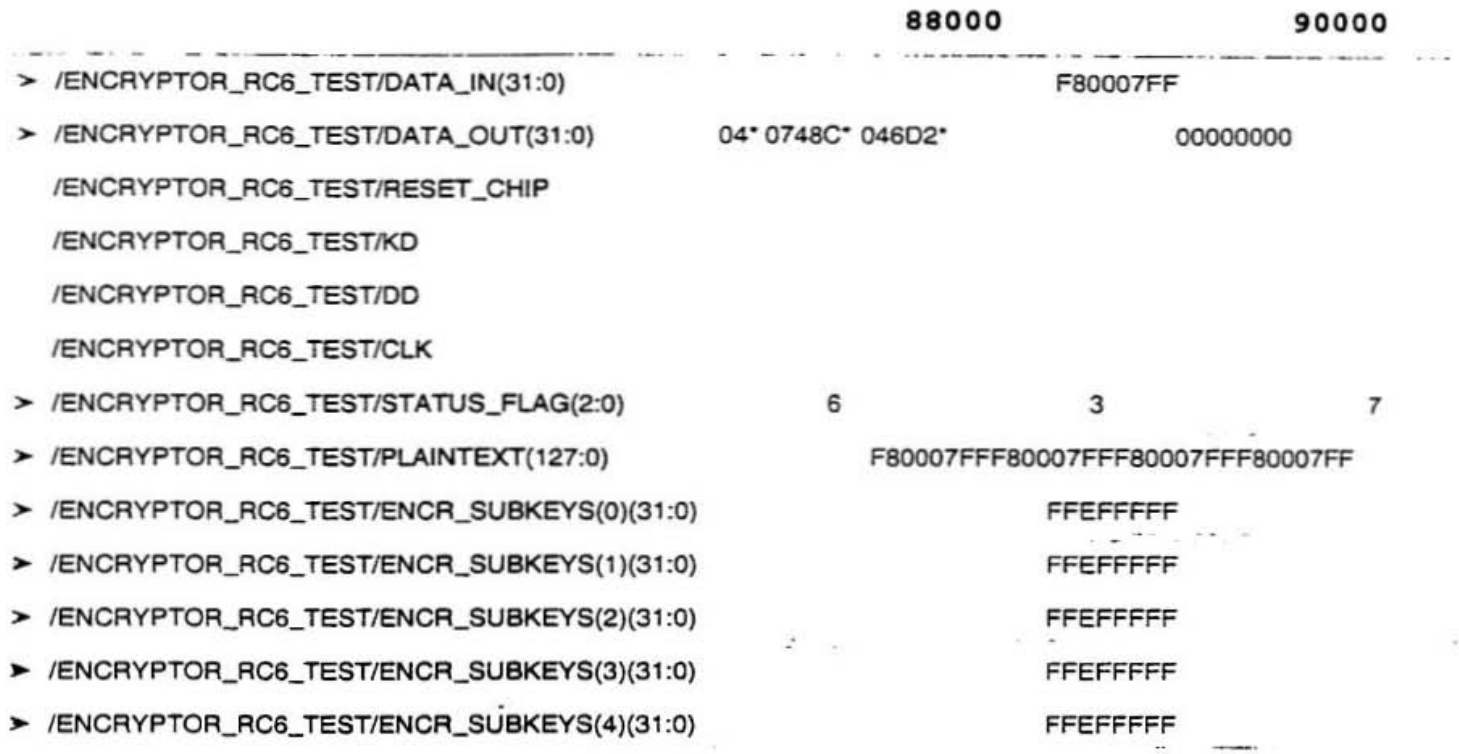

Figure B.9: Gate-Level Simulation of RC6 Cipher Design Cont'd

Appendix C

Gate-Level Simulation of CAST-256 Cipher

This appendix shows the gate-level simulation results for the design of CAST-256 cipher. The entire simulation is divided into eight segments, with each segment illustrating a particular mode of operation of the cipher. The simulation figures illustrate all the six modes.

During the reset-mode, all the control inputs to the datapath are disabled and as such no ciphertext appears at the output of the cipher as illustrated in the simulation figures. During the masking-key-download mode, forty eight 32-bit subkeys are downloaded into the cipher; whereas during the rotation-key-download mode, same number of 5-bit subkeys are downloaded. When the global state machine is in the data-download mode, the 128-bit plaintext block is downloaded into the cipher. Finally when the global state machine is in the data-encrypt mode, the 128-bit plaintext is encrypted. The idle mode is used to provide more flexibilty to the CAST-256 global state machine.

