An FPGA Implementation of the Advanced Encryption Standard with Support for Counter and Feedback Modes

James Steven Grabowski

A Thesis

in

The Department

of

The Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Information Systems Security) at
Concordia University
Montreal, Quebec, Canada

August 2007

Your file Votre référence
ISBN: 978-0-494-34745-4
Our file Notre référence
ISBN: 978-0-494-34745-4

Abstract

An FPGA Implementation of the Advanced Encryption Standard with Support for Counter and Feedback Modes

James Steven Grabowski

The Advanced Encryption Standard (AES) is a symmetric key block cipher approved by the National Institute of Standards and Technology (NIST). AES replaced the Data Encryption Standard (DES) as a standard encryption algorithm within the United States government. It is widely used in both software and hardware applications and transactions.

Different confidentiality modes of operation allow a symmetric key block cipher to provide additional data confidentiality by altering the output in respect to previously processed input data. These modes include Cipher Block Chaining, Cipher Feedback, Output Feedback and Counter modes. Electronic Codebook (ECB) mode does not enhance the confidentiality of the original cipher.

This thesis presents an implementation of AES on a field-programmable gate array (FPGA). The design improves upon similar implementations that only employ ECB mode by supporting all five confidentiality modes of operation. The unified design supports all applicable key sizes and offers competitive throughput and resource utilization compared to designs lacking additional confidentiality modes. The design occupies 7452 slices of a Xilinx Virtex-II Pro XC2VP50 and features a maximum clock speed of 56.3*MHz*. Throughputs up to 480.427*Mbps*, 423.906*Mbps* and 379.284*Mbps* for

128-bit, 192-bit and 256-bit keys are produced for all five modes of operation. A straightforward level of key agility allows encryption and decryption operations to proceed uninterrupted at the expense of throughput. This feature is ideal when it is necessary to change the key for each block of data. A physical hardware prototype of the design is employed as further demonstration of the design's functional abilities.

Acknowledgements

To my professor and supervisor Dr. Amr Youssef, thank you for encouraging my interests in information security, for your guidance and infinite patience toward a full-time student with full-time commitments. I hope that the end result lives up to your expectations.

When I took my first course in VHDL, I couldn't fully grasp how it would translate into a real-world application. That didn't change the fact that I enjoyed it immensely. Many thanks to Ted Obuchowicz, M.Eng., Concordia's resident VLSI/CAD Specialist, and my friend. Without you I probably would not have chosen this route, nor would I have all the devices I've used at my disposal. The late night classes may be gone, but Keith is eternal.

To my friend Adam Szporer, B.Eng., and soon to be trapped writing acknowledgements of his own, it has been a long trip. Thanks for sticking with me through the years and experiencing the same highs and lows. The Fish will always be around for another sequel.

To my love Jennifer Ricard, M.A., thank you for being there when times were tough, for your encouragement and for being my second reader. Because of your dedication, I can pretend it's not my fault when someone finds a grammatical oversight.

Thank you, my family, for your support and for the phone calls. I hope you enjoy everything that follows.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
BRAM	Block RAM
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CTR	Counter
DES	Data Encryption Standard
DIP	Dual In-line Package
ECB	Electronic Codebook
FIPS	Federal Information Processing Standards
FPGA	Field-Programmable Gate Array
GCLK	Global Clock
GF	Galois Field
IOB	Input/Output Block
ISE	Integrated Software Environment
IV	Initialisation Vector
JTAG	Joint Test Action Group
LUT	Lookup Table
Mbps	Megabits per second
MHz	Megahertz

MUX	Multiplexer
NIST	National Institute of Standards and Technology
ns	Nanoseconds
OFB	Output Feedback
PCB	Printed Circuit Board
SP	Special Publication
Rcon	Round Constant
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VLSI	Very-Large-Scale Integration
XOR	Exclusive Or

Chapter 1

Introduction

Cryptography is the study of mathematical techniques which support confidentiality, data integrity, entity authentication and data origin authentication [11]. The word *cryptography* in contemporary use often refers to data encryption. *Encryption* is the masking of data using ciphers. A basic *cipher* is a set of mathematical functions which produce an output value from an input value and a key. A *key* is an input to a cipher that behaves as a variable of the cipher's mathematical functions.

An example of a basic (insecure) cipher is the shift cipher. A *shift cipher* will shift the value of an input up or down a fixed amount. The fixed amount is a shift cipher's key. For example, if the entire number space for data values is the ten integers from *0* to *9*, and an individual wants to encrypt the data values *3-5-9-0-4* with a key of *3*, the output of the cipher will be *6-8-2-3-7*.

1.1 Types of Cryptographic Algorithms

Cryptographic algorithms are generally divided into two categories, asymmetric (public key) and symmetric. Diffie and Hellman [12] were the first to publicly propose *asymmetric*, or *public key* algorithms. Each entity that transfers data in a public key system is assigned a pair of keys, a private key and a public key. The entity keeps their private key secret and makes their public key available to other entities. A message encrypted with an entity's public key may be decrypted with their private key and vice versa. For example, Alice can send Bob a message that only Bob can read by encrypting the message with Bob's public key. The message can only be decrypted with Bob's private key, and only Bob possesses this private key. Similarly, Bob can send Alice a message that only Alice can read by encrypting the message with Alice's public key. Information encrypted by a key pair is compromised only if the secrecy of a pair's private key is compromised. Public key encryption systems typically require more calculations (and time) than symmetric key systems. Well-known examples of asymmetric algorithms include the Diffie-Hellman cipher by Diffie and Hellman [12] and the RSA cipher by Rivest, Shamir and Adleman [13].

Symmetric key algorithms normally require the same key for both encryption and decryption operations. All entities that exchange data in a symmetric key system require access to this key. Information encrypted by a key is compromised if the key is compromised. Symmetric key systems are generally faster than public key systems and are ideal for encrypting large volumes of data in small amounts of time. These systems include *block ciphers*, which perform cryptographic operations on fixed input block sizes, or *stream ciphers*, which combine plaintext with values from the cipher's state. Symmetric key systems generally accommodate alternating keys more easily than public key systems. Well-known examples of symmetric key algorithms include the Data Encryption Standard (DES) cipher [14], the Blowfish cipher by Schneier [15] and the Advanced Encryption Standard (AES) cipher [1].

1.2 The Advanced Encryption Standard

AES originated not as a definitive cryptographic algorithm but as a search for a new cryptographic standard. The National Institute of Standards and Technology (NIST) posted a request for comments in January 1997 regarding the selection of a replacement for DES [16]. This was followed up in September 1997 with a request for draft submissions for AES [17]. The minimum requirement was a symmetric key block cipher that supported encryption of 128-bit data blocks using 128-bit, 192-bit or 256-bit keys. Fifteen algorithm submissions were presented at the First AES Conference in August 1998 [18]. Five of the fifteen algorithms were selected as finalists: MARS [19], RC6 [20], Rijndael [21], Serpent [22] and Twofish [23]. Technical analysis of the five finalists was presented at the Third AES Conference in April 2000 [24]. NIST selected a subset of the Rijndael algorithm as the new standard in October 2000 [25]. The standard was ratified in Federal Information Processing Standards Publication 197 (FIPS 197) in November 2001 [1].

1.3 Confidentiality Modes

Several mechanisms exist which improve data confidentiality for symmetric key ciphers. A block cipher typically exhibits a one-to-one relationship between its input and output. The same input block and same key will always produce the same output regardless of the location of an input block in a message. Confidentiality modes allow a block cipher to provide additional data confidentiality by altering the output in respect to previously processed input data. The five fundamental confidentiality modes of operation published by NIST in Special Publication (SP) 800-38A are Electronic

Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR) [2]. Each mode consists of alternative pre-cipher and post-cipher processing algorithms.

1.4 Scope of the Implementation

This dissertation provides a hardware-based solution that combines the full AES standard with user-configurable data confidentiality capabilities. The design improves upon similar implementations that only employ ECB mode by supporting all five confidentiality modes of operation. The additional modes achieve improved data confidentiality by masking the encrypted data in respect to previously encrypted input data. The unified design supports all applicable key sizes and offers competitive throughput and resource utilization compared to designs lacking additional confidentiality modes.

The findings of this dissertation are detailed in four main chapters. Chapter 2 explains the theoretical concepts of AES and the five modes of operation in detail. A brief introduction to Galois field mathematics relevant to AES [1] provides a basis for several AES mathematical manipulations. Chapter 3 outlines this design's hardware implementation of AES through the relationship of the major hardware components constructed in VHSIC Hardware Description Language (VHDL). Chapter 4 builds upon Chapter 3 by enhancing the design with the five confidentiality modes for all AES key sizes and operations. Chapter 5 verifies the correctness of the design using well-publicised test vectors, explores the design's performance, simulation and practical behaviour, and presents a physical prototype implementation.

Chapter 2

Theoretical Background

The implementation of AES in this design is based on the algorithm outlined in FIPS 197 [1]. Likewise the implementation of the five confidentiality modes of operation is based on the algorithms presented in SP 800-38A [2]. This chapter first presents a brief outline of finite field and Galois mathematics to clarify some of the more complex mathematical manipulations in the AES algorithm. Detailed descriptions of the AES algorithm and an explanation of the five confidentiality modes of operation follow.

2.1 Finite Field and Galois Mathematics

A finite field, also referred to as a Galois field (*GF*), contains a finite number of elements [26]. The number of elements in a given *GF* is equal to p^i for any prime number p and any integer i greater than or equal to *1*. Certain AES mathematical manipulations occur in $GF(2^8)$ which by definition contains 256 elements. *Bytes* are regarded as elements of $GF(2^8)$ since they have 256 distinct values. Galois mathematics allows addition and multiplication using a different rule set than standard addition and multiplication.

2.1.1 Addition and Subtraction

The addition of two elements in a *GF* is performed by adding the coefficients of an element's corresponding polynomials and taking the result modulo *2*. This is more simply represented as a XOR operation between the two elements. Subtraction is an identical operation to addition due to the nature of the XOR operation. An example using the elements *{FA}* and *{B4}* follows.

$$\{x^7 + x^6 + x^5 + x^4 + x^3 + x\} + \{x^7 + x^5 + x^4 + x^2\} = \{x^6 + x^3 + x^2 + x\}$$
$$\{11111010\} \oplus \{10110100\} = \{01001110\}$$
$$\{FA\} \oplus \{B4\} = \{4E\}$$

2.1.2 Multiplication

Multiplication in $GF(2^8)$ is equivalent to the multiplication of polynomials modulo a polynomial of the eighth degree whose divisors are only *1* and itself. The AES implementation uses the following irreducible polynomial.

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

This polynomial can be expressed in hexadecimal notation as *{01}{1b}*. An example of modulo multiplication follows, demonstrating how *{57}•{83}={c1}*. Reduction by *mod m(x)* causes the resulting binary polynomial to have a degree less than eight and can be represented as a byte.

$$
\begin{aligned}
(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) &= x^{13} + x^{11} + x^9 + x^7 + x^5 + x^3 + x^2 + x + \\
&\quad x^6 + x^4 + x^2 + x + 1 \\
&= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1
\end{aligned}
$$

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x + 1 \bmod (x^8 + x^4 + x^3 + x + 1) = x^7 + x^6 + 1$$

The multiplicative inverse $b^{-1}(x)$ of any nonzero polynomial $b(x)$ whose degree is less than eight can be expressed as the modulo reduction of a polynomial $a(x)$ by $m(x)$ where $a(x) \bullet b(x) \bmod m(x) = 1$.

$$b^{-1}(x) = a(x) \bmod m(x)$$

2.1.3 Multiplication by 'x' – *xtime*

A byte can be represented as a binary polynomial $b(x)$ whose degree is seven or less. Multiplying $b(x)$ by x (*{02}* in hexadecimal representation) produces the following polynomial.

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

Reducing this polynomial by $\bmod m(x)$ results in $x \bullet b(x)$. If b_7 is *0*, the polynomial is already reduced, but if b_7 is *1*, the reduced form is obtained by XORing the polynomial with $m(x)$. Multiplication by x can be further expressed as the XOR of *{1b}* with the left shift of $b(x)$; x is represented as *{02}* and $m(x)$ as *{01}{1b}* in hexadecimal notation. This operation is referred to as the *xtime* function [1].

$$xtime(\{Byte\}) = \{b_6 b_5 b_4 b_3 b_2 b_1 b_0 0\} \oplus \{1b\}$$

Higher powers of *x* can be represented by additional applications of *xtime*.

$$\{Byte\} \bullet \{02\} = \{Byte\} \bullet x = xtime(\{Byte\})$$
$$\{Byte\} \bullet \{04\} = \{Byte\} \bullet x^2 = xtime(xtime(\{Byte\}))$$
$$\{Byte\} \bullet \{08\} = \{Byte\} \bullet x^3 = xtime(xtime(xtime(\{Byte\})))$$
$$\{Byte\} \bullet \{10\} = \{Byte\} \bullet x^4 = xtime(xtime(xtime(xtime(\{Byte\}))))$$

Such results can determine the multiplication of any two values in $GF(2^8)$. An example, $\{1D\} = \{10\} \oplus \{08\} \oplus \{04\} \oplus \{01\}$, follows.

$$\begin{aligned} \{Byte\} \bullet \{1D\} &= \{Byte\} \bullet (\{10\} \oplus \{08\} \oplus \{04\} \oplus \{01\}) \\ &= \{Byte\} \oplus xtime(xtime(xtime(xtime(\{Byte\})))) \oplus \\ &\quad time(xtime(xtime(\{Byte\}))) \oplus xtime(xtime\{Byte\})) \end{aligned}$$

2.1.4 Addition of Polynomials with Coefficients in $GF(2^8)$

A four-term polynomial $a(x)$ has coefficients where each value $[a_3, a_2, a_1, a_0]$ has a value of a finite field element in $GF(2^8)$. Each value is one byte, and four bytes together form one *word*. The addition of $a(x)$ and a second four-term polynomial $b(x)$ with the same properties as $a(x)$ can be expressed as follows.

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$$

The XOR of the *a* and *b* terms in the above equations then follows the rules set out in Chapter 2.1.1.

2.1.5 Multiplication of Polynomials with Coefficients in *GF(2^8)*

c(x) is produced by the multiplication of two four-term polynomials *a(x)* and *b(x)*. This is written as $a(x) \bullet b(x) = c(x)$. *c(x)* is first expanded to give the following formula and coefficients.

$$c(x) = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$$

$$c_0 = a_0 \bullet b_0$$
$$c_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1$$
$$c_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2$$
$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$
$$c_4 = a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$
$$c_5 = a_3 \bullet b_2 \oplus a_2 \bullet b_3$$
$$c_6 = a_3 \bullet b_3$$

The result *c(x)* is a seven-term polynomial. This *c(x)* must be reduced modulo a polynomial of degree four to become a word. This 4th degree polynomial is (x^4+1) for AES.

A four-term polynomial *d(x)* is the modulo product of *a(x)* and *b(x)* and is written with the following coefficients.

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0$$

$$d_0 = (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3)$$
$$d_1 = (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3)$$
$$d_2 = (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3)$$
$$d_3 = (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3)$$

The above equation can be represented as the following matrix if the polynomial *a(x)* is fixed.

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

The polynomial (x^4+1) is not irreducible in $GF(2^8)$ and as such multiplication by a fixed $a(x)$ is not necessarily invertible. A specific $a(x)$ that does have an inverse is defined for AES as follows.

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$
$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$$

The resulting simplification of the modulo product formulas are ideal for the MixColumns and InvMixColumns of AES and are further explained in Chapters 2.2.3.4 and 2.2.4.4.

2.2 Advanced Encryption Standard

The Advanced Encryption Standard is a block cipher derived from the Rijndael algorithm. Both algorithms support the use of 128-bit, 192-bit and 256-bit keys. AES supports 128-bit block sizes while Rijndael also supports 192-bit and 256-bit block sizes. AES involves two major components, key expansion and encryption/decryption of a data block, or state, over a series of rounds. Each round consists of a series of four operations collectively called the *round function.* Figures 1 and 2 show the round functions for encryption and decryption respectively.

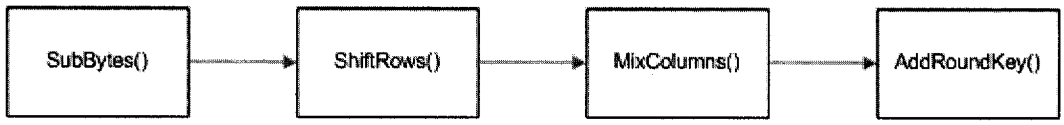

Figure 1: AES Encryption Round Function

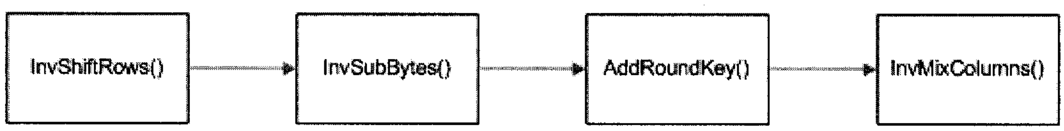

Figure 2: AES Decryption Round Function

Chapter 2.2.1 first describes the functionality of a state. Chapter 2.2.2 outlines the key expansion procedure. Chapters 2.2.3 and 2.2.4 explain the above encryption and decryption round functions in greater detail.

2.2.1 States

The 16 individual bytes composing the data block are referred to as a *state*. The state is processed during each round's encryption or decryption calculations. Figure 3 shows the ordering of the state bytes as identified with numbers 1-16.

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Figure 3: State Byte Ordering (4x4 Matrix)

Figure 4 shows the byte ordering as referred to using row and column indices.

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)
S(2,0)	S(2,1)	S(2,2)	S(2,3)
S(3,0)	S(3,1)	S(3,2)	S(3,3)

Figure 4: State Byte Ordering (4x4 Matrix; Row-Column)

2.2.2 Keys and Key Expansion

The key used for encryption or decryption may be 128-bits, 192-bits or 256-bits in size. The required number of rounds (iterations of the encryption/decryption round function) is 10, 12 and 14 for the respective key sizes. Both encryption and decryption processes require an initial *round key* of identical size to the state being processed plus one round key for each round. Therefore the original key is expanded to (# of rounds + 1) 128-bit keys. This is 11, 13 or 15 keys of 128-bits each, or alternatively 44, 52 or 60 words of 32-bits each.

Figure 5 shows the pseudo-code for the key expansion process. *Nk* is the number of 32-bit words in the key, *Nb* is the number of 32-bit words used in a cipher block (always *4* for AES) and *Nr* is the number of rounds for the associated key size.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp

    i = 0

    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while

    i = Nk

    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Figure 5: Key Expansion Pseudo-Code [1]

The expanded key is constructed as follows:

1. The original key forms the beginning of the expanded key (4, 6 or eight 32-bit words for the three respective key sizes), indexed as subkeys 0-3, 0-5 or 0-7 respectively.
2. For all remaining indices of the expanded key (4-43, 6-51 or 8-59 for the three respective key sizes), store the most recently calculated subkey from the expanded key.
    a. If the index is a multiple of the number of words in the original key, obtain the quotient of the index and the number of words in the original key. Perform a RotWord and SubWord operation on the stored word and XOR it with the Rcon of the quotient.

   b. Otherwise if the original key is 256-bits and the current index is 12, 20, 28, 36, 44 or 52, only perform a SubWord operation on the stored word.
3. XOR the result of Step 2 with the subkey of the expanded key that is one original key length prior to the current index. The final value is the most current subkey in the expanded key. Increment the index by *1*.
4. Repeat steps 2 and 3 until the entire expanded key has been calculated.

2.2.2.1 **RotWord Operation**. The RotWord operation mentioned in Step 2a of Chapter 2.2.2 rotates the first byte of a 32-bit word from the beginning of the word to its end. Figure 6 demonstrates RotWord on a 32-bit word subdivided into four bytes.

(before RotWord)

w_0	w_1	w_2	w_3

(after RotWord)

w_1	w_2	w_3	w_0

Figure 6: RotWord Operation

2.2.2.2 **SubWord Operation**. The SubWord operation mentioned in Steps 2a and 2b of Chapter 2.2.2 performs an S-box substitution on the four separate bytes of a 32-bit word and returns a 32-bit word with the four results ordered in the same relative positions. Figure 7 demonstrates SubWord on a 32-bit word subdivided into four bytes. Chapter 2.2.3.2 provides more information about S-box substitution. One S-box is capable of operating on one byte of data at a time. Four S-boxes are required to perform SubWord as a parallel operation.

(before SubWord)

w_0	w_1	w_2	w_3

(after SubWord)

sbox(w_0)	sbox(w_1)	sbox(w_2)	sbox(w_3)

Figure 7: SubWord Operation

2.2.2.3 **Rcon**. The Rcon mentioned in Step 2a of Chapter 2.2.2 refers to a *round constant* array. Table 1 shows the values of this array.

Table 1: Rcon Array

Index	Value (Hex)
1	01
2	02
3	04
4	08
5	10
6	20
7	40
8	80
9	1B
10	36

2.2.3 Encryption

Encryption may begin once the original key is at least partially expanded. Encryption consists of 10, 12 or 14 rounds depending on whether a 128-bit, 192-bit or 256-bit key is used. Figure 8 shows pseudo-code for the encryption algorithm.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[0, Nb-1]) // See Sec. 5.1.4

    for round = 1 step 1 to Nr-1
        SubBytes(state) // See Sec. 5.1.1
        ShiftRows(state) // See Sec. 5.1.2
        MixColumns(state) // See Sec. 5.1.3
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    out = state
end
```

Figure 8: Encryption Pseudo-Code [1]

The following steps summarise the encryption process:

1. Store the original 128-bit block of data as the state.
2. Perform AddRoundKey on the state using the first round key from the expanded key.
3. For each subsequent round excluding the final round:
    a. Perform SubBytes on the state.
    b. Perform ShiftRows on the state.
    c. Perform MixColumns on the state.
    d. Perform AddRoundKey on the state using the next available round key from the expanded key.
4. Repeat Step 3 for the final round, excluding the MixColumns transformation. The final state is the encrypted output of the AES cipher.

2.2.3.1 **AddRoundKey Operation.** The AddRoundKey operation mentioned in Steps 2 and 3d of Chapter 2.2.3 is a bitwise XOR operation of the given round key and the state, effectively adding the round key to the state.

2.2.3.2 **SubBytes Operation.** The SubBytes operation mentioned in Step 3a of Chapter 2.2.3 performs an S-box substitution on the 16 separate bytes of the 128-bit state. SubBytes returns a new 128-bit state with the 16 results ordered in the same relative positions. Figure 9 demonstrates SubBytes on a 128-bit state. One S-box is capable of operating on one byte of data at a time. Sixteen S-boxes are required to perform SubBytes as a parallel operation.

S(0,0)	S(0,1)	S(0,2)	S(0,3)		S'(0,0)	S'(0,1)	S'(0,2)	S'(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)	→ [S-box] →	S'(1,0)	S'(1,1)	S'(1,2)	S'(1,3)
S(2,0)	S(2,1)	S(2,2)	S(2,3)		S'(2,0)	S'(2,1)	S'(2,2)	S'(2,3)
S(3,0)	S(3,1)	S(3,2)	S(3,3)		S'(3,0)	S'(3,1)	S'(3,2)	S'(3,3)

Figure 9: SubBytes Operation

The S-box is a 256 byte table where the hexadecimal values of the input value reference the position in the table. For example, if the input value is *{57}* in hexadecimal, then the x coordinate of the table is *5*, the y coordinate of the table is *7* and the result is an output value of *{5b}*. The S-box is constructed by taking the multiplicative inverse in $GF(2^8)$ (mapping *{00}* to itself) and applying the following affine transformation over $GF(2)$. b_i is the i^{th} bit of a byte, and c_i is the i^{th} bit of a byte c with the value *{63}*.

$$b'_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i \longrightarrow 0 \le i < 8$$

Table 2 [1] shows the AES S-box.

Table 2: AES S-box [1]

Byte (XY)		Bits 3-0 (Y)															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
Bits 7-4 (X)	**0**	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

2.2.3.3 **ShiftRows Operation**. The ShiftRows operation mentioned in Step 3b of Chapter 2.2.3 performs a forward cyclical shift on each row of the state by *0*, *1*, *2* or *3* bytes for the first, second, third and fourth rows respectively. Figure 10 demonstrates ShiftRows using both the numbering notation and the row/column notation for depicting the state.

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

→ Rotation →

1	5	9	13
6	10	14	2
11	15	3	7
16	4	8	12

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)
S(2,0)	S(2,1)	S(2,2)	S(2,3)
S(3,0)	S(3,1)	S(3,2)	S(3,3)

→ Rotation →

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,1)	S(1,2)	S(1,3)	S(1,0)
S(2,2)	S(2,3)	S(2,0)	S(2,1)
S(3,3)	S(3,0)	S(3,1)	S(3,2)

Figure 10: ShiftRows Operation

2.2.3.4 **MixColumns Operation.** The MixColumns operation mentioned in Step 3c of Chapter 2.2.3 multiplies each column of the state modulo (x^4+1) over $GF(2^8)$ with the polynomial $a(x)$ from Chapter 2.1.5. The new column of the state is presented as the following matrix multiplication.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$0 \le c \le N_b \qquad (N_b = 4)$$

This is reduced to the following calculations per column as an expression of logical XOR operations [1].

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus (\{01\} \bullet s_{2,c}) \oplus (\{01\} \bullet s_{3,c})$$
$$s'_{1,c} = (\{01\} \bullet s_{0,c}) \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus (\{01\} \bullet s_{3,c})$$
$$s'_{2,c} = (\{01\} \bullet s_{0,c}) \oplus (\{01\} \bullet s_{1,c}) \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$
$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus (\{01\} \bullet s_{1,c}) \oplus (\{01\} \bullet s_{2,c}) \oplus (\{02\} \bullet s_{3,c})$$

Multiplication over $GF(2^8)$ can be simplified as an expression of simpler logic functions. Multiplication by *{01}* over $GF(2^8)$ by definition is multiplication by *1*. All terms denoted as $(\{01\}*s_{r,c})$ can be replaced by $(s_{r,c})$.

The function *xtime* in Chapter 2.1.3 shows the multiplication of *{02}* over $GF(2^8)$. A version of *xtime* targeting bit operations in hardware is presented by Zhang and Parhi [3]. *{02}X* can be expressed as the following (*X* is represented by an 8-bit value $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$).

$$\{02\} \bullet a = a_6 a_5 a_4 a_3 a_2 a_1 a_0 0 \oplus 000 a_7 a_7 0 a_7 a_7$$

Multiplication by *{03}* over $GF(2^8)$ can be calculated as the XOR of the value of *{01}* and *{02}*. This is depicted as follows.

$$\{03\} \bullet a = (\{01\} \bullet a) \oplus (\{02\} \bullet a) = a \oplus (\{02\} \bullet a)$$

MixColumns can be developed in a format suitable for the target platform using the above methods for calculating *{02}* and *{03}*.

2.2.4 Decryption

Decryption begins once the original key is at least partially expanded. AES decryption is the inverse of AES encryption. Decryption consists of 10, 12 or 14 rounds depending on whether a 128-bit, 192-bit or 256-bit key is used. Figure 11 shows pseudo-code for decryption.

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4

    for round = Nr-1 step -1 downto 1
        InvShiftRows(state) // See Sec. 5.3.1
        InvSubBytes(state) // See Sec. 5.3.2
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        InvMixColumns(state) // See Sec. 5.3.3
    end for

    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[0, Nb-1])

    out = state
end
```

Figure 11: Decryption Pseudo-Code [1]

The following steps summarise decryption:

1. Store the original 128-bit block of data as the state.
2. Perform AddRoundKey on the state using the last round key from the expanded key.
3. For each subsequent round excluding the final round:
    a. Perform InvShiftRows on the state.
    b. Perform InvSubBytes on the state.
    c. Perform AddRoundKey on the state using the preceding round key from the expanded key.
    d. Perform InvMixColumns on the state.
4. Repeat Step 3 for the final round, excluding the InvMixColumns transformation. The final state is the decrypted output of the AES cipher.

2.2.4.1 **AddRoundKey Operation.** The AddRoundKey operation mentioned in Steps 2 and 3c of Chapter 2.2.4 is a bitwise XOR operation of the given round key and the state, effectively adding the round key to the state. There is no difference between AddRoundKey in encryption and decryption.

2.2.4.2 **InvShiftRows Operation**. The InvShiftRows operation mentioned in Step 3a of Chapter 2.2.4 performs a reverse cyclical shift on each row of the state by *0*, *1*, *2* or *3* bytes for the first, second third and fourth rows respectively. Figure 12 demonstrates InvShiftRows using both the numbering notation and the row/column notation for depicting the state.

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

→ Rotation →

1	5	9	13
14	2	6	10
11	15	3	7
8	12	16	4

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)
S(2,0)	S(2,1)	S(2,2)	S(2,3)
S(3,0)	S(3,1)	S(3,2)	S(3,3)

→ Rotation →

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,3)	S(1,0)	S(1,1)	S(1,2)
S(2,2)	S(2,3)	S(2,0)	S(2,1)
S(3,1)	S(3,2)	S(3,3)	S(3,0)

Figure 12: InvShiftRows Operation

2.2.4.3 **InvSubBytes Operation**. The InvSubBytes operation mentioned in Step 3b of Chapter 2.2.4 performs an Inverse S-box substitution on the 16 separate bytes of the 128-bit state. InvSubBytes returns a new 128-bit state with the 16 results ordered in the same relative positions. Figure 13 demonstrates InvSubBytes on a 128-bit state. One inverse S-box is capable of operating on one byte of data at a time. Sixteen inverse S-boxes are required to perform InvSubBytes as a parallel operation.

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)
S(2,0)	S(2,1)	S(2,2)	S(2,3)
S(3,0)	S(3,1)	S(3,2)	S(3,3)

→ [S-box^{-1}] →

S'(0,0)	S'(0,1)	S'(0,2)	S'(0,3)
S'(1,0)	S'(1,1)	S'(1,2)	S'(1,3)
S'(2,0)	S'(2,1)	S'(2,2)	S'(2,3)
S'(3,0)	S'(3,1)	S'(3,2)	S'(3,3)

Figure 13: InvSubBytes Operation

The inverse S-box is a 256 byte table where the hexadecimal values of the input value reference the position in the table. For example, if the input value is *{57}* in hexadecimal, then the x coordinate of the table is *5*, the y coordinate of the table is *7* and the result is an output value of *{da}*. For the inverse S-box, the inverse of the affine transformation in Chapter 2.2.3.2 is applied and then the multiplicative inverse in $GF(2^8)$ is performed. Table 3 shows the inverse S-box [1].

Table 3: Inverse S-box [1]

Byte (XY)		**Bits 3-0 (Y)**															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
Bits 7-4 (X)	**0**	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

2.2.4.4 **InvMixColumns Operation.** The InvMixColumns operation mentioned in Step 4d of Chapter 2.2.4 multiplies each column of the state modulo (x^4+1) over $GF(2^8)$ with the inverse of the polynomial in MixColumns, $a^{-1}(x)$, from Chapter 2.1.5. The new column of the state is presented as the following matrix multiplication.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$0 \le c \le N_b \qquad (N_b = 4)$$

This is reduced to the following calculations per column [1].

$$
\begin{aligned}
s'_{0,c} &= (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c}) \\
s'_{1,c} &= (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c}) \\
s'_{2,c} &= (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})
\end{aligned}
$$

Multiplication over $GF(2^8)$ can again be simplified as an expression of simpler logic functions. *{01}* and *{02}* were calculated in Chapter 2.2.3.4. *{04}* is calculated as the *xtime* of *{02}* and *{08}* is calculated as the *xtime* of *{04}* as per the following.

$$
\begin{aligned}
\{04\} \bullet a = \quad & \{02\}a_6 \{02\}a_5 \{02\}a_4 \{02\}a_3 \{02\}a_2 \{02\}a_1 \{02\}a_0 0 \oplus \\
& 000\{02\}a_7 \{02\}a_7 0\{02\}a_7 \{02\}a_7 \\
\{08\} \bullet a = \quad & \{04\}a_6 \{04\}a_5 \{04\}a_4 \{04\}a_3 \{04\}a_2 \{04\}a_1 \{04\}a_0 0 \oplus \\
& 000\{04\}a_7 \{04\}a_7 0\{04\}a_7 \{04\}a_7
\end{aligned}
$$

{03} was calculated as the addition (bitwise XOR) of *{02}* and *{01}* in Chapter 2.2.3.4. This method is used to calculate *{09}*, *{0b}*, *{0d}* and *{0e}*.

$$
\begin{aligned}
\{09\} \bullet a &= (\{08\} \bullet a) \oplus (\{01\} \bullet a) = (\{08\} \bullet a) \oplus a \\
\{0b\} \bullet a &= (\{09\} \bullet a) \oplus (\{02\} \bullet a) \\
\{0d\} \bullet a &= (\{09\} \bullet a) \oplus (\{04\} \bullet a) \\
\{0e\} \bullet a &= (\{08\} \bullet a) \oplus (\{04\} \bullet a) \oplus (\{02\} \bullet a)
\end{aligned}
$$

InvMixColumns can be developed in a format suitable for the target platform using these methods for calculating *{09}*, *{0b}*, *{0d}* and *{0e}*.

2.3 Modes of Operation

There are five primary modes of operation defined by SP 800-38A [2] for use with symmetric key block ciphers to provide data confidentiality. These modes are ECB, CBC, CFB, OFB and CTR.

2.3.1 ECB Mode

ECB mode [2] consists of a direct one-to-one relationship between plaintext and ciphertext. For example, if *word1* encrypted by a key results in *word2*, then *word2* decrypted by that same key results in *word1*. Each block of data is operated on independently. The cipher function is applied to a block of plaintext to produce a block of ciphertext and the inverse cipher function is applied to a block of ciphertext to produce a block of plaintext. Figure 14 displays the dataflow for encryption and decryption operations in ECB mode.

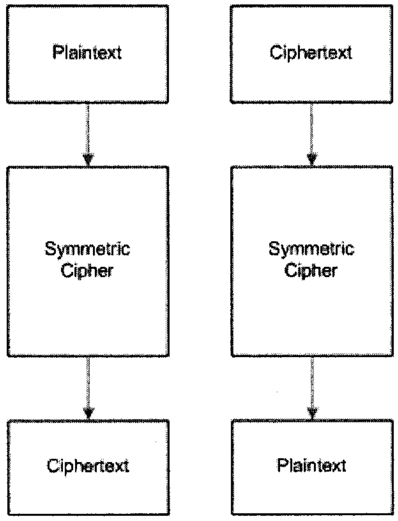

Figure 14: ECB Mode (Encryption and Decryption) [2]

This mode has the advantage of allowing for parallel computation of multiple data blocks as there is no dependency on the order in which the data is computed. This mode's drawback is its one-to-one correlation between ciphertext and plaintext. This consistent relationship may be undesirable in certain applications.

2.3.2 CBC Mode

CBC mode [2] combines the plaintext block with a previous ciphertext block of a CBC calculation. This mode requires the preceding block's ciphertext be calculated to determine the subsequent block's ciphertext, effectively chaining the calculation from one block to the next. Unlike ECB mode, there is no one-to-one correlation between plaintext and ciphertext blocks since the result of a CBC calculation depends on the previous ciphertext value in addition to the current input value. The first calculation in the chain requires a non-secret but unpredictable initialisation vector (IV) in place of a ciphertext block.

Encryption steps in CBC mode XOR together the plaintext and previous step's ciphertext block and input the result into the cipher function. Decryption steps XOR together the previous step's ciphertext block and the output of the inverse cipher function. Figures 15 and 16 show the dataflow for CBC mode encryption and decryption respectively.