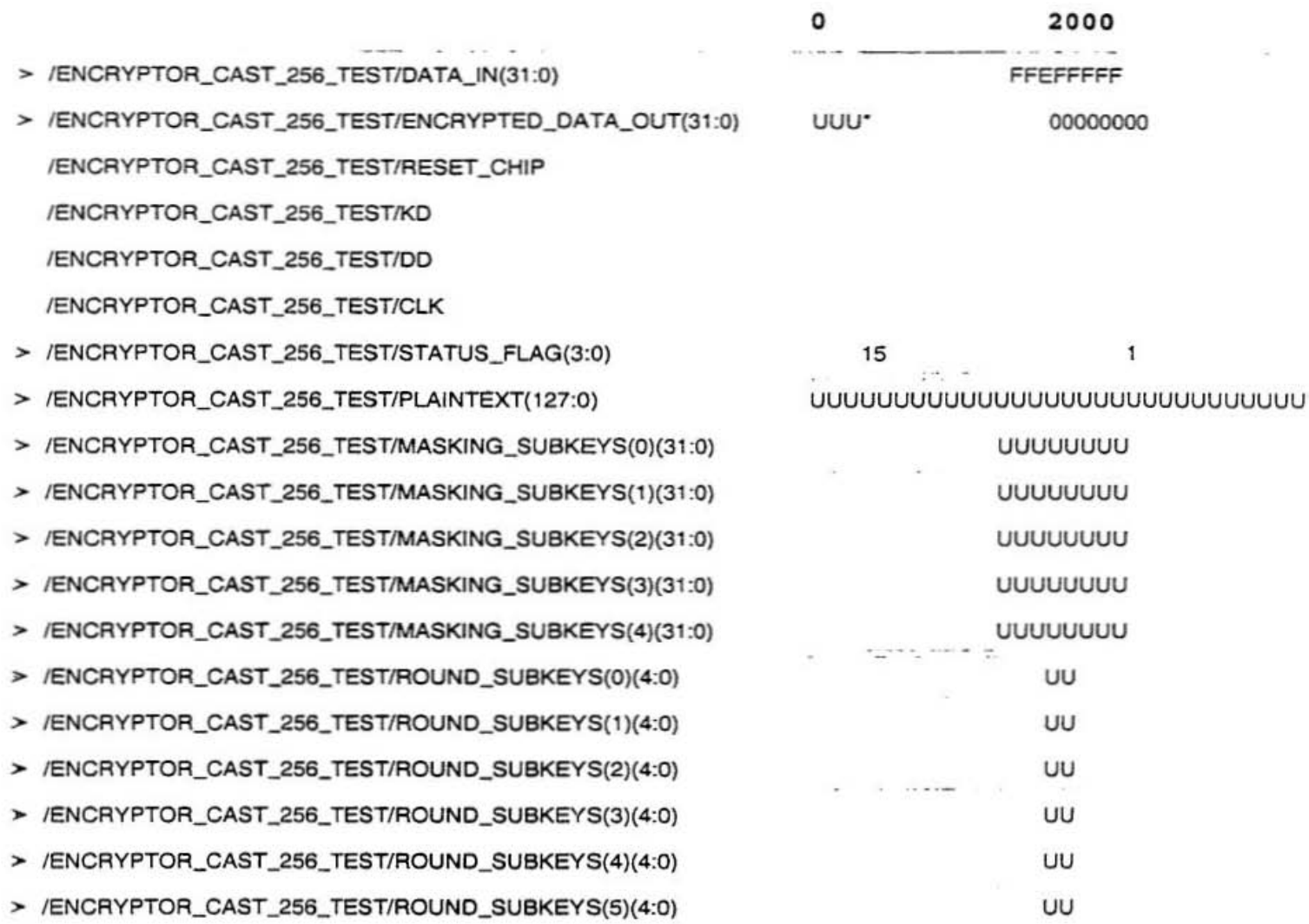

Figure C.1: Gate-Level Simulation of CAST-256 Cipher Design

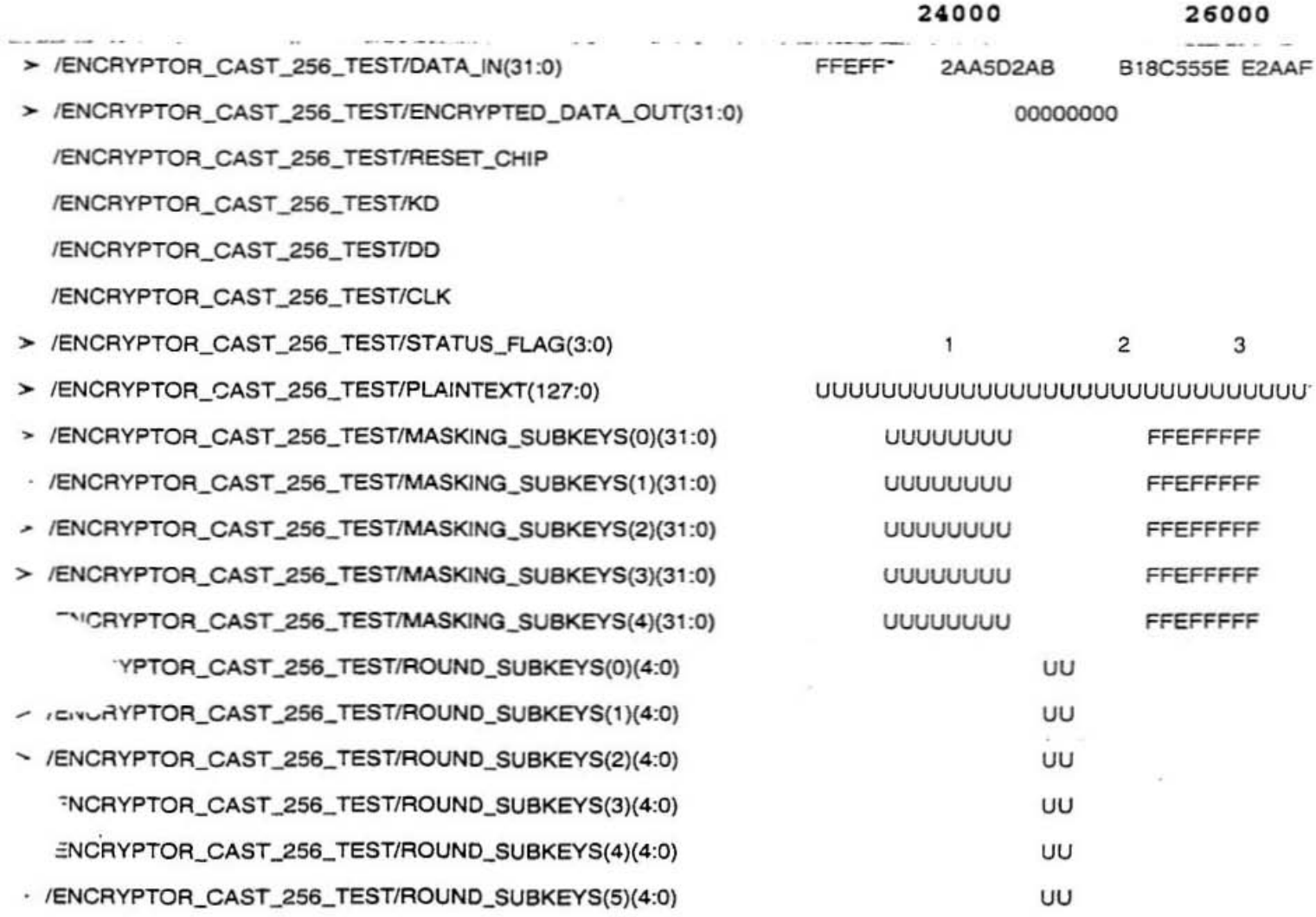

Figure C.2: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

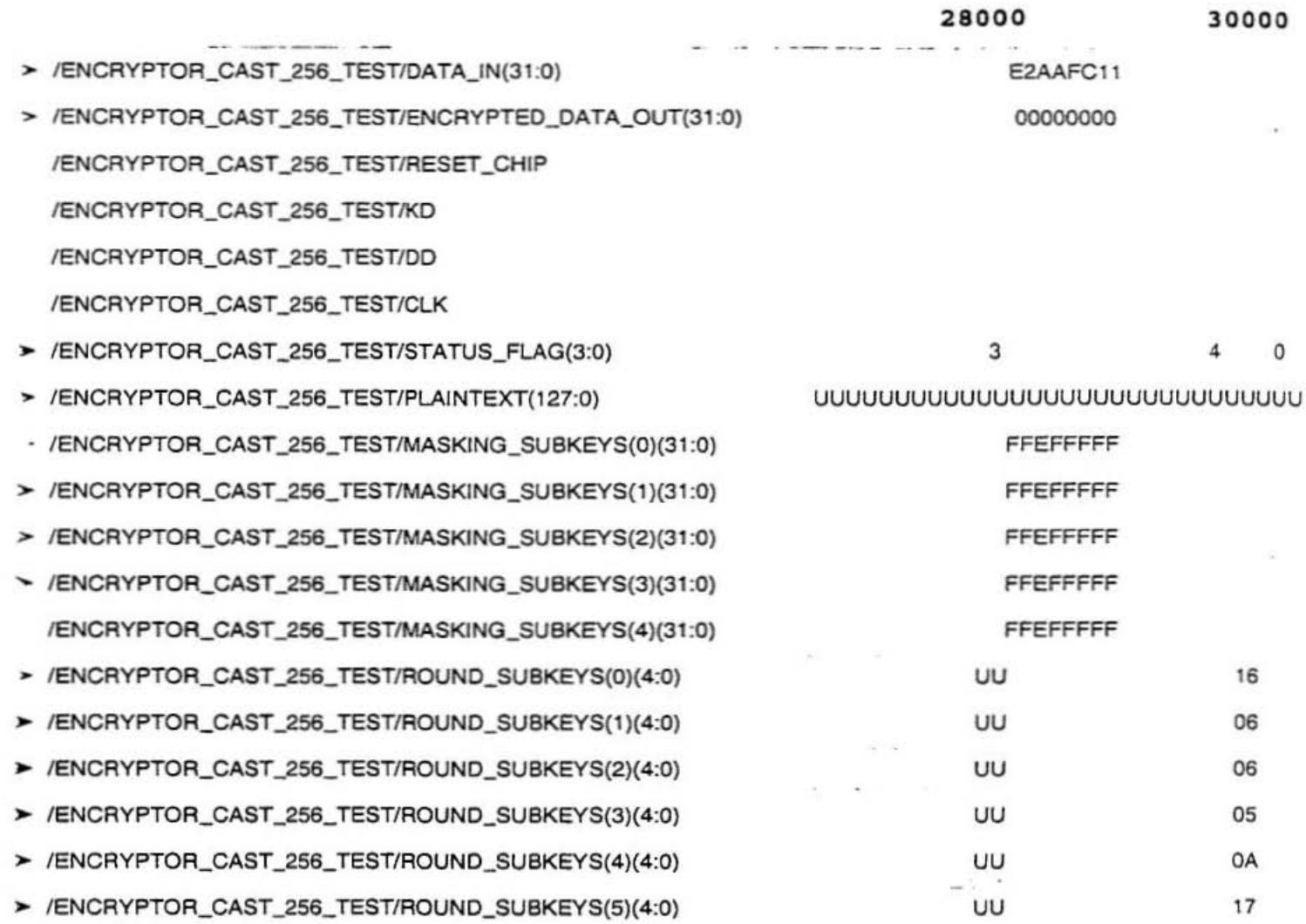

Figure C.3: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

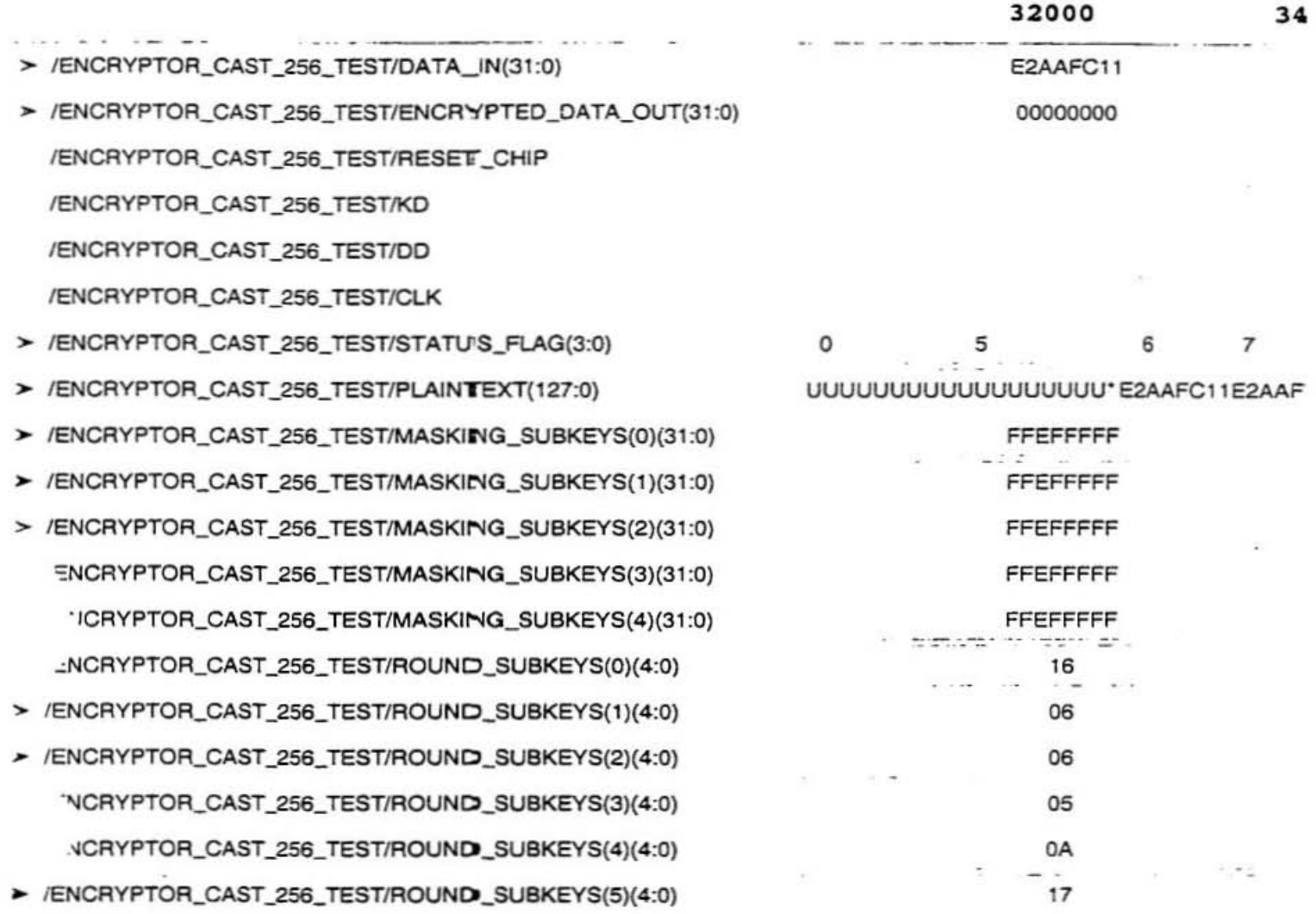

Figure C.4: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

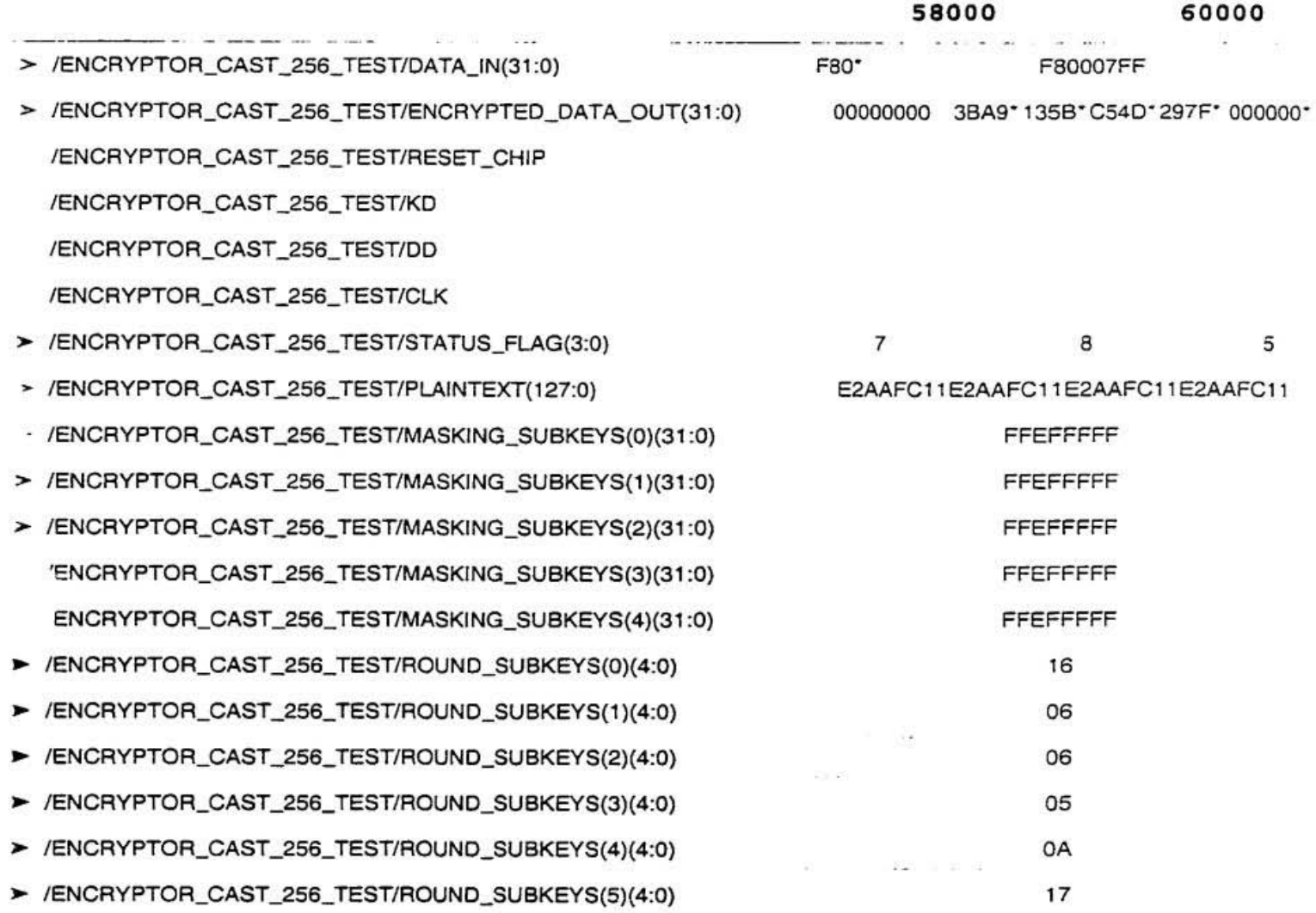