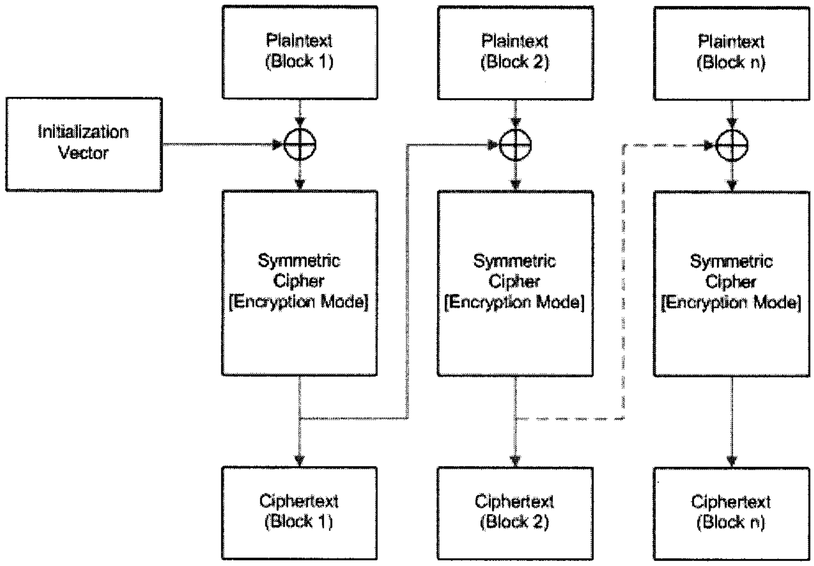

Figure 15: CBC Mode (Encryption) [2]

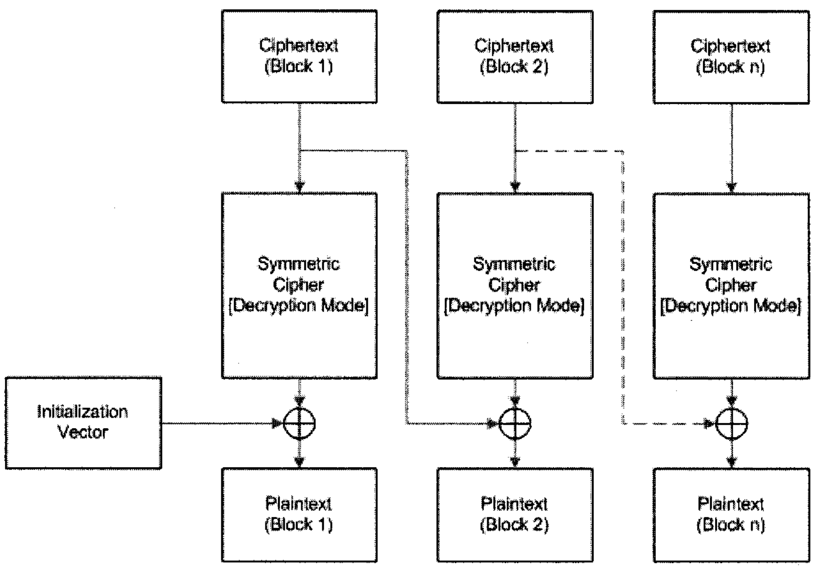

Figure 16: CBC Mode (Decryption) [2]

This mode does not feature the potentially undesirable property of a one-to-one correspondence between plaintext and ciphertext. However, it is not possible to encrypt in parallel due to the chaining property of the encryption process. Decryption can still be performed in parallel presuming the availability of the entire ciphertext stream.

2.3.3 CFB Mode

CFB mode [2] requires the feedback of a calculated ciphertext block as the input to the next calculation. This mode requires calculating the preceding block's ciphertext to determine the subsequent block's ciphertext as with CBC mode. The first calculation requires a non-secret but unpredictable IV in place of a ciphertext block. The block size of the input has *b* bits.

CFB mode encryption XORs together the *s* most significant bits of the cipher function output with the corresponding *s* bits of the plaintext to produce *s* bits of ciphertext output. The remaining *b-s* bits of the input are then concatenated with the *s* bits of ciphertext to produce the next input block. The procedure is repeated until all corresponding *s* bit segments of the plaintext have been processed. CFB mode decryption XORs together the *s* most significant bits of the inverse cipher function output with the corresponding *s* bits of the ciphertext to produce *s* bits of plaintext output. The remaining *b-s* bits of the input are then concatenated with the *s* bits of plaintext to produce the next input block. The procedure is repeated until all corresponding *s* bit segments of the ciphertext have been processed. Figures 17 and 18 show the dataflow for CFB mode encryption and decryption respectively.

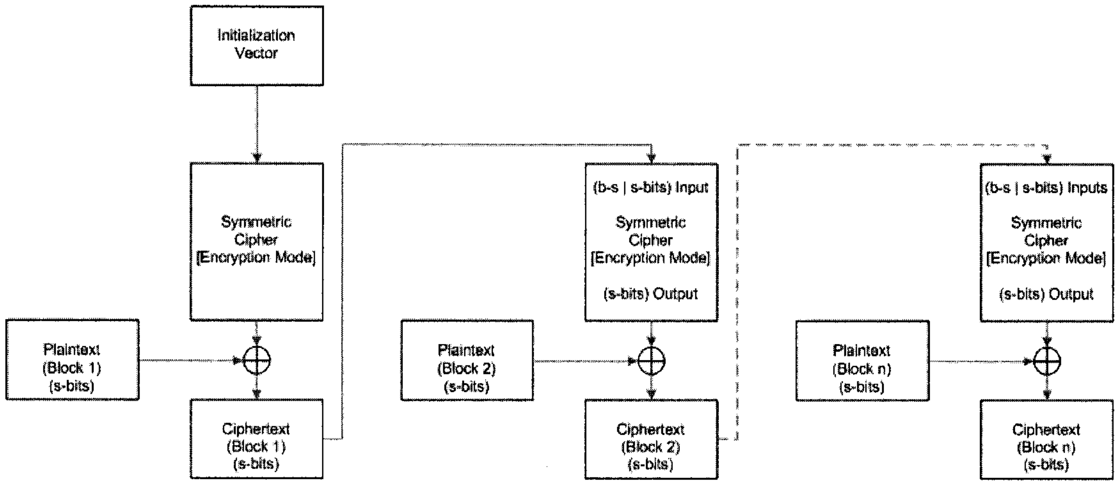

Figure 17: CFB Mode (Encryption) [2]

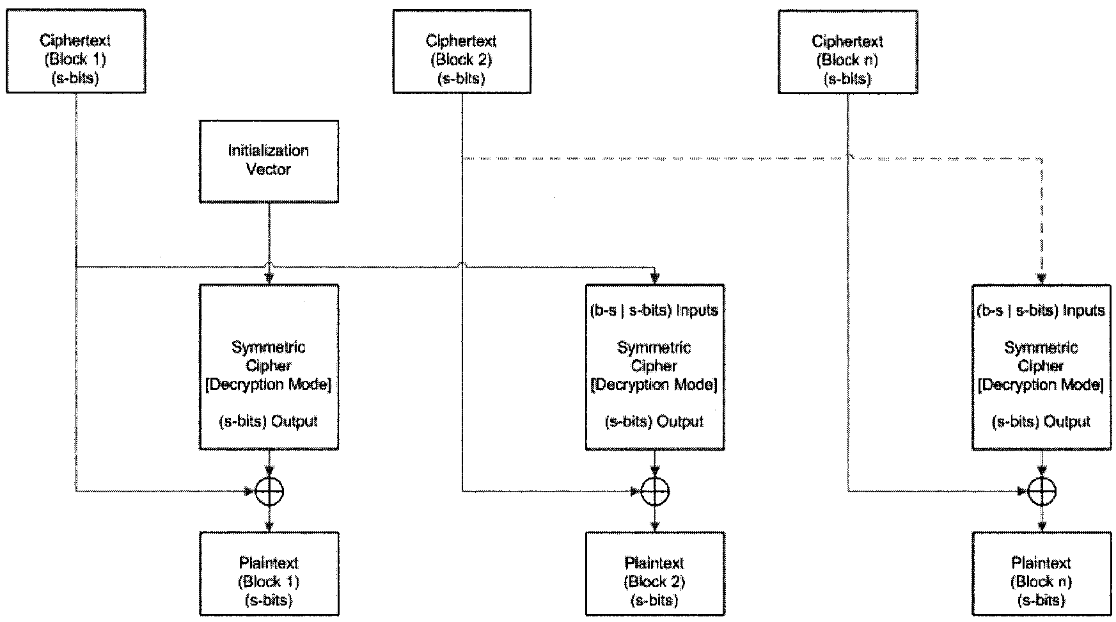

Figure 18: CFB Mode (Decryption) [2]

1-bit, 8-bit, 64-bit and 128-bit CFB modes are all listed in SP 800-38A [2]. Theoretically, any symmetric cipher processing data of b bytes can operate in a CFB mode of s bits where s evenly divides b. For the purposes of this design's hardware implementation, s is a constant 128 bits. In this encryption scenario the entire 128-bit block of plaintext is XORed with the entire cipher function output, producing a 128-bit block of ciphertext to serve as the next input block. The same scenario applies to decryption. The entire 128-bit block of ciphertext is XORed with the entire inverse cipher function output, producing a 128-bit block of plaintext.

This mode is similar to CBC in that it does not feature the potentially undesirable property of a one-to-one ratio between plaintext and ciphertext. Encryption cannot take place in parallel. Decryption can be performed in parallel presuming the availability of the entire ciphertext stream.

2.3.4 OFB Mode

OFB mode [2] requires the output of the forward cipher function for the input of the next calculation. This mode requires calculating the preceding block's ciphertext to determine the subsequent block's ciphertext as with CBC and CFB modes. The first calculation requires the IV as an input block. The IV must be a unique nonce for each key used, otherwise it is possible to compromise data confidentiality.

Encryption and decryption steps in OFB mode are identical, the only difference being the application of plaintext or ciphertext. The output of the forward cipher function is XORed together with the plaintext to produce a ciphertext block for encryption. Similarly, for decryption the output of the forward cipher function is XORed together with the ciphertext to produce a plaintext block. In both cases the output of the forward cipher function also serves as the input of the subsequent calculation. Figures 19 and 20 show the dataflow for OFB mode encryption and decryption respectively.

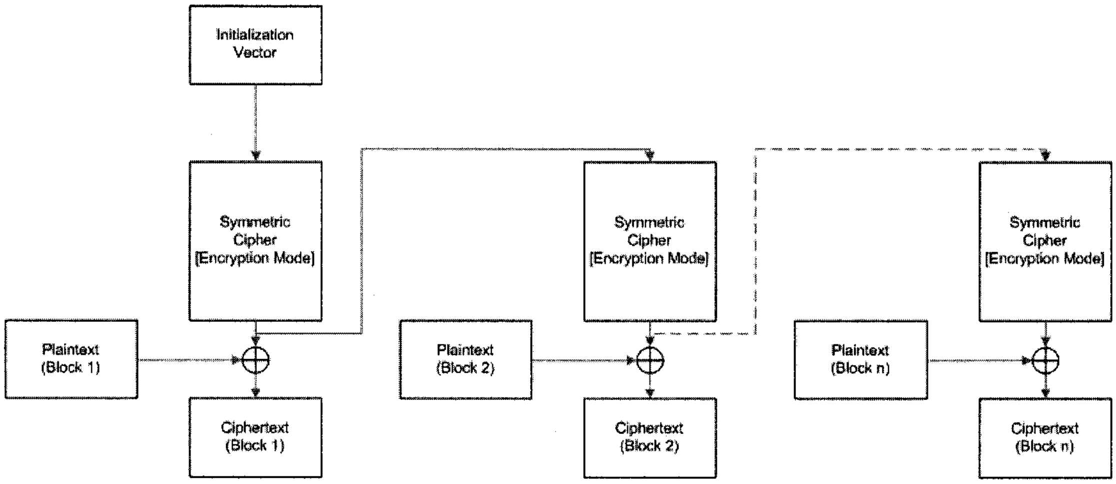

Figure 19: OFB Mode (Encryption) [2]

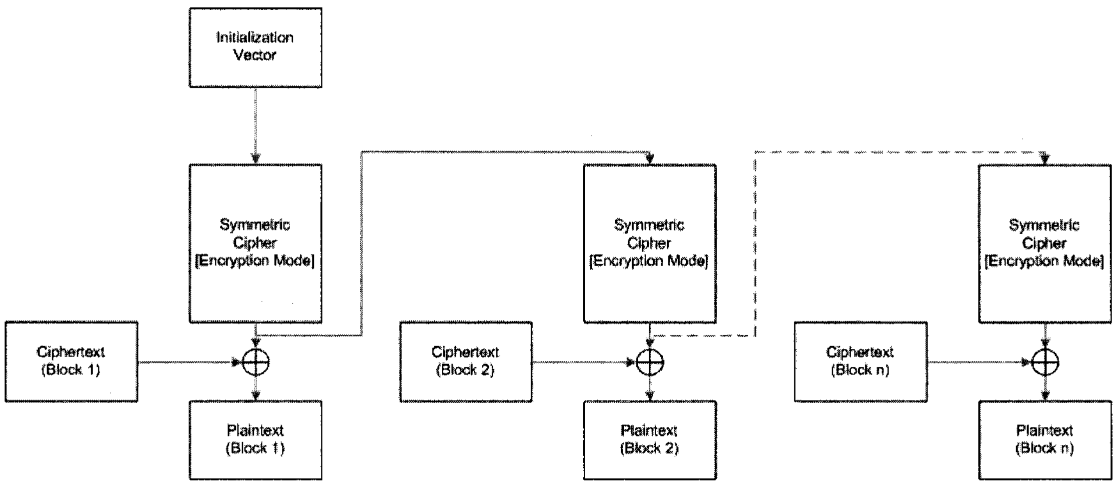

Figure 20: OFB Mode (Decryption) [2]

The same IV supplied with the same key will result in the same forward cipher function outputs for each data block. Encryption and decryption may be performed in parallel if the IV is known and the forward cipher function outputs are calculated in advance. The inverse cipher function is not used in the decryption stage.

2.3.5 CTR Mode

CTR mode [2], similar to OFB mode, only uses the forward cipher function. The IV is replaced by a series of input blocks, called *counters*. These counters must be distinct for each message block. These counters must also be distinct for all messages for a specific key to preserve data confidentiality.

Encryption and decryption steps in CTR mode are identical, the only difference being the application of plaintext or ciphertext. The output of the forward cipher function is XORed together with the plaintext to produce a ciphertext block for encryption. Similarly, for decryption the output of the forward cipher function is XORed together with the ciphertext to produce a plaintext block. In both cases the input of the cipher function consists of unique counters. Identical counters should only be used for the corresponding encryption and decryption stages. Figures 21 and 22 show the dataflow for CTR mode encryption and decryption respectively.

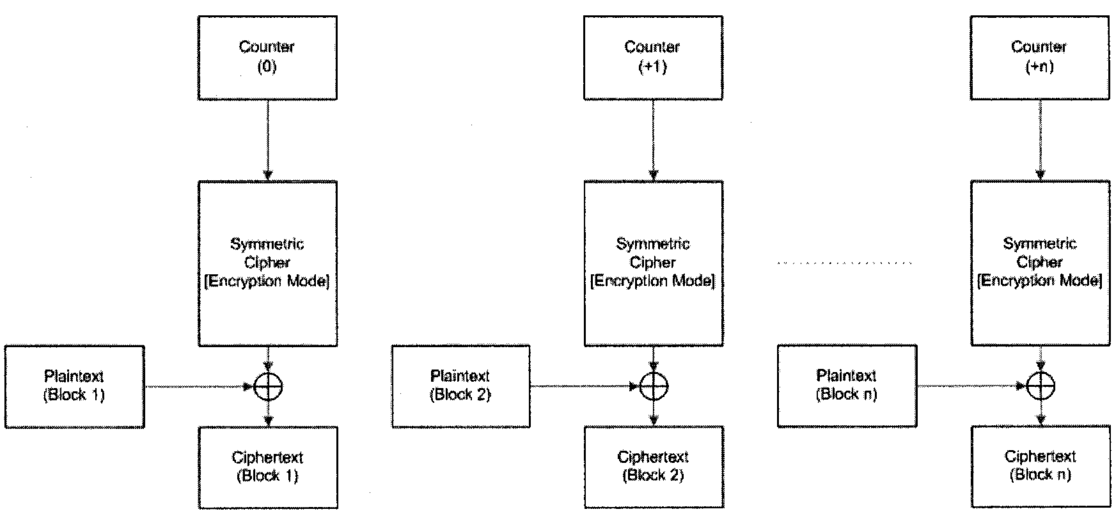

Figure 21: CTR Mode (Encryption) [2]

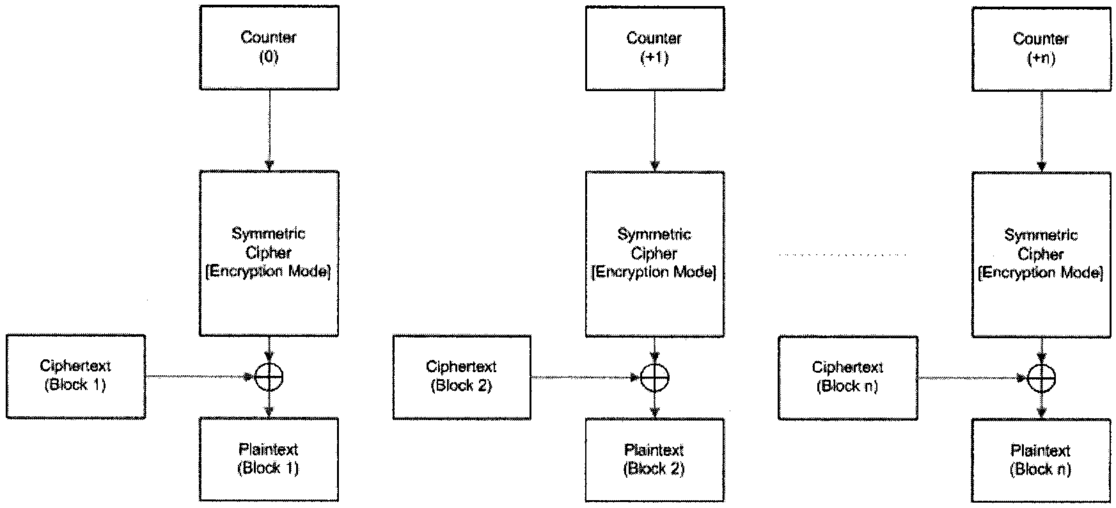

Figure 22: CTR Mode (Decryption) [2]

The same counter supplied with the same key will result in the same forward cipher function outputs for each data block. A counter value from an encryption calculation should only be reused with the same key for the corresponding decryption calculation. Encryption and decryption may be performed in parallel if the counters are known and the forward cipher function outputs are calculated in advance.

The counter function has been made a simple incremental function for the purposes of this design's hardware implementation. Once an initial counter value is supplied the remainder of the counter values are automatically calculated at each stage.

Chapter 3

Core AES Implementation

The core AES implementation is written in VHDL. An overview of the core AES implementation and a description of its major hardware components follow. The chapter first presents a top-level view with a description of how an external user or device interacts with the implementation. This is supplemented with a summary of the implementation's major hardware components and an expanded description of each.

3.1 Top-Level View

The core AES device consists of seven input and two output ports. All inputs and outputs are parallel, comprising a total of 519 bits. Table 4 lists the ports.

Table 4: Core Implementation Ports

Signal	Bits	Type	Description
CLK	1	In	Clock signal; internal processes wake up and calculations are performed when this signal is high.
DataIn	128	In	128-bit data block bus.
KeyIn	256	In	256-bit key block bus (128/192/256-bit sizes).
KeySize	2	In	2-bit input; 3 selectable key sizes.
Enc/Dec	1	In	1-bit input; selectable encryption/decryption.
NewKey	1	In	1-bit input; selectable key expansion operation.
Enable	1	In	1-bit input; processing enable.
DataOut	128	Out	128-bit ciphertext block bus.
OutputReady	1	Out	1-bit data strobe alerts external devices that a new ciphertext block has stabilised on the *DataOut* bus.

The device is operated via the following procedure:

1. Attach an independently functioning clock to the *CLK* output.
2. Set the *Enc/Dec* input to *0* for an encryption operation and *1* for a decryption operation.
3. Set the *NewKey* input to *0* for AES processing only and *1* to expand a new key in addition to AES processing.
4. Set the *KeySize* input to *00* or *01* for a 128-bit key, *10* for a 192-bit key or *11* for a 256-bit key. This input value is irrelevant when *NewKey* is set to *0*.
5. Supply a 256-bit value to the *KeyIn* bus. Bits 1-128 are used for calculations of all key sizes. Bits 129-192 are also used for 192-bit and 256-bit key sizes. Bits 193-256 are used for 256-bit keys only. This input value is irrelevant when *NewKey* is set to *0*.
6. Supply a 128-bit value to the *DataIn* bus.
7. Trigger the *Enable* input. The device produces a signal at the *DataOut* bus corresponding to the stored input values. The signal *OutputReady* is high for one clock cycle at the same time a new stable value appears at *DataOut*. This operation cannot be interrupted until it has been completed. Changing the input values during this process has no effect on the operation.

3.2 Hardware Component View

The design is separated into five major hardware components. Table 5 summarises these components. The remainder of the chapter expands on each of these hardware components.

Table 5: Major Hardware Components

Name	**Description**
PREPROC	Buffers and stores *Enable*, *Enc/Dec*, *DataIn*, *KeyIn*, *KeySize* and *NewKey* signals. Signals stored by PREPROC are used by the other hardware components for an AES processing cycle.
KEYEXP	Performs key expansion based on the *KeySize* and *KeyIn* signals stored by PREPROC. This component is only active when a *1* is stored for *NewKey*.
KEYSTR	Contains the expanded subkeys produced by KEYEXP. The contents of KEYSTR are updated only when a *1* is stored for *NewKey*.
ENCDEC	Performs the encryption/decryption operation based on the *DataIn* signal stored by PREPROC and the stored expanded key contained in KEYSTR. The result is a 128-bit vector.
POSTPROC	Outputs *DataOut* and sets *OutputReady* high during the same clock cycle.

Figure 23 shows the relationship between the hardware components as written in VHDL. At this abstraction level there are six logical constructs which correspond to the five major hardware components. These are *TRIGGER* and *VALUES* (PREPROC), *EXPANSION* (KEYEXP), *MEMORY* (KEYSTR), *ENDEC* (ENCDEC) and *OUTPUT* (POSTPROC).

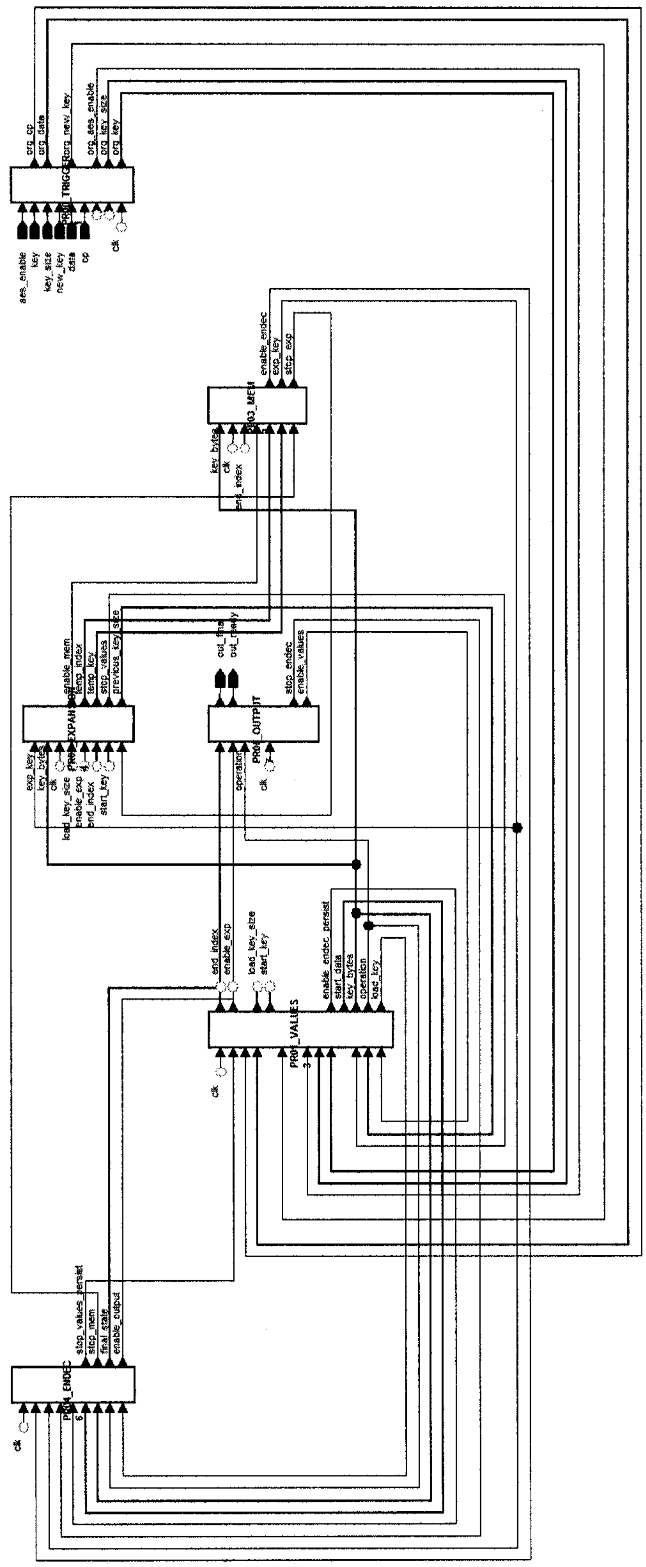

Figure 23: Hardware Component Flowchart (VHDL Code, AES Only)

A central control unit is not listed among either the major hardware components or the VHDL constructs. This is because the major hardware components are each responsible for signalling and micro-managing other components as to when they may operate and what functions they may perform. Therefore, the role of a central control unit is a distributed function of this design. Discussion of any control constructs is intentionally simplified.

3.2.1 PREPROC Component

The PREPROC hardware component buffers and stores the *Enable*, *Enc/Dec*, *DataIn*, *KeyIn*, *KeySize* and *NewKey* signals. The component consists of registers that buffer the value of these signals every clock cycle. The buffered values are referred to in the diagrams as *Buffered* signals, the currently stable input signals. The PREPROC component uses the buffered signals, as opposed to directly accessing the inputs, to avoid signal glitches caused by unstable or rapidly changing inputs.

Figure 24 shows PREPROC's hardware design for manipulating *Buffered* signals.

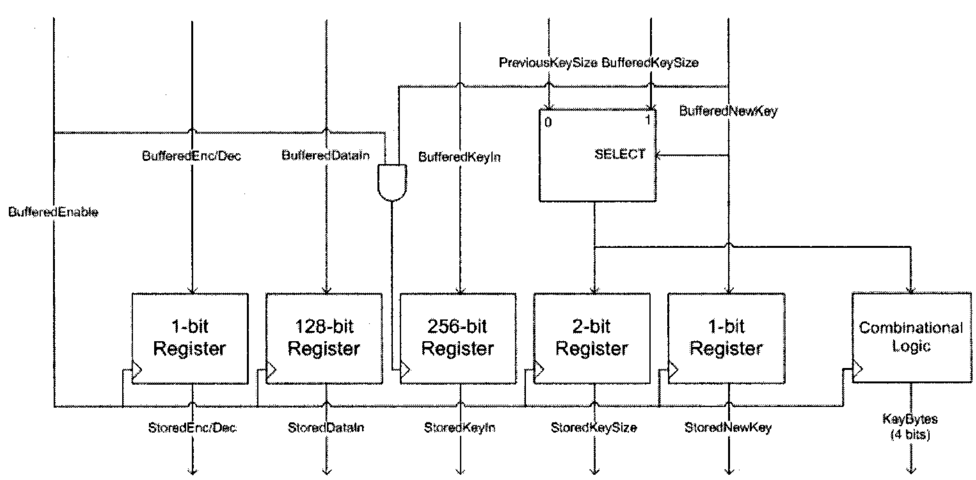

Figure 24: PREPROC Component (AES Only)

The majority of signals stored by PREPROC are referred to in the diagrams as the *Stored* signals, the stable input signals that have been selected for AES processing. While the *Enc/Dec*, *DataIn* and *NewKey* values are stored in a straightforward manner, the remaining signals require additional processing.

The stored *KeySize* depends on whether a new key has been stored or the previously expanded key has been retained. The previous key size, recorded by the KEYEXP component, is stored for *KeySize* if *0* is buffered for *NewKey*. The buffered *KeyIn* is disregarded in this case. The buffered *KeySize* and *KeyIn* are stored if *1* is buffered for *NewKey*. The value *KeyBytes* is calculated based on the output of the MUX. *KeyBytes* corresponds to the number of bytes in the input key, either four (128-bit), six (192-bit) or eight (256-bit). This values is used by the KEYEXP and KEYSTR components.

3.2.2 KEYEXP Component

The KEYEXP hardware component performs key expansion based on the *KeySize* and *KeyIn* signals stored by PREPROC. This component is only active when a *1* is stored for *NewKey*. One 32-bit subkey is created per clock cycle, requiring 44 clock cycles for a 128-bit key, 52 clock cycles for a 192-bit key and 60 clock cycles for a 256-bit key.

The first portion of the expanded key is a copy of the original key. The relevant bits of the stored *KeyIn* are temporarily stored in 32-bit increments. Each subkey in turn is stored by the KEYSTR component. Figure 25 shows KEYEXP's hardware design for all rounds after the original key has been copied.

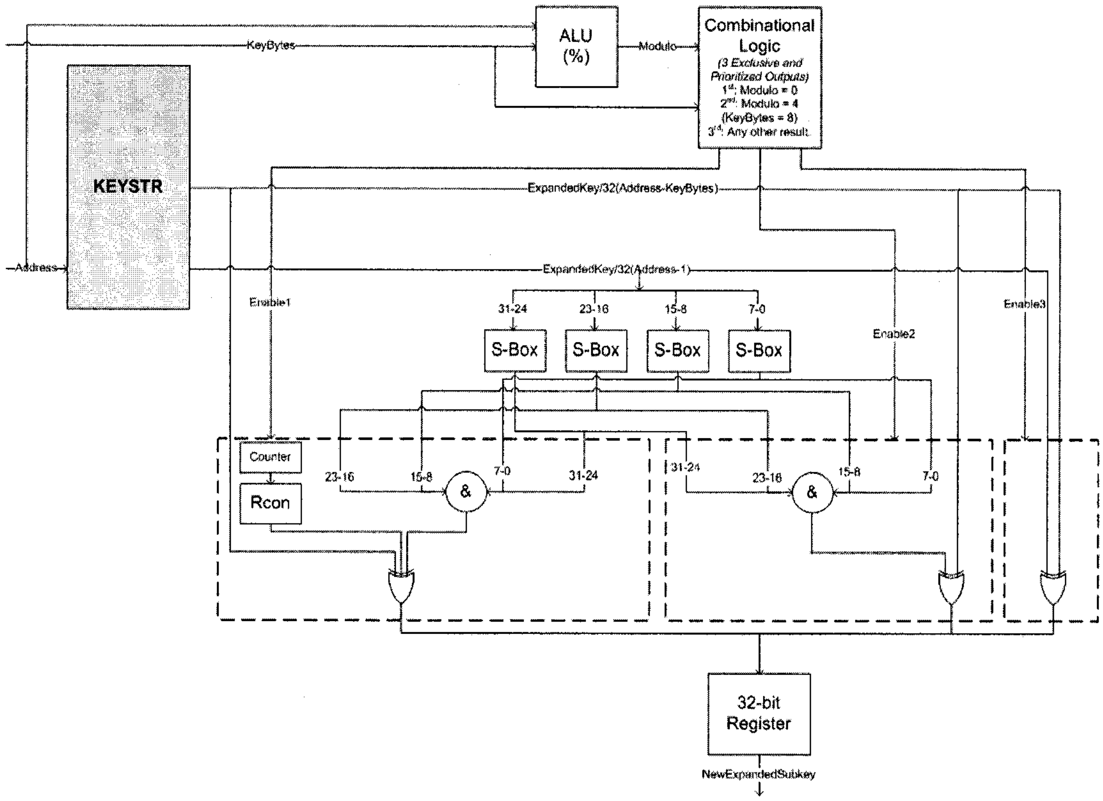

Figure 25: KEYEXP Component (Normal Rounds)

For each key expansion round, the previous subkey and the subkey one *KeyBytes* length away from the current subkey are read. The key expansion algorithm varies based on the value of *KeyBytes* and the modulus of the current subkey index and *KeyBytes*. Three mutually exclusive paths accommodate this in hardware. If the resulting modulo operation is *0* the first enable signal is active, otherwise if the resulting modulo operation is *4* for a 256-bit key (*KeyBytes* is *8*) the second enable signal is active. If both these conditions fail the third enable signal is active.

The first and second paths separate the previous subkey into four 8-bit units that are each processed by an S-box.

The first path concatenates the 8-bit outputs of each S-box in the order of bits *23-16*, *15-8*, *7-0* and *31-24*. This value is XORed with the Rcon value and the subkey one *KeyBytes* length away from the current subkey. The counter attached to the Rcon look-up table (LUT) is the LUT's index and increments each time the first enable signal changes from low to high.

The second path concatenates the 8-bit outputs of each S-box in their original order. This value is XORed with the subkey one *KeyBytes* length away from the current subkey.

The third path concatenates the previous subkey and the subkey one *KeyBytes* length away from the current subkey.

KEYEXP records the value of the stored key for the next AES processing cycle in the event *NewKey* is stored low. This storage is not explicitly depicted in the above figure.

3.2.3 KEYSTR Component

The KEYSTR hardware component contains the expanded subkeys produced by KEYEXP. The contents of KEYSTR are updated only when a *1* is stored for *NewKey*. Figure 32 shows KEYSTR's hardware design. The component consists of 60 registers of 32-bits each. The first 44 are used for an expanded 128-bit key. The next eight are also used for an expanded 192-bit key and the remaining eight are also used for an expanded 256-bit key. Each register is updated with a corresponding subkey after that subkey is calculated by KEYEXP.

3.2.4 ENCDEC Component

The ENCDEC hardware component performs the encryption/decryption operation based on the *DataIn* signal stored by PREPROC and the stored expanded key contained in KEYSTR. The result is a 128-bit vector. The encryption round function follows the pattern of SubBytes(), ShiftRows(), MixColumns() and AddRoundKey(). The decryption round function follows the pattern of InvShiftRows(), InvSubBytes(), AddRoundKey() and InvMixColumns(). First, the hardware required for initialising the state and the hardware required for the major components of an encryption round are presented. Second, the hardware required for initialising the state and the hardware required for the major components of a decryption round are presented.

3.2.4.1 **Encryption Operation.** The first four subkeys of the expanded key are concatenated and XORed with the stored *DataIn,* producing the initial *State.* The encryption round function performs all calculations starting from the initial *State*.

The encryption round function is divided into three hardware components: a SubBytes/ShiftRows unit, a MixColumns unit and an AddRoundKey unit. The output of AddRoundKey is the new *State* and is the input for the next iteration of the round function. The round function iterates 10 times for a 128-bit key, 12 times for a 192-bit key and 14 times for a 256-bit key. Figure 26 shows ENCDEC's SubBytes/ShiftRows encryption unit.

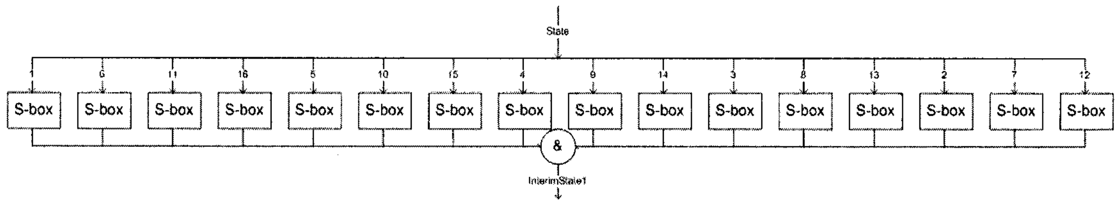

Figure 26: ENCDEC Component (Encryption / SubBytes & ShiftRows)

The functionality of the SubBytes and ShiftRows operations are merged into a single hardware unit. This takes advantage of the reversible property of SubBytes and ShiftRows. The *State* is separated into sixteen 8-bit units and reordered as in Chapter 2.2.3.3. Each unit is applied to an S-box that generates a corresponding value. The resulting values are concatenated together producing an interim state. This completes the SubBytes and ShiftRows operations of the round function. This hardware unit is active for all rounds.