Figure C.5: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

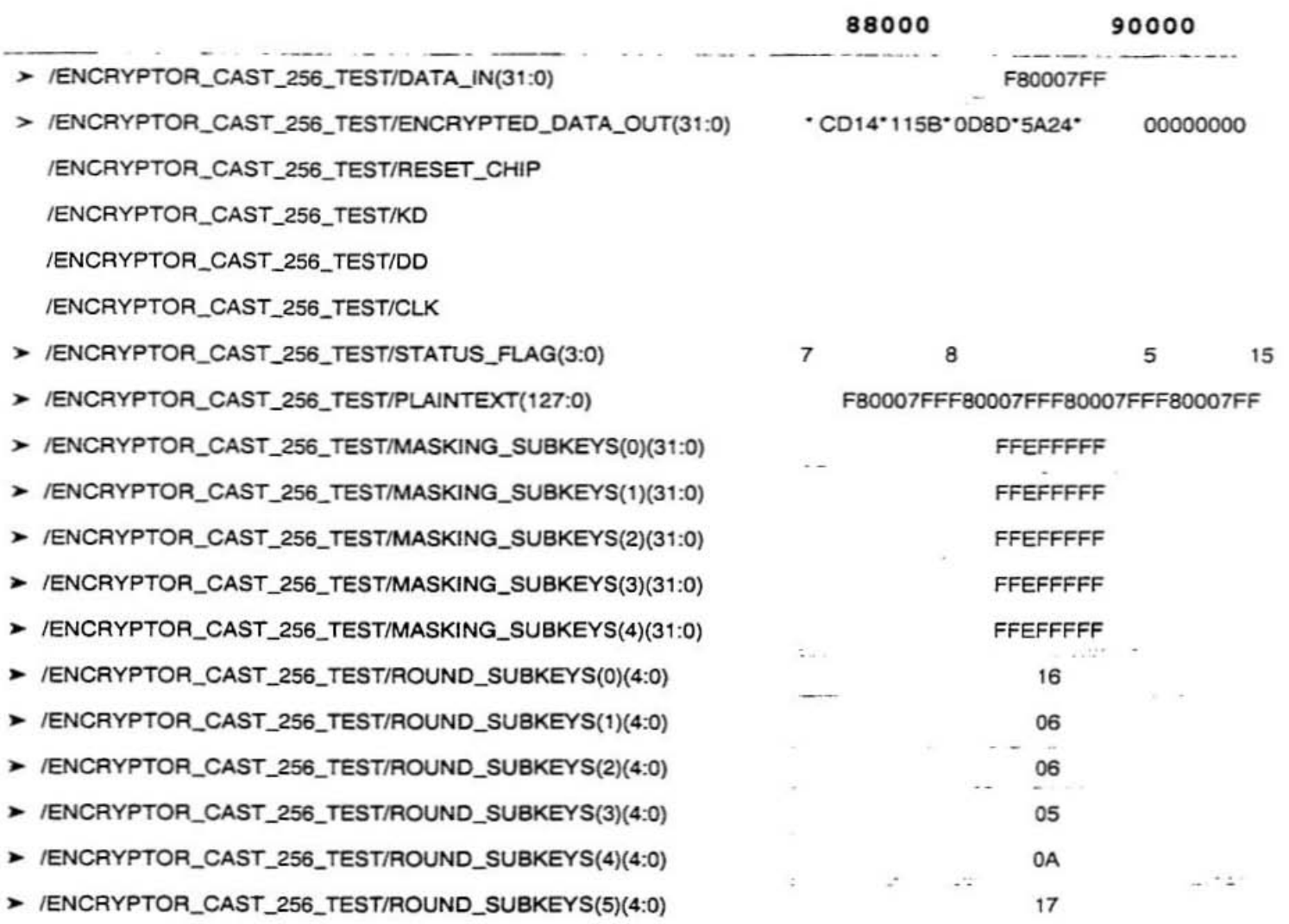

Figure C.6: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

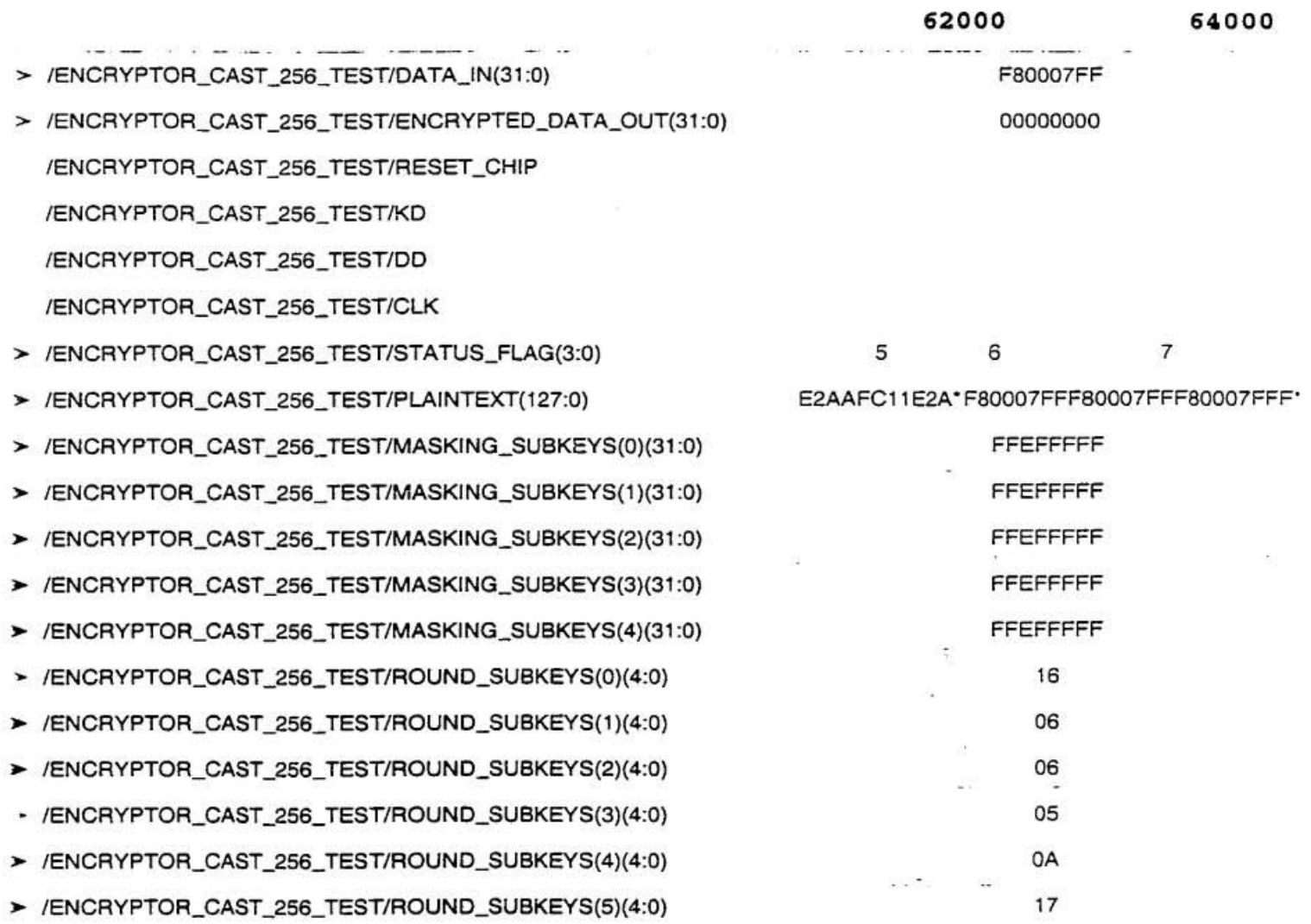

Figure C.7: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

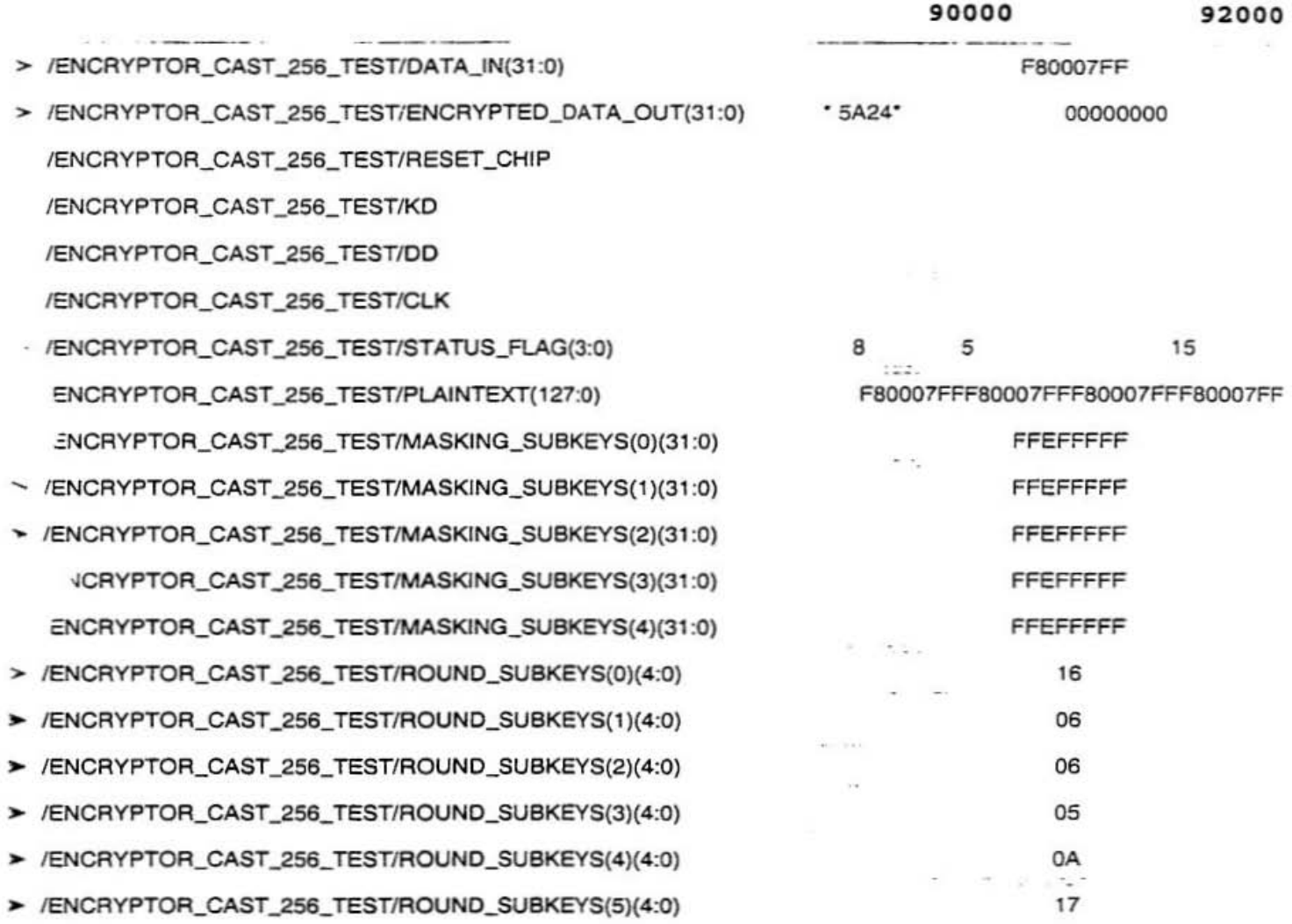

Figure C.8: Gate-Level Simulation of CAST-256 Cipher Design Cont'd

Appendix D

Gate-Level Simulation of Fast Hardware Cipher (FHC)

This appendix shows the gate-level simulation results for the design of the proposed cipher, referred to as Fast Hardware Cipher or FHC. The entire simulation is divided into eight segments, with each segment illustrating a particular mode of operation of the cipher. The simulation figures illustrate all the five modes - reset mode (7), key-download mode (1,2), data-download mode (3, 4), idle mode (0), and data-encrypt mode (5,6).

During the reset-mode, all the control inputs to the datapath are disabled and as such no ciphertext appears at the ouptut of the cipher as illustrated in the simulation figures. During the key-download mode, ninety six 32-bit subkeys are downloaded into the cipher; whereas during the data-download mode, the 128-bit plaintext block is downloaded into the cipher. Finally when the global state machine is in the data-encrypt mode, the 128-bit plaintext is encrypted synchronously until finally at the end of the required number of rounds of encryption, the 128-bit ciphertext is available at the 32-bit output bus of the cipher.