Figure 27 shows ENCDEC's *XTime* function as presented by Zhang and Parhi [3]. The XTime function was presented in Chapter 2.1.3 and greatly simplifies the MixColumns and InvMixColumns *GF* operations. Any appearances of *XTime* blocks hereon can be replaced by this implementation. Figure 28 shows ENCDEC's MixColumns encryption unit.

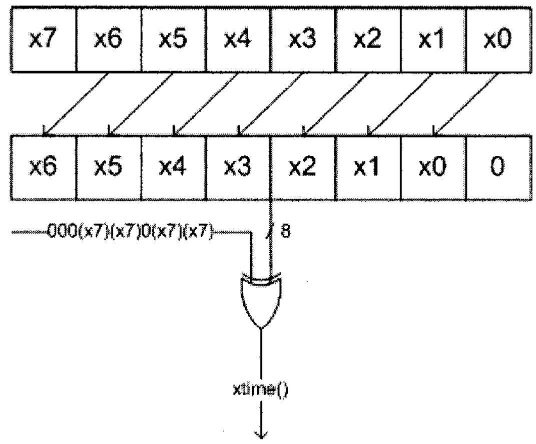

Figure 27: ENCDEC Component (XTime)

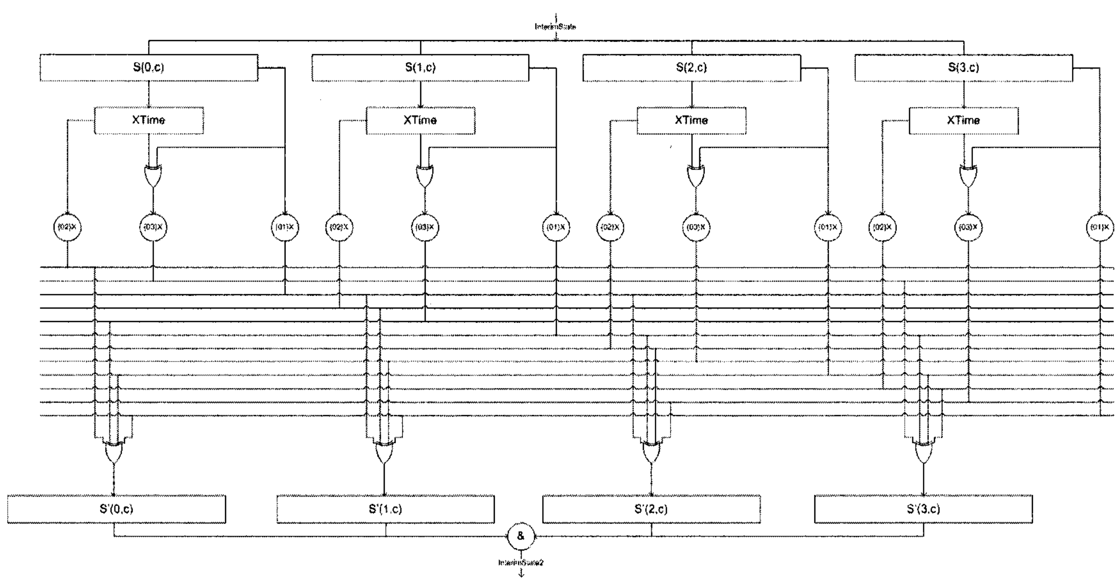

Figure 28: ENCDEC Component (Encryption / MixColumns)

The MixColumns unit is based off an implementation proposal by Zhang and Parhi [3]. The interim state is separated into four 32-bit words corresponding to the four columns of the state. Each column in turn is separated into four 8-bit subwords. An *XTime* unit is applied to the initial subword to produce *{02}X*. A subword's *{02}X* and the original subword are XORed to produce *{03}X*. *{03}X*, *{02}X*, *{01}X* and *{01}X* values are XORed together as in Chapter 2.2.3.4 to produce the transformed subwords. This is repeated for the other three columns of the state. The transformed subwords are concatenated together producing a second interim state value. This completes the MixColumns operation of the round function. This hardware unit is active for all but the final round.

The AddRoundKey unit is similar to the initialisation unit. The second interim state is XORed with the next four unprocessed subkeys of the expanded key, producing the new *State*. This completes the AddRoundKey operation of the round function, and the current round. This hardware unit is active for all rounds. The *State* produced in the final round is the AES ciphertext value for the expanded key and stored *DataIn*.

3.2.4.2 **Decryption Operation.** The last four subkeys of the expanded key are concatenated and XORed with the stored *DataIn*, producing the initial *State*. The decryption round function performs all calculations starting from the initial *State*.

The decryption round function is divided into three hardware components: an InvShiftRows/InvSubBytes unit, an AddRoundKey unit and an InvMixColumns unit. The output of InvMixColumns is the new *State* and is the input to the next iteration of the round function. The round function iterates 10 times for a 128-bit key, 12 times for a 192-bit key and 14 times for a 256-bit key.

Figure 29 shows ENCDEC's InvShiftRows/InvSubBytes decryption unit.

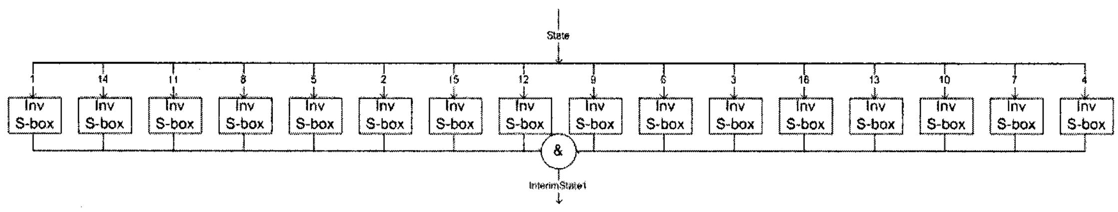

Figure 29: ENCDEC Component (Decryption / InvShiftRows & InvSubBytes)

The functionality of the InvShiftRows and InvSubBytes operations are merged into a single hardware unit as with SubBytes and ShiftRows. The *State* is separated into sixteen 8-bit units and reordered as in Chapter 2.2.4.2. Each unit is applied to an inverse S-box that generates a corresponding value. The resulting values are concatenated together producing an interim state. This completes the InvSubBytes and InvShiftRows operations of the round function. This hardware unit is active for all rounds.

The AddRoundKey unit is similar to the initialisation unit. The interim state is XORed with the next group of the last four unprocessed subkeys of the expanded key, producing a second interim state. This completes the AddRoundKey operation of the round function. This hardware unit is active for all rounds. The interim state produced in the final round is the AES ciphertext value for the expanded key and stored *DataIn*. The InvMixColumns operation is bypassed.

Figure 30 shows ENCDEC's InvMixColumns decryption unit.

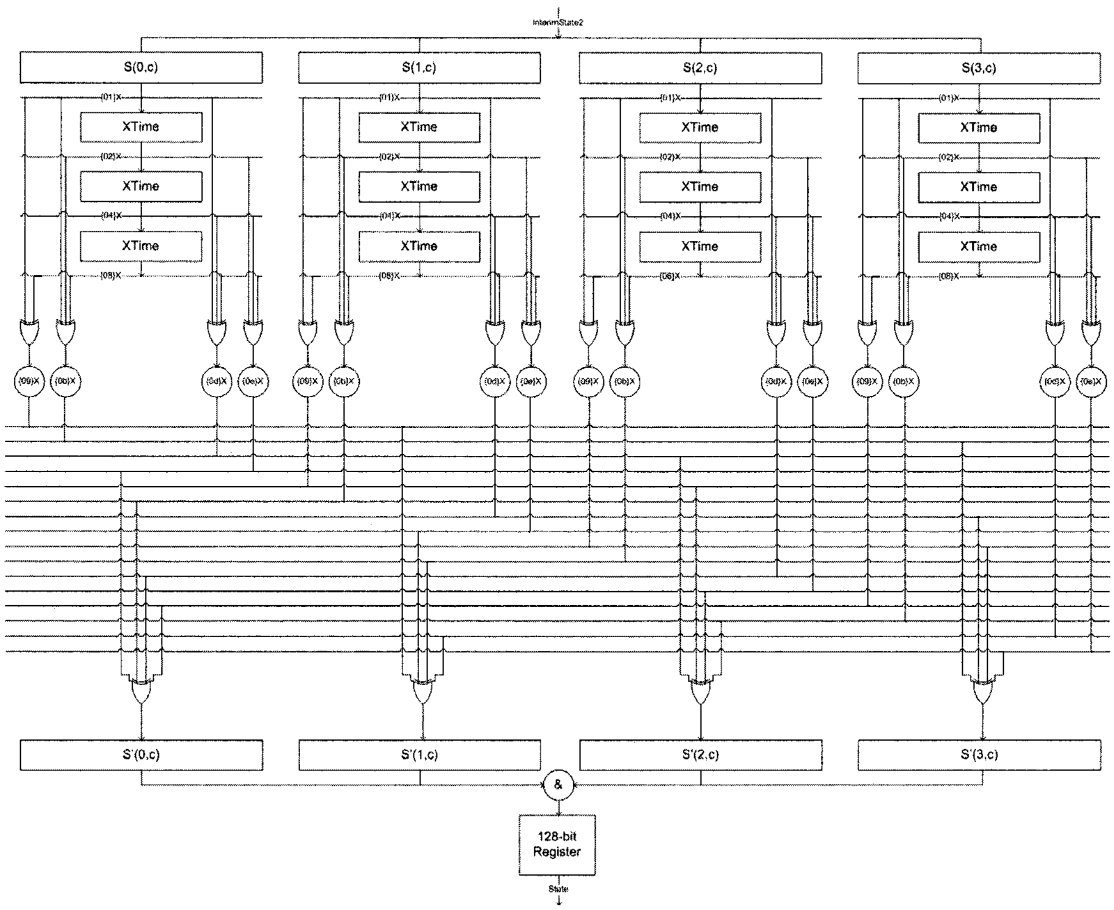

Figure 30: ENCDEC Component (Decryption / InvColumns)

The InvMixColumns unit is based off an implementation proposal by Zhang and Parhi [3]. The second interim state is separated into four 32-bit words corresponding to the four columns of the state. Each column in turn is separated into four 8-bit subwords. An *XTime* unit is applied once, twice and thrice to produce a subword's *{02}X*, *{04}X* and *{08}X*. A subword's *{09}X*, *{0b}X*, *{0d}X* and *{0e}X* are produced by XORing corresponding combinations of the subword and its *{02}X*, *{04}X* and *{08}X*. *{09}X*, *{0b}X*, *{0d}X* and *{0e}X* are XORed together as in Chapter 2.2.4.4 to produce the transformed subwords. This is repeated for the other three columns of the state. The transformed subwords are concatenated together producing the new *State*. This completes the InvMixColumns operation of the round function. This hardware unit is active for all but the final round.

3.2.5 POSTPROC Component

The POSTPROC hardware component outputs *DataOut* and sets *OutputReady* high during the same clock cycle. The component consists of a 128-bit register and a 1-bit register corresponding to *DataOut* and *OutputReady* respectively. The final value of *State* is supplied to a 128-bit register when the ENCDEC component completes an encryption or decryption operation. Simultaneously, a high signal is driven to the 1-bit register. Each register stores its respective signal, producing *DataOut* and *OutputReady* respectively. After one clock cycle a low signal is driven to the 1-bit register. This value is stored, producing a low *OutputReady*.

Chapter 4

Confidentiality Modes

ECB is an inherent part of basic AES operation. Several modifications to the base design are required to add support for CBC, CFB, OFB and CTR operating modes. The overall system requires four additional bits of input versus the base AES implementation. One new major hardware component is added and two existing components are modified. The chapter first presents a top-level view with a description of how an external user or device interacts with the updated implementation. This is supplemented with a summary of the implementation's additional and modified major hardware components and an expanded description of each.

4.1 Top-Level View

The complete AES device with five modes consists of nine input and two output ports. All inputs and outputs are parallel, comprising a total of 523 bits. Table 6 lists the additional ports.

Table 6: Additional Implementation Ports

Signal	Bits	Type	Description
Mode	3	In	3-bit input; 5 selectable operating modes.
LoadIV	1	In	1-bit input; selectable IV storage / regular operation.

The device operates in a fundamentally identical manner to the base AES design:

1. Attach an independently functioning clock to the *CLK* output.
2. Set the *Enc/Dec* input to *0* for an encryption operation and *1* for a decryption operation.
3. Set the *NewKey* input to *0* for AES/mode processing only and *1* to expand a new key in addition to AES/mode processing.
4. Set the *KeySize* input to *00* or *01* for a 128-bit key, *10* for a 192-bit key and *11* for a 256-bit key. This input value is irrelevant when *NewKey* is set to *0*.
5. Supply a 256-bit value to the *KeyIn* bus. Bits 1-128 are used for calculations of all key sizes. Bits 129-192 are also used for 192-bit and 256-bit key sizes. Bits 193-256 are used for 256-bit keys only. This input value is irrelevant when *NewKey* is set to *0*.
6. Supply a 128-bit value to the *DataIn* bus.
7. Set the *Mode* input to *000* for ECB mode, *001* for CBC mode, *010* for CFB mode, *011* for OFB mode or *1xx* for CTR mode (where *x* can be *0* or *1*).
8. Set the *LoadIV* input to *0* for regular operation and *1* for storing an IV from the *DataIn* bus input.
9. Trigger the *Enable* input.
    a. The device sets the IV as the stored *DataIn* if the stored *LoadIV* is *1*.
    b. Otherwise if the stored *LoadIV* is *0* the device produces a signal at the *DataOut* bus corresponding to the stored input values. The signal *OutputReady* is high for one clock cycle at the same time a new stable value appears at *DataOut*. This operation cannot be interrupted until it has

been completed. Changing the input values during this process has no effect on the operation.

4.2 Hardware Component View

The complete design is separated into six major hardware components. One component is added and adjustments are made to two existing components. Table 7 summarises the additional and adjusted components. The remainder of the chapter expands on each of these hardware components.

Table 7: New and Updated Major Hardware Components

Name	**Description**
IVREG	Stores either a user-supplied or calculated IV.
PREPROC	Buffers and stores *Enable*, *Enc/Dec*, *DataIn*, *KeyIn*, *KeySize*, *NewKey*, *Mode* and *LoadIV* signals. Pre-processes values required for modes other than ECB. Signals stored by PREPROC are used by the other hardware components for an AES processing cycle.
POSTPROC	Post-processes values required for modes other than ECB. Calculates and outputs *DataOut* and sets *OutputReady* high during the same clock cycle.

Figure 31 shows the relationship between all hardware components as written in VHDL. At this abstraction level there are seven logical constructs which correspond to the six major hardware components. These constructs are identical to the base design with the addition of *IV* (IVREG).

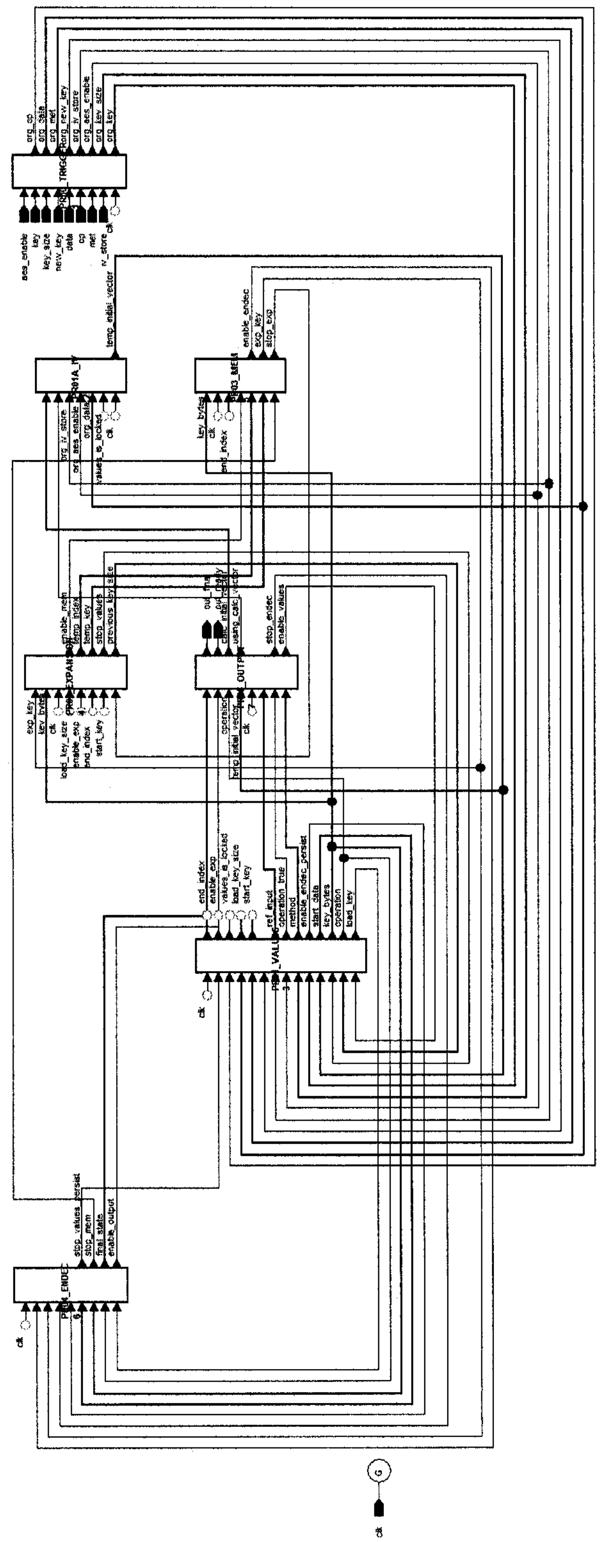

Figure 31: Hardware Component Flowchart (VHDL Code, Complete)

The role of a central control unit remains a distributed function of this design. Discussion of any control constructs is intentionally simplified.

4.2.1 IVREG Component

The IVREG hardware component stores either a user-supplied or calculated IV. Figure 32 shows IVREG's hardware design.