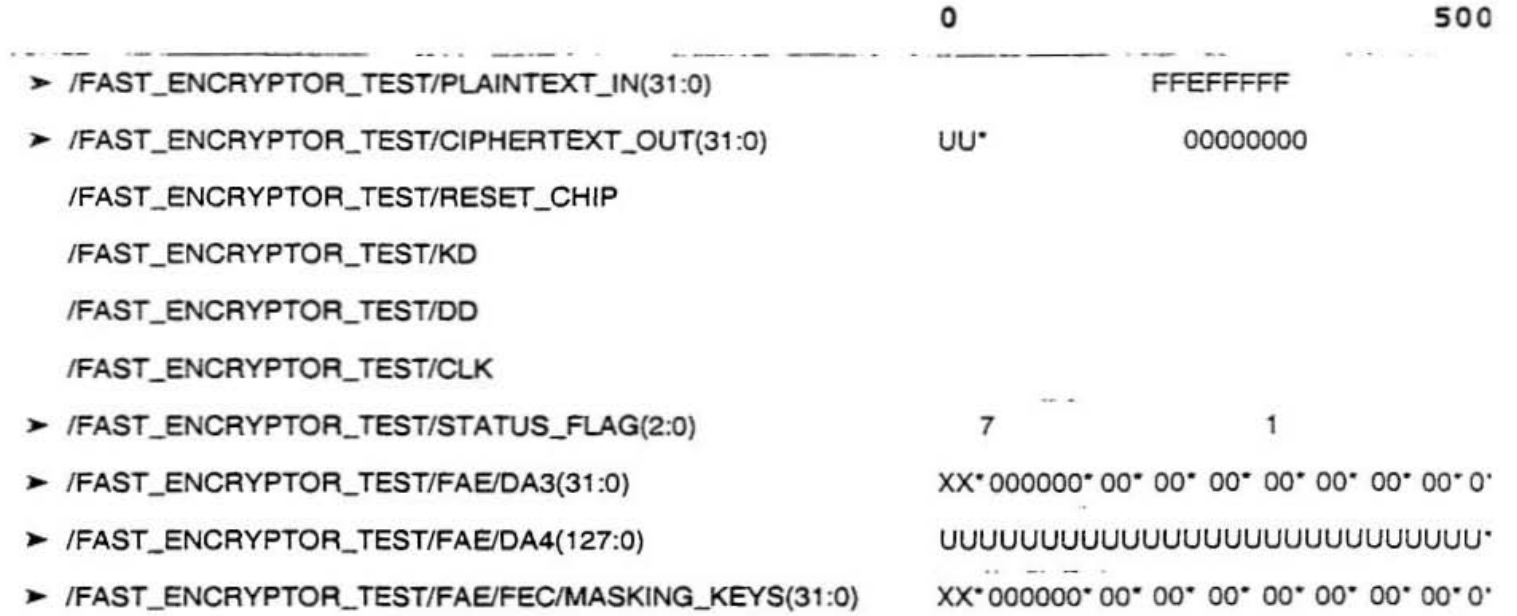

Figure D.1: Gate-Level Simulation of FHC Design

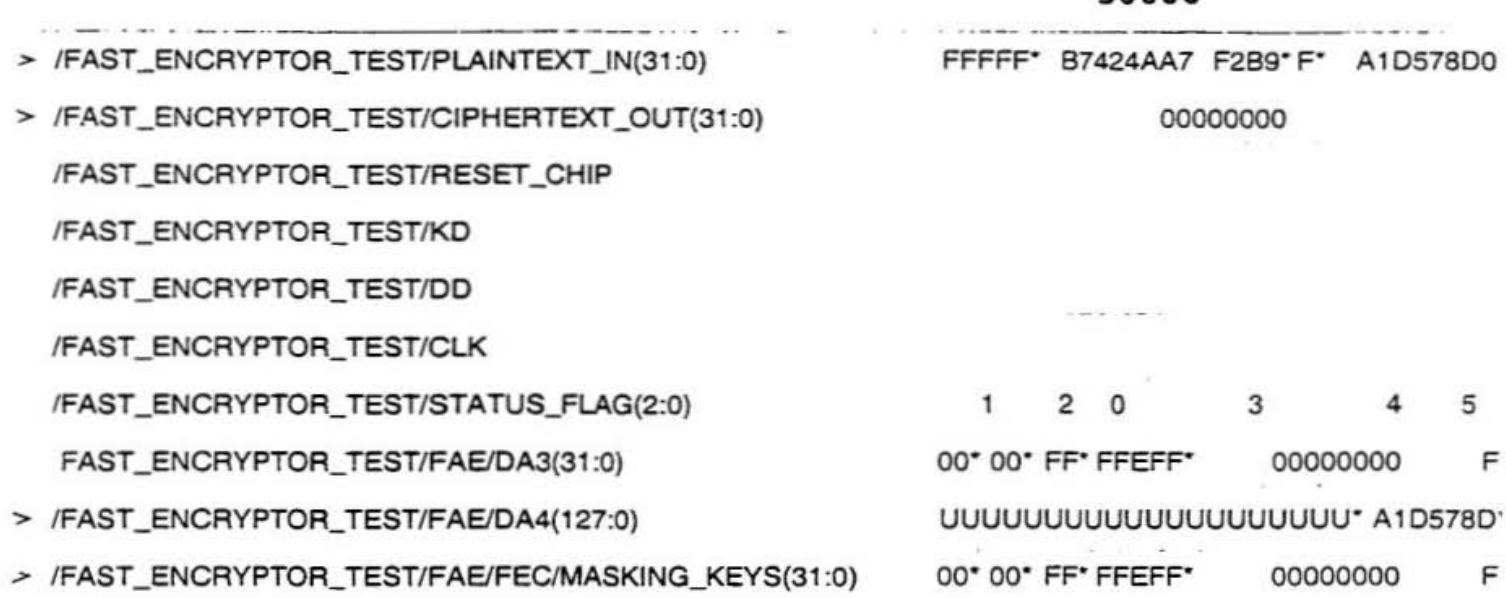

Figure D.2: Gate-Level Simulation of FHC Design Cont'd

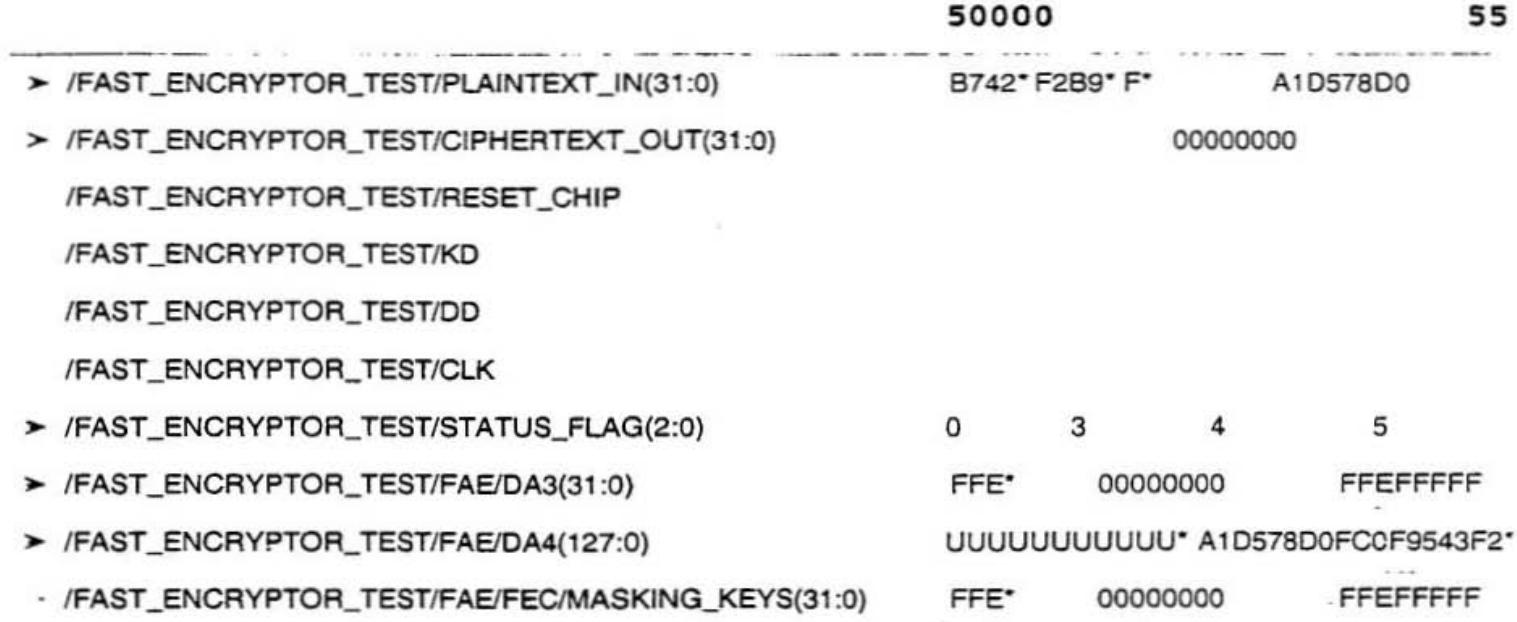