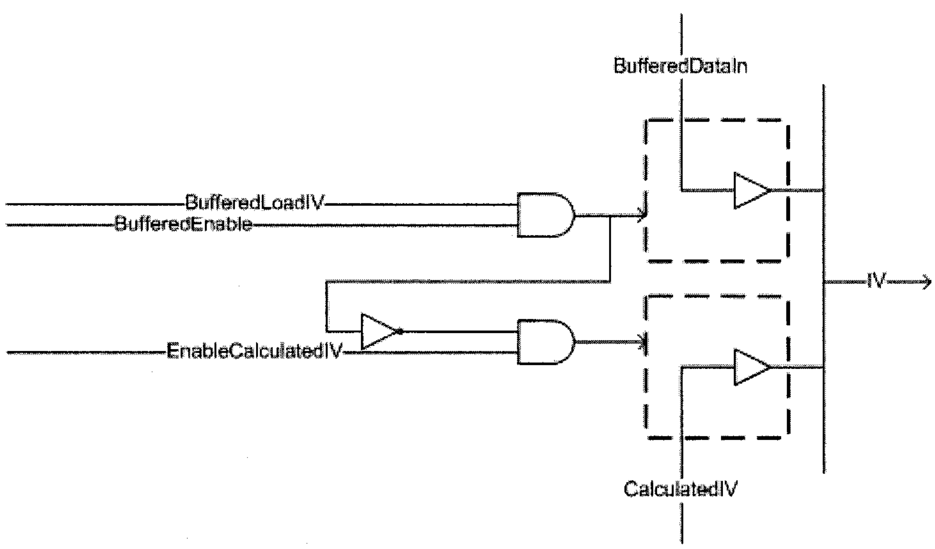

Figure 32: IVREG Component

The component consists of combinational logic that determines which of two values are assigned to the internal *IV* bus. The user stores the buffered *DataIn* when the buffered *LoadIV* and *Enable* signals are high. Alternatively, the updated POSTPROC component calculates a new IV corresponding to the mode of operation, and notifies IVREG when this value is ready. This calculated IV is stored on the *IV* bus when the notification signal is high and either condition for storing the buffered *DataIn* is not met.

Storing the buffered *DataIn* is disabled during key expansion, encryption/decryption or output procedures.

4.2.2 PREPROC Component

The PREPROC hardware component buffers and stores the *Enable*, *Enc/Dec*, *DataIn*, *KeyIn*, *KeySize*, *NewKey*, *Mode* and *LoadIV* signals. The component is functionally identical to the implementation in Chapter 3.2.1. An additional 3-bit and 1-bit register are added to buffer the *Mode* and *LoadIV* inputs. *LoadIV* is used exclusively by the IVREG component.

PREPROC also pre-processes values required for modes other than ECB. Figure 33 shows PREPROC's updated hardware design for manipulating *Buffered* signals.

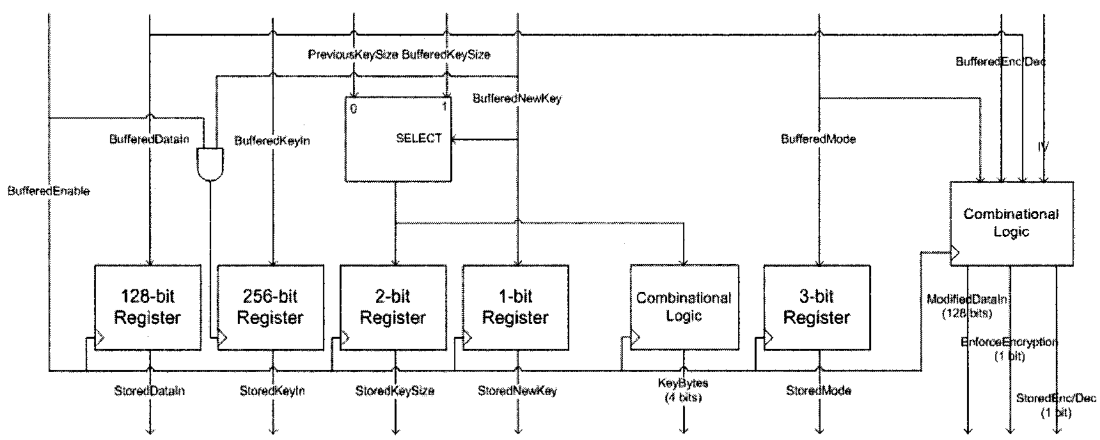

Figure 33: PREPROC Component (Complete)

The component contains additional combinational logic and a new 3-bit register. The *Mode* value is stored in a straightforward manner. There is a distinction between the stored *DataIn* and the data block created by pre-processing and supplied to ENCDEC.

This is important for all modes except ECB which manipulate the input block before processing. Likewise the stored *Enc/Dec* and the operation supplied to ENCDEC are differentiated. This is used by CFB, OFB and CTR modes which always perform an encryption operation during AES processing, but have different post-processing procedures based on the original *Enc/Dec*.

The behaviour of the combinational logic differs based on the buffered *Mode* and *Enc/Dec*. The buffered *DataIn* is stored when the buffered *Mode* corresponds to ECB mode, or alternatively CBC mode and the buffered *Enc/Dec* is set to decryption. The XOR of the buffered *DataIn* and the value on the *IV* bus are stored as the input data to ENCDEC when the buffered *Mode* corresponds to CBC mode and the buffered *Enc/Dec* is set to encryption. The value on the *IV* bus is stored as the input data to ENCDEC and encryption is enforced when the buffered *Mode* corresponds to CFB, OFB and CTR modes.

4.2.3 POSTPROC Component

The POSTPROC hardware component post-processes values required for modes other than ECB. It calculates and outputs *DataOut* and sets *OutputReady* high during the same clock cycle.

Production of *DataOut* and *OutputReady* is similar to that in Chapter 3.2.5, only the *State* value received from ENCDEC is post-processed to create the correct ciphertext for the stored *Mode*. POSTPROC also calculates an IV, based on the stored *Mode*, that is read and stored by IVREG when the encryption key is retained.

ECB mode is identical to the base AES implementation and results in no alteration of the *State* value. The calculated IV is set to *0* since *IV* is not used during any ECB pre-processing or post-processing.

Figure 34 shows the post-processing logic for CBC mode.

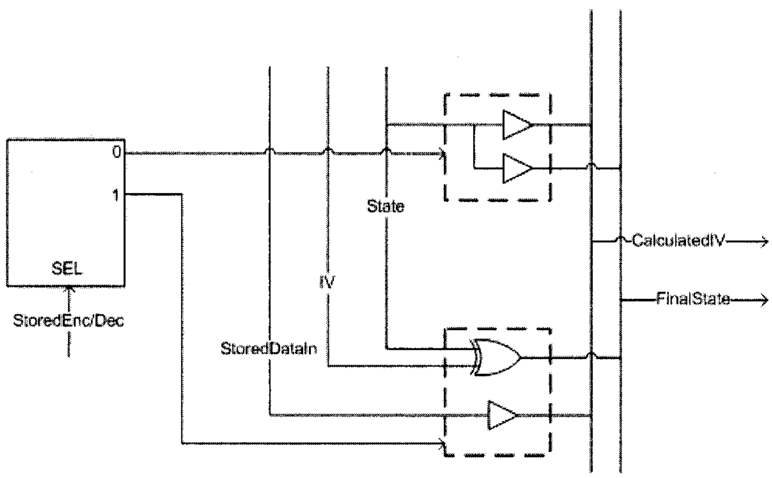

Figure 34: POSTPROC Component (CBC Post-Processing)

State is assigned to the calculated IV and directly supplied to *DataOut* when encryption is stored as the *Enc/Dec* value for CBC mode. This result is significantly different following a decryption operation. The value supplied to *DataOut* is the XOR of the *State* and the *IV* bus, and the stored *DataIn* is becomes the calculated IV.

Figure 35 shows the post-processing logic for CFB mode.

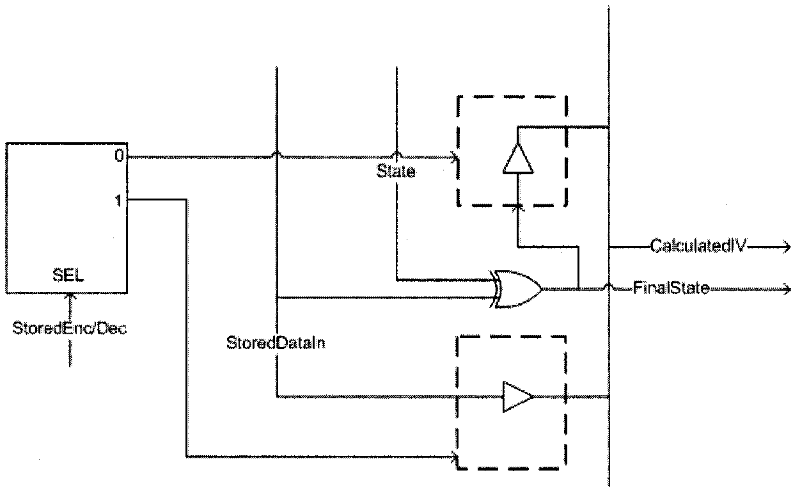

Figure 35: POSTPROC Component (CFB Post-Processing)

The value supplied to *DataOut* in all CBC configurations is the XOR of the *State* and the stored *DataIn*. The calculated IV is identical to the value supplied to *DataOut* after an encryption operation, and is equivalent to the stored *DataIn* after a decryption operation.

OFB and CTR mode post-processing functions are less complex as their behaviour is identical whether the stored *Enc/Dec* corresponds to encryption or decryption. The value supplied to *DataOut* is the XOR of the *State* and the stored *DataIn* for both OFB and CTR modes. The calculated IV is identical to the *State* in OFB mode and is the value on the *IV* bus incremented by *1* in CTR mode.

Chapter 5

Hardware Implementation

The targeted hardware platform was the Xilinx Virtex-II Pro XC2VP50-7FF1152 using Xilinx ISE 8.1i as the Synthesis, Translation, Mapping, Place & Route and Program File Generation tool. Simulations were conducted in Mentor Graphics ModelSim SE using the Post-Place & Route simulation model generated from the VHDL code. This ensures performance representative of the actual FPGA.

The chapter first presents the simulation results from key expansion. The simulation results of AES encryption and decryption in ECB mode follow, including interim state values. The chapter next presents the simulation results of AES encryption and decryption for CBC, CFB, OFB and CTR modes. A summary of the hardware resources allocated after synthesis, timing diagrams based on simulation inputs and a summary of the actual throughput performance of the design follow. The chapter ends by comparing the performance of this design to similar published designs, and detailing a working hardware prototype built to support and test the design.

5.1 Key Expansion Verification

Table 8 lists the 32-bit subkeys generated in simulation during the expansion of a 128-bit key. The 44 subkeys are sequentially ordered from 59 to 16.

Table 8: AES Key Expansion (128-bit) – Fourty-four 32-bit Subkeys

Key: 2b7e151628aed2a6abf7158809cf4f3c							
#	**Subkey**	#	**Subkey**	#	**Subkey**	#	**Subkey**
59	2b7e1516	48	7359f67f	37	caf2b8bc	26	b58dbad2
58	28aed2a6	47	3d80477d	36	11f915bc	25	312bf560
57	abf71588	46	4716fe3e	35	6d88a37a	24	7f8d292f
56	09cf4f3c	45	1e237e44	34	110b3efd	23	ac7766f3
55	a0fafe17	44	6d7a883b	33	dbf98641	22	19fadc21
54	88542cb1	43	ef44a541	32	ca0093fd	21	28d12941
53	23a33939	42	a8525b7f	31	4e54f70e	20	575c006e
52	2a6c7605	41	b671253b	30	5f5fc9f3	19	d014f9a8
51	f2c295f2	40	db0bad00	29	84a64fb2	18	c9ee2589
50	7a96b943	39	d4d1c6f8	28	4ea6dc4f	17	e13f0cc8
49	5935807a	38	7c839d87	27	ead27321	16	b6630ca6

Table 9 lists the 32-bit subkeys generated in simulation during the expansion of a 192-bit key. The 52 subkeys are sequentially ordered from 59 to 8.

Table 9: AES Key Expansion (192-bit) – Fifty-two 32-bit Subkeys

Key: 8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b							
#	**Subkey**	#	**Subkey**	#	**Subkey**	#	**Subkey**
59	8e73b0f7	46	69b54118	33	27f93943	20	458c553e
58	da0e6452	45	85a74796	32	6a94f767	19	a7e1466c
57	c810f32B	44	e92538fd	31	c0a69407	18	9411f1df
56	809079e5	43	e75fad44	30	d19da4e1	17	821f750a
55	62f8ead2	42	bb095386	29	ec1786eb	16	ad07d753
54	522c6b7B	41	485af057	28	6fa64971	15	ca400538
53	fe0c91f7	40	21efb14f	27	485f7032	14	8fcc5006
52	2402f5a5	39	a448f6d9	26	22cb8755	13	282d166a
51	ec12068e	38	4d6dce24	25	e26d1352	12	bc3ce7b5
50	6c827f6b	37	aa326360	24	33f0b7b3	11	e98ba06f
49	0e7a95b9	36	113b30e6	23	40beeb28	10	448c773c
48	5c56fec2	35	a25e7ed5	22	2f18a259	9	8ecc7204
47	4db7b4bd	34	83b1cf9a	21	6747d26b	8	01002202

Table 10 lists the 32-bit subkeys generated in simulation during the expansion of a 256-bit key. The 60 subkeys are sequentially ordered from 59 to 0.

Table 10: AES Key Expansion (256-bit) – Sixty 32-bit Subkeys

Key: 603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4							
#	**Subkey**	**#**	**Subkey**	**#**	**Subkey**	**#**	**Subkey**
59	603deb10	44	b75d5b9a	29	268c3ba7	14	2e2f31d7
58	15ca71be	43	d59aecb8	28	09e04214	13	7e0af1fa
57	2b73aef0	42	5bf3c917	27	68007bac	12	27cf73c3
56	857d7781	41	fee94248	26	b2df3316	11	749c47ab
55	1f352c07	40	de8ebe96	25	96e939e4	10	18501dda
54	3b6108d7	39	b5a9328a	24	6c518d80	9	e2757e4f
53	2d9810a3	38	2678a647	23	c814e204	8	7401905a
52	0914dff4	37	98312229	22	76a9fb8a	7	cafaaae3
51	9ba35411	36	2f6c79b3	21	5025c02d	6	e4d59b34
50	8e6925af	35	812c81ad	20	59c58239	5	9adf6ace
49	a51a8b5f	34	dadf48ba	19	de136967	4	bd10190d
48	2067fcde	33	24360af2	18	6ccc5a71	3	fe4890d1
47	a8b09c1a	32	fab8b464	17	fa256395	2	e6188d0b
46	93d194cd	31	98c5bfc9	16	9674ee15	1	046df344
45	be49846e	30	bebd198e	15	5886ca5d	0	706c631e

All values are consistent with the results of the key expansion vectors in FIPS 197 [1].

5.2 AES Round Verification (ECB Verification)

The final round corresponds to the system's output. Round *0* corresponds to the initialised state for all AES rounds. Table 11 lists the round values calculated in simulation during encryption in ECB mode with a 128-bit key.

Table 11: AES ECB Encryption (128-bit)

Data: 00112233445566778899aabbccddeeff **Key:** 000102030405060708090a0b0c0d0e0f			
Round	**Subkey**	**Round**	**Subkey**
0	00102030405060708090a0b0c0d0e0f0	6	c62fe109f75eedc3cc79395d84f9cf5d
1	89d810e8855ace682d1843d8cb128fe4	7	d1876c0f79c4300ab45594add66ff41f
2	4915598f55e5d7a0daca94fa1f0a63f7	8	fde3bad205e5d0d73547964ef1fe37f1
3	fa636a2825b339c940668a3157244d17	9	bd6e7c3df2b5779e0b61216e8b10b689
4	247240236966b3fa6ed2753288425b6c	10/END	69c4e0d86a7b0430d8cdb78070b4c55a
5	c81677bc9b7ac93b25027992b0261996		

Table 12 lists the round values calculated in simulation during decryption in ECB mode with a 128-bit key.

Table 12: AES ECB Decryption (128-bit)

Data: 69c4e0d86a7b0430d8cdb78070b4c55a **Key:** 000102030405060708090a0b0c0d0e0f			
Round	**Subkey**	**Round**	**Subkey**
0	7ad5fda789ef4e272bca100b3d9ff59f	6	2d6d7ef03f33e334093602dd5bfb12c7
1	54d990a16ba09ab596bbf40ea111702f	7	3bd92268fc74fb735767cbe0c0590e2d
2	3e1c22c0b6fcbf768da85067f6170495	8	a7be1a6997ad739bd8c9ca451f618b61
3	b458124c68b68a014b99f82e5f15554c	9	6353e08c0960e104cd70b751bacad0e7
4	e8dab6901477d4653ff7f5e2e747dd4f	10/END	00112233445566778899aabbccddeeff
5	36339d50f9b539269f2c092dc4406d23		

Table 13 lists the round values calculated in simulation during encryption in ECB mode with a 192-bit key.

Table 13: AES ECB Encryption (192-bit)

Data: 00112233445566778899aabbccddeeff **Key:** 000102030405060708090a0b0c0d0e0f1011121314151617			
Round	**Subkey**	**Round**	**Subkey**
0	00102030405060708090a0b0c0d0e0f0	7	0c0370d00c01e622166b8accd6db3a2c
1	4f63760643e0aa85aff8c9d041fa0de4	8	7255dad30fb80310e00d6c6b40d0527c
2	cb02818c17d2af9c62aa64428bb25fd7	9	a906b254968af4e9b4bdb2d2f0c44336
3	f75c7778a327c8ed8cfebfc1a6c37f53	10	88ec930ef5e7e4b6cc32f4c906d29414
4	22ffc916a81474416496f19c64ae2532	11	afb73eeb1cd1b85162280f27fb20d585
5	80121e0776fd1d8a8d8c31bc965d1fee	12/END	dda97ca4864cdfe06eaf70a0ec0d7191
6	671ef1fd4e2a1e03dfdcb1ef3d789b30		

Table 14 lists the round values calculated in simulation during decryption in ECB mode with a 192-bit key.