Figure D.3: Gate-Level Simulation of FHC Design Cont'd

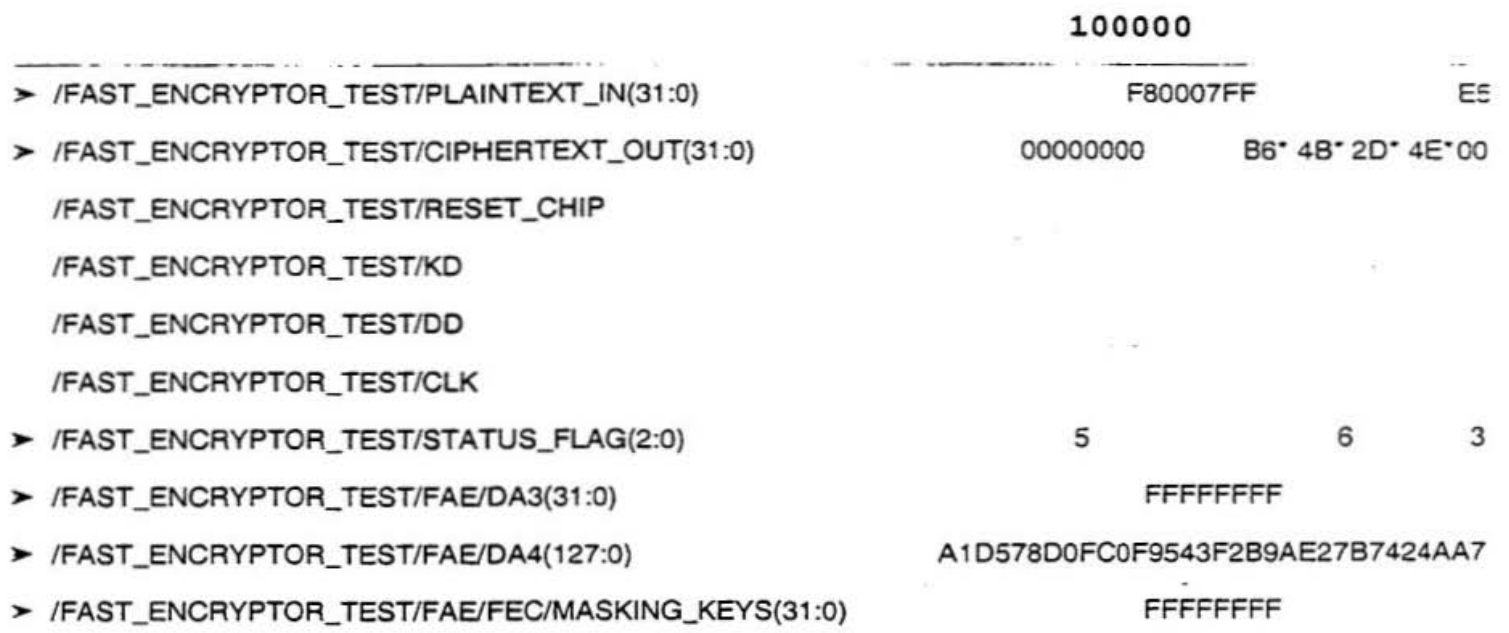

Figure D.4: Gate-Level Simulation of FHC Design Cont'd

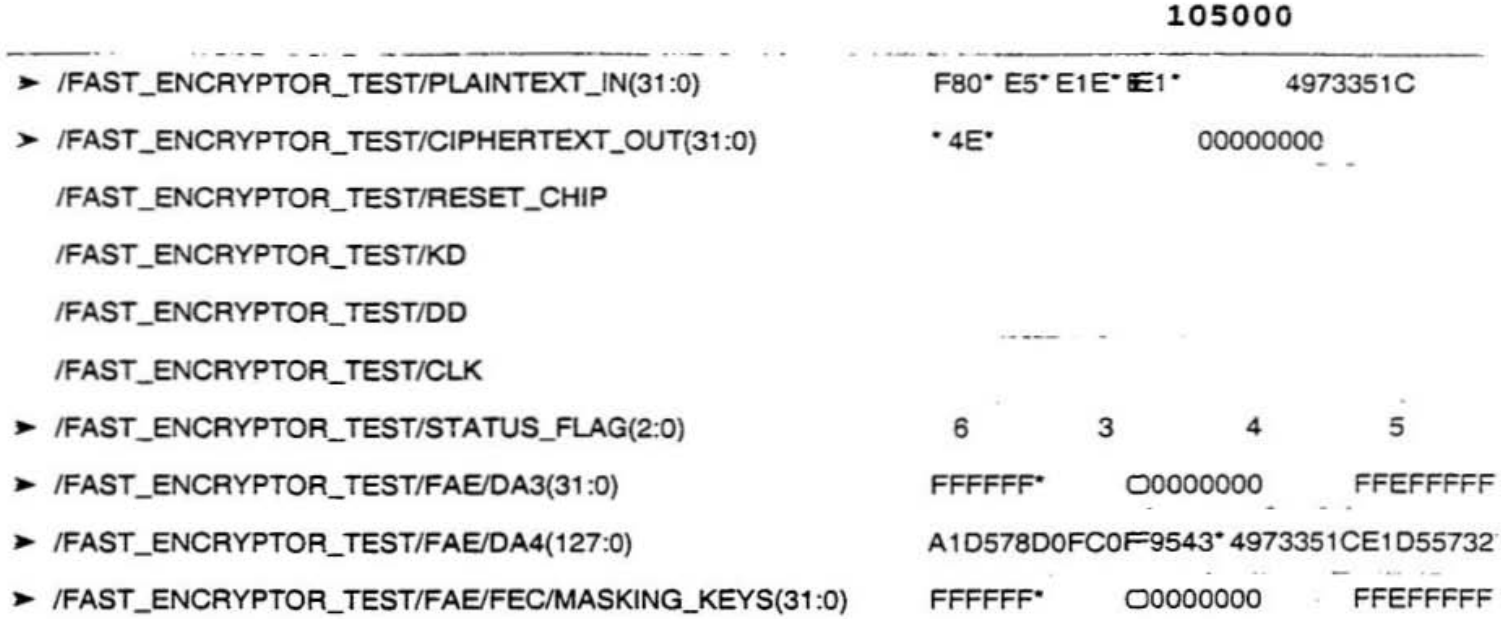

Figure D.5: Gate-Level Simulation of FHC Design Cont'd

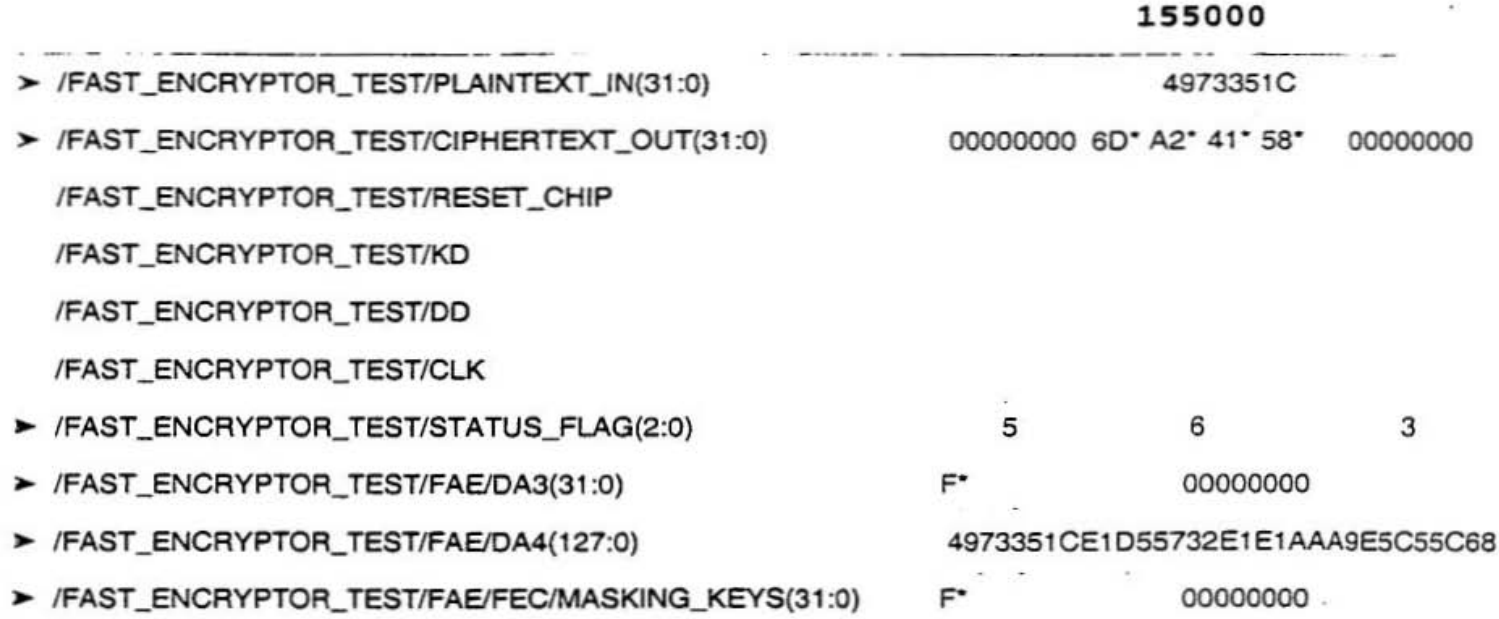