Table 14: AES ECB Decryption (192-bit)

Data: dda97ca4864cdfe06eaf70a0ec0d7191 **Key:** 000102030405060708090a0b0c0d0e0f1011121314151617			
Round	**Subkey**	**Round**	**Subkey**
0	793e76979c3403e9aab7b2d10fa96ccc	7	93faa123c2903f4743e4dd83431692de
1	c494bffae62322ab4bb5dc4e6fce69dd	8	68cc08ed0abbd2bc642ef555244ae878
2	d37e3705907a1a208d1c371e8c6fbfb5	9	1fb5430ef0accf64aa370cde3d77792c
3	406c501076d70066e17057ca09fc7b7f	10	84e1dd691a41d76f792d389783fbac70
4	fe7c7e71fe7f807047b95193f67b8e4b	11	6353e08c0960e104cd70b751bacad0e7
5	85e5c8042f8614549ebca17b277272df	12/END	00112233445566778899aabbccddeeff
6	cd54c7283864c0c55d4c727e90c9a465		

Table 15 lists the round values calculated in simulation during encryption in ECB mode with a 256-bit key.

Table 15: AES ECB Encryption (256-bit)

Data: 00112233445566778899aabbccddeeff **Key:** 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f			
Round	**Subkey**	**Round**	**Subkey**
0	00112233445566778899aabbccddeeff	8	5aa858395fd28d7d05e1a38868f3b9c5
1	4f63760643e0aa85efa7213201a4e705	9	4a824851c57e7e47643de50c2af3e8c9
2	1859fbc28a1c00a078ed8aadc42f6109	10	c14907f6ca3b3aa070e9aa313b52b5ec
3	975c66c1cb9f3fa8a93a28df8ee10f63	11	5f9c6abfbac634aa50409fa766677653
4	1c05f271a417e04ff921c5c104701554	12	516604954353950314fb86e401922521
5	c357aae11b45b7b0a2c7bd28a8dc99fa	13	627bceb9999d5aaac945ecf423f56da5
6	7f074143cb4e243ec10c815d8375d54c	14/END	8ea2b7ca516745bfeafc49904b496089
7	d653a4696ca0bc0f5acaab5db96c5e7d		

Table 16 lists the round values calculated in simulation during decryption in ECB mode with a 256-bit key.

Table 16: AES ECB Encryption (256-bit)

Data: 8ea2b7ca516745bfeafc49904b496089 **Key:** 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f			
Round	**Subkey**	**Round**	**Subkey**
0	aa5ece06ee6e3c56dde68bac2621bebf	8	2e6e7a2dafc6eef83a86ace7c25ba934
1	d1ed44fd1a0f3f2afa4ff27b7c332a69	9	9cf0a62049fd59a399518984f26be178
2	cfb4dbedf4093808538502ac33de185c	10	88db34fb1f807678d3f833c2194a759e
3	78e2acce741ed5425100c5e0e23b80c7	11	ad9c7e017e55ef25bc150fe01ccb6395
4	d6f3d9dda6279bd1430d52a0e513f3fe	12	84e1fd6b1a5c946fdf4938977cfbac23
5	beb50aa6cff856126b0d6aff45c25dc4	13	6353e08c0960e104cd70b751bacad0e7
6	f6e062ff507458f9be50497656ed654c	14/END	00112233445566778899aabbccddeeff
7	d22f0c291ffe031a789d83b2ecc5364c		

All values are consistent with the results of the AES-128 vectors in FIPS 197 [1].

5.3 Verification of Other Modes of Operation

Table 17 lists the input, AES input, AES output and final output values calculated in simulation during encryption in CBC mode with a 128-bit key over a data stream of four blocks.

Table 17: CBC Mode Encryption (128-bit, 4 Blocks)

IV: 000102030405060708090a0b0c0d0e0f **Key:** 2b7e151628aed2a6abf7158809cf4f3c				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	6bc1bee22e409f96 e93d7e117393172a	6bc0bce12a459991 e134741a7f9e1925	7649abac8119b246 cee98e9b12e9197d	7649abac8119b246 cee98e9b12e9197d
2	ae2d8a571e03ac9c 9eb76fac45af8e51	d86421fb9f1a1eda 505ee1375746972c	5086cb9b507219ee 95db113a917678b2	5086cb9b507219ee 95db113a917678b2
3	30c81c46a35ce411 e5fbc1191a0a52ef	604ed7ddf32efdff 7020d0238b7c2a5d	73bed6b8e3c1743b 7116e69e22229516	73bed6b8e3c1743b 7116e69e22229516
4	f69f2445df4f9b17 ad2b417be66c3710	8521f2fd3c8eef2c dc3da7e5c44ea206	3ff1caa1681fac09 120eca307586e1a7	3ff1caa1681fac09 120eca307586e1a7

Table 18 lists the input, AES input, AES output and final output values calculated in simulation during decryption in CBC mode with a 128-bit key over a data stream of four blocks.

Table 18: CBC Mode Decryption (128-bit, 4 Blocks)

IV: 000102030405060708090a0b0c0d0e0f **Key:** 2b7e151628aed2a6abf7158809cf4f3c				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	7649abac8119b246 cee98e9b12e9197d	7649abac8119b246 cee98e9b12e9197d	6bc0bce12a459991 e134741a7f9e1925	6bc1bee22e409f96 e93d7e117393172a
2	5086cb9b507219ee 95db113a917678b2	5086cb9b507219ee 95db113a917678b2	d86421fb9f1a1eda 505ee1375746972c	ae2d8a571e03ac9c 9eb76fac45af8e51
3	73bed6b8e3c1743b 7116e69e22229516	73bed6b8e3c1743b 7116e69e22229516	604ed7ddf32efdff 7020d0238b7c2a5d	30c81c46a35ce411 e5fbc1191a0a52ef
4	3ff1caa1681fac09 120eca307586e1a7	3ff1caa1681fac09 120eca307586e1a7	8521f2fd3c8eef2c dc3da7e5c44ea206	f69f2445df4f9b17 ad2b417be66c3710

Table 19 lists the input, AES input, AES output and final output values calculated in simulation during encryption in CFB128 mode with a 192-bit key over a data stream of four blocks.

Table 19: CFB128 Mode Encryption (192-bit, 4 Blocks)

IV: 000102030405060708090a0b0c0d0e0f **Key:** 8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	6bc1bee22e409f96 e93d7e117393172a	0001020304050607 08090a0b0c0d0e0f	a609b38df3b1133d ddff2718ba09565e	cdc80d6fddf18cab 34c25909c99a4174
2	ae2d8a571e03ac9c 9eb76fac45af8e51	cdc80d6fddf18cab 34c25909c99a4174	c9e3f5289f149abd 08ad44dc52b2b32b	67ce7f7f81173621 961a2b70171d3d7a
3	30c81c46a35ce411 e5fbc1191a0a52ef	67ce7f7f81173621 961a2b70171d3d7a	1ed6965b76c76ca0 2d1dcef404f09626	2e1e8a1dd59b88b1 c8e60fed1efac4c9
4	f69f2445df4f9b17 ad2b417be66c3710	2e1e8a1dd59b88b1 c8e60fed1efac4c9	36c0bbd976ccd4b7 ef85cec1be273eef	c05f9f9ca9834fa0 42ae8fba584b09ff

Table 20 lists the input, AES input, AES output and final output values calculated in simulation during decryption in CFB128 mode with a 192-bit key over a data stream of four blocks.

Table 20: CFB128 Mode Decryption (192-bit, 4 Blocks)

IV: 000102030405060708090a0b0c0d0e0f **Key:** 8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	cdc80d6fddf18cab 34c25909c99a4174	0001020304050607 08090a0b0c0d0e0f	a609b38df3b1133d ddff2718ba09565e	6bc1bee22e409f96 e93d7e117393172a
2	67ce7f7f81173621 961a2b70171d3d7a	cdc80d6fddf18cab 34c25909c99a4174	c9e3f5289f149abd 08ad44dc52b2b32b	ae2d8a571e03ac9c 9eb76fac45af8e51
3	2e1e8a1dd59b88b1 c8e60fed1efac4c9	67ce7f7f81173621 961a2b70171d3d7a	1ed6965b76c76ca0 2d1dcef404f09626	30c81c46a35ce411 e5fbc1191a0a52ef
4	c05f9f9ca9834fa0 42ae8fba584b09ff	2e1e8a1dd59b88b1 c8e60fed1efac4c9	36c0bbd976ccd4b7 ef85cec1be273eef	f69f2445df4f9b17 ad2b417be66c3710

Table 21 lists the input, AES input, AES output and final output values calculated in simulation during encryption in OFB mode with a 256-bit key over a data stream of four blocks.

Table 21: OFB Mode Encryption (256-bit, 4 Blocks)

IV: 000102030405060708090a0b0c0d0e0f **Key:** 603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	6bc1bee22e409f96 e93d7e117393172a	0001020304050607 08090a0b0c0d0e0f	b7bf3a5df43989dd 97f0fa97ebce2f4a	dc7e84bfda79164b 7ecd8486985d3860
2	ae2d8a571e03ac9c 9eb76fac45af8e51	b7bf3a5df43989dd 97f0fa97ebce2f4a	e1c656305ed1a7a6 563805746fe03edc	4febdc6740d20b3a c88f6ad82a4fb08d
3	30c81c46a35ce411 e5fbc1191a0a52ef	e1c656305ed1a7a6 563805746fe03edc	41635be625b48afc 1666dd42a09d96e7	71ab47a086e86eed f39d1c5bba97c408
4	f69f2445df4f9b17 ad2b417be66c3710	41635be625b48afc 1666dd42a09d96e7	f7b93058b8bce0ff fea41bf0012cd394	0126141d67f37be8 538f5a8be740e484

Table 22 lists the input, AES input, AES output and final output values calculated in simulation during decryption in OFB mode with a 256-bit key over a data stream of four blocks.

Table 22: OFB Mode Decryption (256-bit, 4 Blocks)

IV: 000102030405060708090a0b0c0d0e0f **Key:** 603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	dc7e84bfda79164b 7ecd8486985d3860	0001020304050607 08090a0b0c0d0e0f	b7bf3a5df43989dd 97f0fa97ebce2f4a	6bc1bee22e409f96 e93d7e117393172a
2	4febdc6740d20b3a c88f6ad82a4fb08d	b7bf3a5df43989dd 97f0fa97ebce2f4a	e1c656305ed1a7a6 563805746fe03edc	ae2d8a571e03ac9c 9eb76fac45af8e51
3	71ab47a086e86eed f39d1c5bba97c408	e1c656305ed1a7a6 563805746fe03edc	41635be625b48afc 1666dd42a09d96e7	30c81c46a35ce411 e5fbc1191a0a52ef
4	0126141d67f37be8 538f5a8be740e484	41635be625b48afc 1666dd42a09d96e7	f7b93058b8bce0ff fea41bf0012cd394	f69f2445df4f9b17 ad2b417be66c3710

Table 23 lists the input, AES input, AES output and final output values calculated in simulation during encryption in CTR mode with a 128-bit key over a data stream of four blocks.

Table 23: CTR Mode Encryption (128-bit, 4 Blocks)

IV: f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff **Key:** 2b7e151628aed2a6abf7158809cf4f3c				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	6bc1bee22e409f96 e93d7e117393172a	f0f1f2f3f4f5f6f7 f8f9fafbfcfdfeff	ec8cdf7398607cb0 f2d21675ea9ea1e4	874d6191b620e326 1bef6864990db6ce
2	ae2d8a571e03ac9c 9eb76fac45af8e51	f0f1f2f3f4f5f6f7 f8f9fafbfcfdff00	362b7c3c67735163 18a077d7fc5073ae	9806f66b7970fdff 8617187bb9fffdff
3	30c81c46a35ce411 e5fbc1191a0a52ef	f0f1f2f3f4f5f6f7 f8f9fafbfcfdff01	6a2cc3787889374f beb4c81b17ba6c44	5ae4df3edbd5d35e 5b4f09020db03eab
4	f69f2445df4f9b17 ad2b417be66c3710	f0f1f2f3f4f5f6f7 f8f9fafbfcfdff02	e89c399ff0f198c6 d40a31db156cabfe	1e031dda2fbe03d1 792170a0f3009cee

Table 24 lists the input, AES input, AES output and final output values calculated in simulation during decryption in CTR mode with a 128-bit key over a data stream of four blocks.

Table 24: CTR Mode Decryption (128-bit, 4 Blocks)

IV: f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff **Key:** 2b7e151628aed2a6abf7158809cf4f3c				
Block	**Input**	**AES In**	**AES Out**	**Output**
1	874d6191b620e326 1bef6864990db6ce	f0f1f2f3f4f5f6f7 f8f9fafbfcfdfeff	ec8cdf7398607cb0 f2d21675ea9ea1e4	6bc1bee22e409f96 e93d7e117393172a
2	9806f66b7970fdff 8617187bb9fffdff	f0f1f2f3f4f5f6f7 f8f9fafbfcfdff00	362b7c3c67735163 18a077d7fc5073ae	ae2d8a571e03ac9c 9eb76fac45af8e51
3	5ae4df3edbd5d35e 5b4f09020db03eab	f0f1f2f3f4f5f6f7 f8f9fafbfcfdff01	6a2cc3787889374f beb4c81b17ba6c44	30c81c46a35ce411 e5fbc1191a0a52ef
4	1e031dda2fbe03d1 792170a0f3009cee	f0f1f2f3f4f5f6f7 f8f9fafbfcfdff02	e89c399ff0f198c6 d40a31db156cabfe	f69f2445df4f9b17 ad2b417be66c3710

All values are consistent with the results of the corresponding vectors for AES-128 for CBC and CTR, AES-192 for CFB and AES-256 for OFB in SP 800-38A [2].

5.4 Resource Utilisation

Synthesis of the VHDL design in Xilinx ISE resulted in the design consuming approximately 30% of the XC2VP50's available resources. The VHDL code is written to support targeting additional hardware platforms in the future. Xilinx-specific VHDL constructs are generally avoided to support future development. This has the side-effect of foregoing certain specialised Xilinx hardware characteristics such as BlockRAM. Other specialised features of the XC2VP50, such as embedded PowerPC 405 processors and RocketIO transceivers, are irrelevant to this design.

Table 25 summarises actual hardware resource consumption.

Table 25: Hardware Resource Summary

Category	Used	Total Available	% Used
Slice Flip-Flops	4045	47232	8%
4-input LUTs	10266	47232	21%
Slices	7452	23616	31%
In/Out (IOBs)	523	692	75%
GCLKs	1	16	6%

Flip-Flops and LUTs can be regarded as components available on slices. Multiple slice Flip-Flops and LUTs can be allocated on a given slice. The Slices value offers the most relative comparison of resource usage to other Xilinx designs. IOBs are the in/out pins required to interact with the device. This design favours parallel operation and loading multiple external data sources simultaneously. Since the key and the data block can be loaded at the same time a large number of IOBs are required. This number is greater versus a design that does not allow concurrent loading of multiple data sources, or that streams data over a serial bus.

The design is further estimated to be the equivalent of 121,526 logic gates.

5.5 Timing Behaviour and Throughput

Initial synthesis values estimate a minimum period of 7.497*ns* corresponding to a maximum clock frequency of 133.382*MHz*. However, the "place and route" timing constraints limit the minimum supported clock period to 17.762*ns*, corresponding to a maximum clock frequency of 56.3*MHz*. This frequency was used for comparative and simulation testing purposes since it reflects the prospective physical implementation. A 50*MHz* clock, corresponding to a period of 20*ns*, is used to produce easily read timing simulation graphs.

Simulations show there is approximately 100*ns* from the time the device is active until it can begin accepting inputs. Inputs and external enable signals applied before this time have no effect on the internal operation of the device.

The number of clock cycles and the resulting throughput varies considerably depending on the key size and whether or not the key must be expanded. The system takes the same amount of time to complete regardless of the operation type (encryption/decryption) or the operating mode (ECB/CBC/CFB/OFB/CTR). A description of observable signal manipulations for each key size follows for the cases where the key is new or the key was previously expanded.

5.5.1 Processing with a New 128-bit Key

A total of 62 clock cycles are needed after the input values are stored before the system can accept new inputs. The following list summarises the major events that occur on each clock edge:

- 1^{st} Edge: Inputs are buffered by the PREPROC component. If *Enable* is high then processing follows on the next clock cycle.
- 2^{nd} Edge: PREPROC stores the buffered values and pre-processes them based on *Mode* and *Enc/Dec*. Alternatively the IVREG component updates the *IV* bus. PREPROC stores no further values until POSTPROC completes.
- 3^{rd} Edge: KEYEXP initialises the expanded key index.
- 4^{th} Edge: The 0^{th} subkey is calculated. KEYSTR is enabled.
- 5^{th} Edge: The 1^{st} subkey is calculated. The 0^{th} subkey is stored by KEYSTR.
- 47^{th} Edge: The 43^{rd} subkey is calculated; the 42^{nd} subkey is stored by KEYSTR. KEYEXP will not calculate any other subkeys.

- 48^{th} Edge: The 43^{rd} subkey is stored by memory. KEYSTR will not store any other subkeys.
- 49^{th} Edge: ENCDEC initialises the round index.
- 50^{th} Edge: The 0^{th} round value is calculated; this corresponds to the initialisation of *State*.
- 60^{th} Edge: The 10^{th} round value is calculated. ENCDEC will not perform any further AES calculations.
- 61^{st} Edge: POSTPROC post-processes the last *State* calculated by ENCDEC. *DataOut* is stored with the result of post-processing and *OutputReady* is set high. A high *OutputReady* indicates the user may supply new inputs to the system. The PREPROC component receives an enable signal indicating it may store new inputs it buffers after the next clock edge.
- 62^{nd} Edge: POSTPROC sets *OutputReady* low. No further output signal updates occur until a new data block has been processed. The IVREG component updates the *IV* bus with the calculated IV produced during the 61^{st} clock edge. If the buffered *Enable* is high at the time of the 62^{nd} clock edge, this clock cycle corresponds behaviourally to the 1^{st} clock edge.

Figure 36 shows the timing diagram corresponding to CBC encryption with a new 128-bit key. This corresponds to the encryption of the first block in Table 17.