Figure D.6: Gate-Level Simulation of FHC Design Cont'd

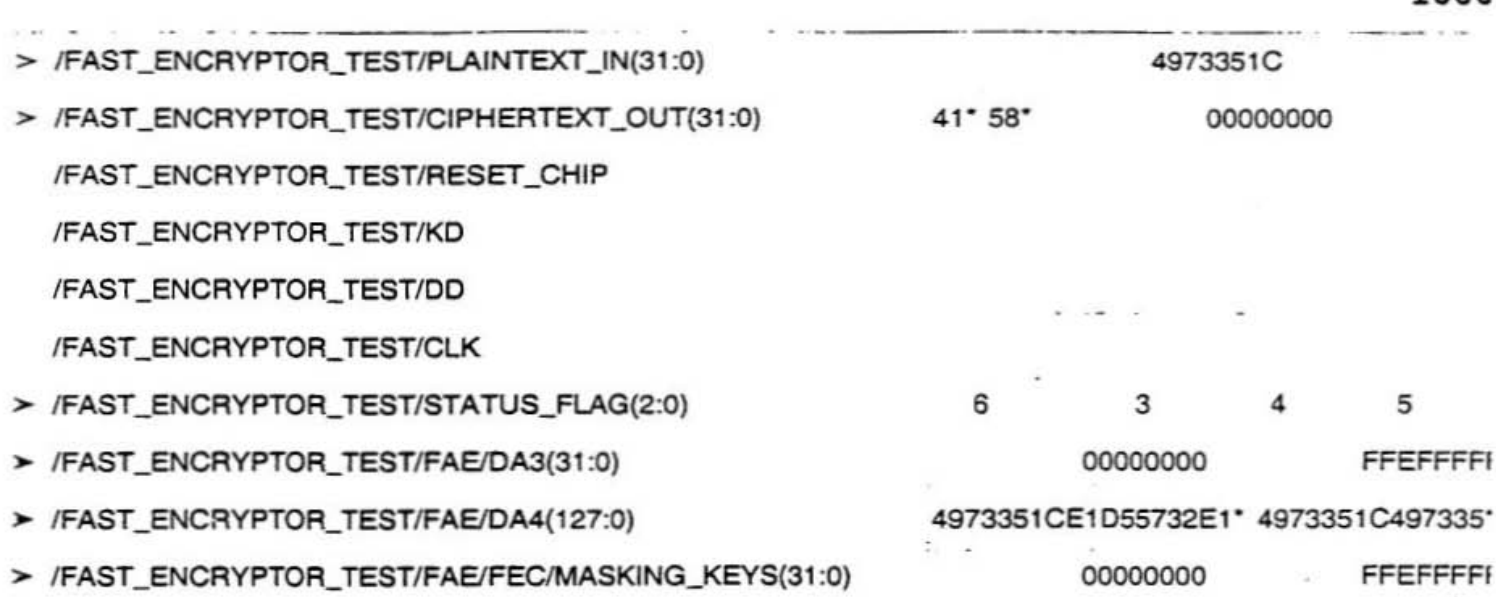

Figure D.7: Gate-Level Simulation of FHC Design Cont'd

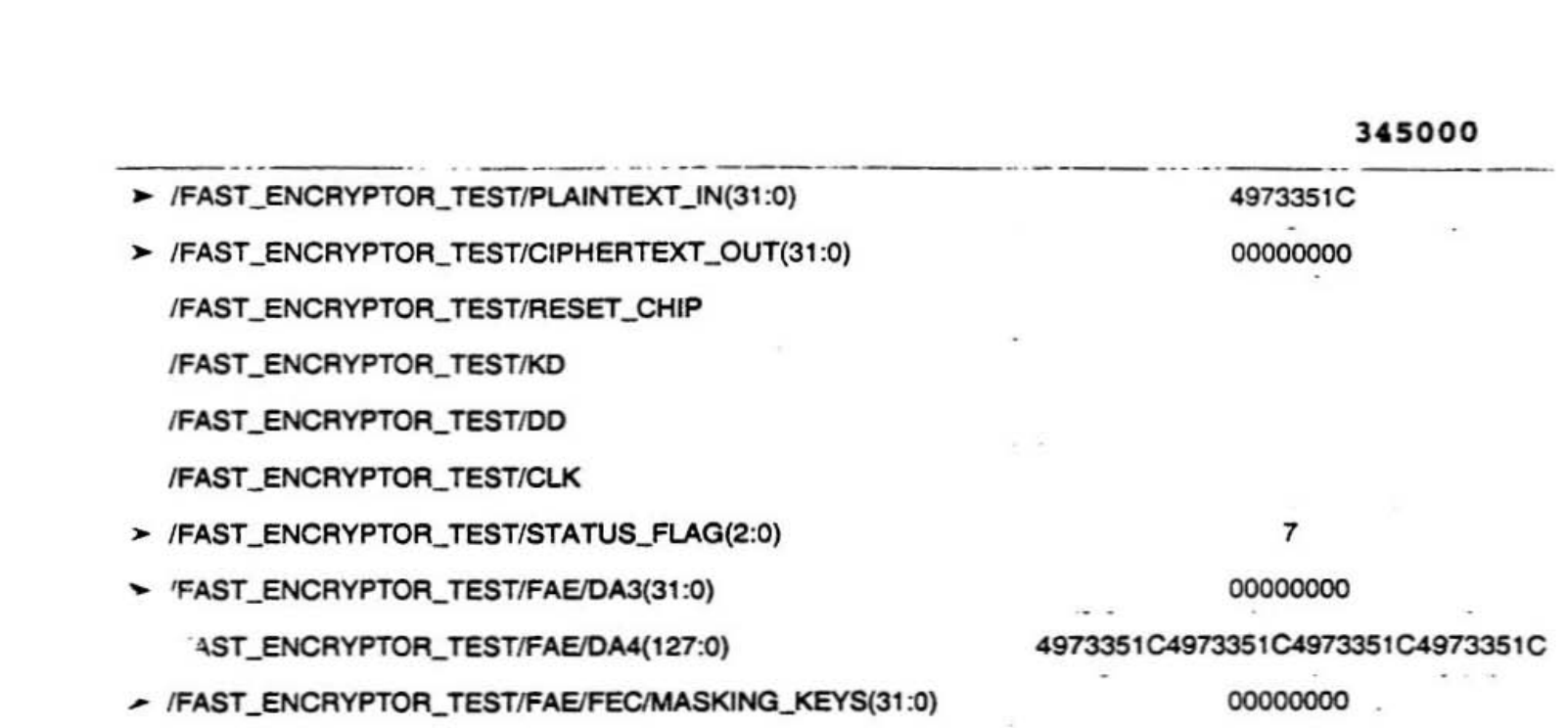

Figure D.8: Gate-Level Simulation of FHC Design Cont'd