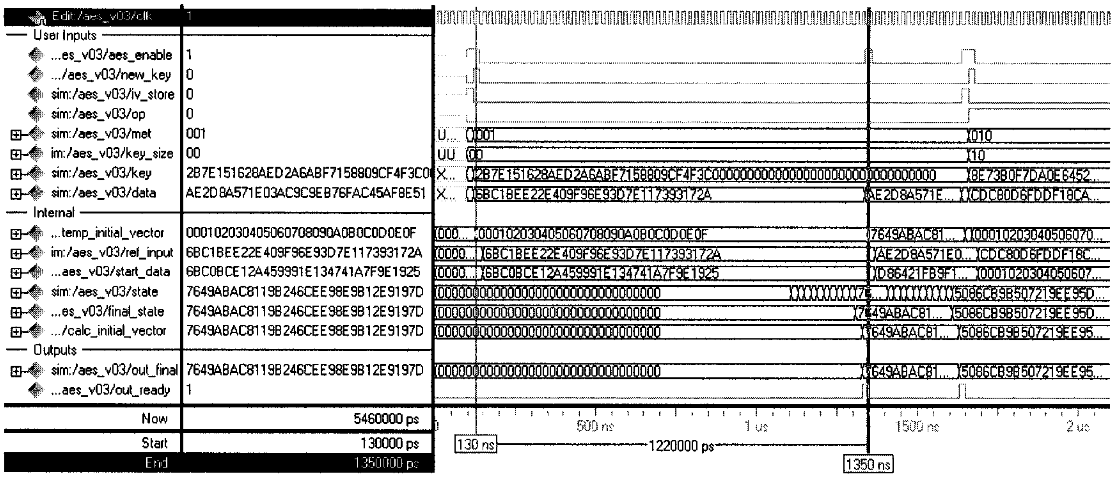

Figure 36: Timing Diagram for AES-CBC Encryption with a New 128-bit Key

The time elapsed between the 1st and 62nd clock cycles corresponds to 61 total clock cycles. Based on a 56.3*MHz* clock and 128-bit block size, the maximum throughput for 128-bit key rounds that require key expansion is 118.138*Mbps*.

5.5.2 Processing with a Previously Expanded 128-bit Key

A total of 16 clock cycles are needed after the input values are stored before the system can accept new inputs. The following list summarises the major events that occur on each clock edge:

- 1st Edge: Inputs are buffered by the PREPROC component. If *Enable* is high then processing follows on the next clock cycle.
- 2nd Edge: PREPROC stores the buffered values and pre-processes them based on *Mode* and *Enc/Dec*. Alternatively the IVREG component updates the *IV* bus. PREPROC stores no further values until POSTPROC completes.
- 3rd Edge: ENCDEC initialises the round index.

- 4th Edge: The 0th round value is calculated; this corresponds to the initialisation of *State*.
- 14th Edge: The 10th round value is calculated. ENCDEC will not perform any further AES calculations.
- 15th Edge: POSTPROC post-processes the last *State* calculated by ENCDEC. *DataOut* is stored with the result of post-processing and *OutputReady* is set high. A high *OutputReady* indicates the user may supply new inputs to the system. The PREPROC component receives an enable signal indicating it may store new inputs it buffers after the next clock edge.
- 16th Edge: POSTPROC sets *OutputReady* low. No further output signal updates occur until a new data block has been processed. The IVREG component updates the *IV* bus with the calculated IV produced during the 15th clock edge. If the buffered *Enable* is high at the time of the 16th clock edge, this clock cycle corresponds behaviourally to the 1st clock edge.

Figure 37 shows the timing diagram corresponding to CBC encryption with a previously expanded 128-bit key. This corresponds to the encryption of the second block in Table 17.

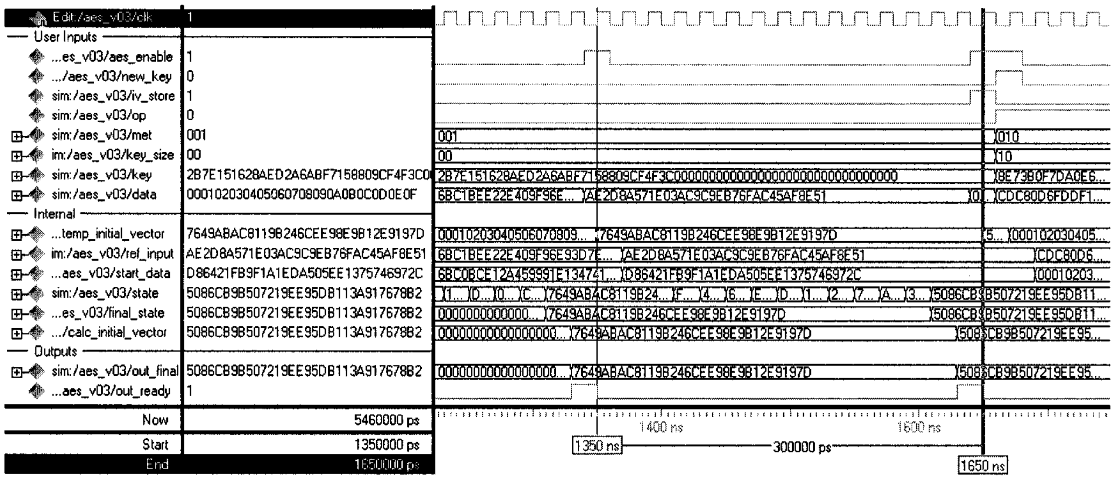

Figure 37: Timing Diagram for AES-CBC Encryption with an Expanded 128-bit Key

The time elapsed between the 1st and 16th clock cycles corresponds to 15 total clock cycles. Based on a 56.3*MHz* clock and 128-bit block size, the maximum throughput for 128-bit key rounds that don't require key expansion is 480.427*Mbps*.

5.5.3 Processing with a New 192-bit Key

A total of 72 clock cycles are needed after the input values are stored before the system can accept new inputs. The major events are identical to those in Chapter 5.5.1, only key expansion takes an additional eight clock cycles and round processing takes an additional two clock cycles. Figure 38 shows the timing diagram corresponding to CFB128 decryption with a new 192-bit key. This corresponds to the decryption of the first block in Table 20.

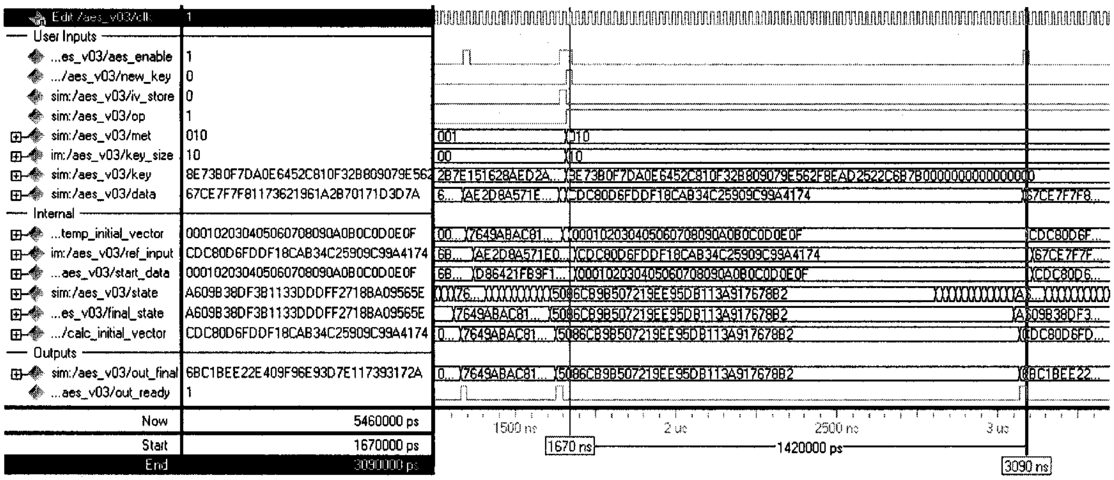

Figure 38: Timing Diagram for AES-CFB Decryption with a New 192-bit Key

The time elapsed between the 1st and 72nd clock cycles corresponds to 71 total clock cycles. Based on a 56.3*MHz* clock and 128-bit block size, the maximum throughput for 192-bit key rounds that require key expansion is 101.499*Mbps*.

5.5.4 Processing with a Previously Expanded 192-bit Key

A total of 18 clock cycles are needed after the input values are stored before the system can accept new inputs. The major events are identical to those in Chapter 5.5.2, only round processing takes an additional two clock cycles. Figure 39 shows the timing diagram corresponding to CFB128 decryption with a previously expanded 192-bit key. This corresponds to the decryption of the second block in Table 20.

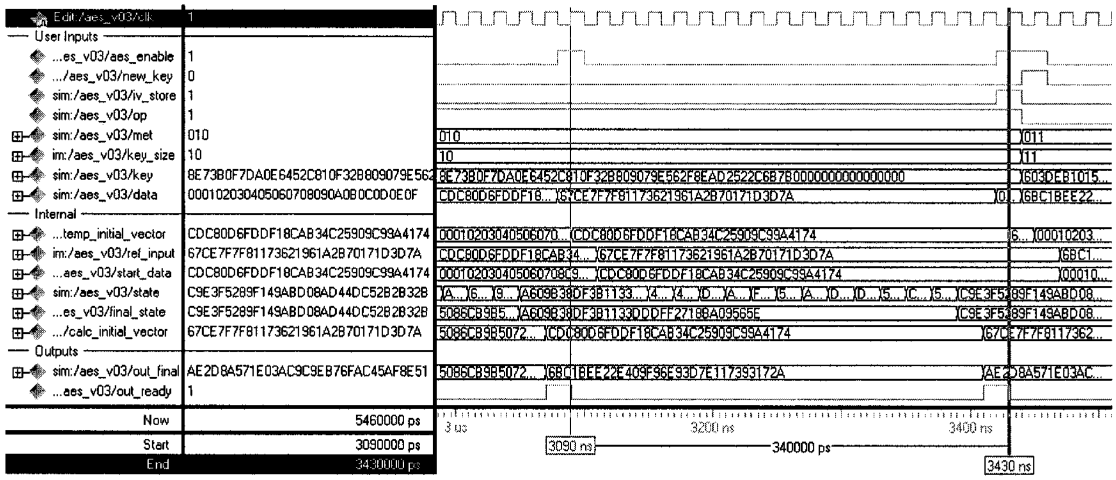

Figure 39: Timing Diagram for AES-CFB Decryption with an Expanded 192-bit Key

The time elapsed between the 1st and 18th clock cycles corresponds to 17 total clock cycles. Based on a 56.3*MHz* clock and 128-bit block size, the maximum throughput for 192-bit key rounds that preserve their key is 423.906*Mbps*.

5.5.5 Processing with a New 256-bit Key

A total of 82 clock cycles are needed after the input values are stored before the system can accept new inputs. The major events are identical to those in Chapter 5.5.1, only key expansion takes an additional sixteen clock cycles and round processing takes an additional four clock cycles. Figure 40 shows the timing diagram corresponding to OFB encryption with a new 256-bit key. This corresponds to the encryption of the first block in Table 21.

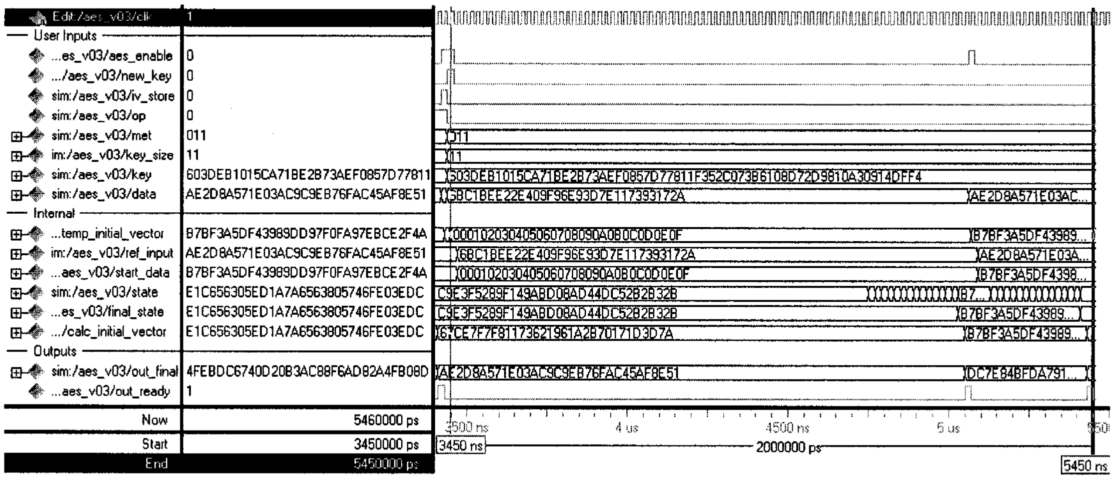

Figure 40: Timing Diagram for AES-OFB Encryption with a New 256-bit Key

The time elapsed between the 1st and 82nd clock cycles corresponds to 81 total clock cycles. Based on a 56.3*MHz* clock and 128-bit block size, the maximum throughput for 256-bit key rounds that require key expansion is 88.968*Mbps*.

5.5.6 Processing with a Previously Expanded 256-bit Key

A total of 20 clock cycles are needed after the input values are stored before the system can accept new inputs. The major events are identical to those in Chapter 5.5.2, only round processing takes an additional four clock cycles. Figure 41 shows the timing diagram corresponding to OFB encryption with a previously expanded 256-bit key. This corresponds to the encryption of the second block in Table 21.

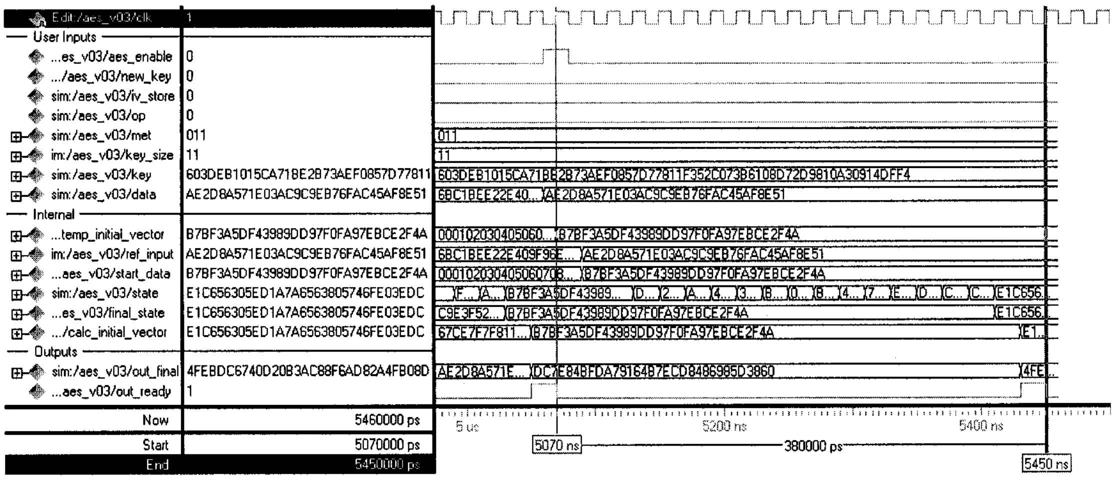

Figure 41: Timing Diagram for AES-OFB Encryption with an Expanded 256-bit Key

The time elapsed between the 1st and 20th clock cycles corresponds to 19 total clock cycles. Based on a 56.3*MHz* clock and 128-bit block size, the maximum throughput for 256-bit key rounds that preserve their key is 379.284*Mbps*.

5.6 Comparison to Similar Works

Table 26 compares other FPGA designs of various specialties to the design presented in this thesis.

Table 26: Comparison of AES Hardware Implementations

Ref.	Function	Hardware	Clock (*MHz*)	Slices	BRAMs	Through-put (*Mbps*)
[4]	AES-128 En/De	Xilinx XCV2000E	34.2	5677	80	4121
[5]	AES-128 En	Xilinx XCV800-6	71.8	9406	0	9184
		Xilinx XCV1000-6	125.3	11014	0	16032
[6]	AES-128/192/256 En/De	Xilinx XC2S50	510	10K Gates*7	0	370
[7]	AES-128 En/De	Xilinx XC3S50-4	71.5	163	3	208
		Xilinx XC2V40-6	123	146	3	358
[8]	AES-128 En	XC2V1000-5	159.21	1122	8	1940.9
[9]	AES-128 En/De	XC2VP100	196.3	2703 LUTs*7	44	1197
	AES-192 En/De		170.9	2710 LUTs*7	44	876
	AES-256 En/De		178.6	2745 LUTs*7	44	778
[10]	Rijndael 128/192/256 Data 128/192/256 Key En/De	XC2V8000	65	8378	4	832*1 693*2 594*3 *128-bit data only
This	AES-128/192/256 En/De ECB/CBC/ CFB/OFB/ CTR Modes	XC2VP50-7	56.3	7452	0	480.427*1 423.906*2 379.284*3 118.138*4 101.499*5 88.968*6

*1 = 128-bit key, *2 = 192-bit key, *3 = 256-bit key,

*4 = 128-bit key + Expansion, *5 = 192-bit key + Expansion, *6 = 256-bit key + Expansion

*7 = Alternate unit of resource consumption.

The bulk of proposed designs focus on AES-128 specialisation. In the case of Rodriguez-Henriquez *et al* [4] large throughput is possible with a design consuming 76%

of the slices in this design, however it also uses a very large number of BRAMs, a Xilinx-specific component with non-trivial cost. Similarly the designs presented by Zhang and Parhi [5] display exceptional throughput but are limited to AES-128 encryption only and require significant slice resources.

Hernandez *et al* [6] present an AES design supporting all key sizes and requiring a low-cost device and significantly few resources. The throughput is lower than this design's but at approximately one tenth of the size. However, it requires a clock nearly 10 times the speed of this design's to produce its rated throughput. The design by Rouvroy *et al* [7] is also aimed at the low-cost market; its slice consumption is only 2% that of this design. This comes at the expense of lower throughput and fewer key selections than Henandez *et al's* [6].

Sever *et al* [8] provide another limited AES-128 encryption design with approximately half the throughput of Rodriguez-Henriquez *et al* [4], however its size is also significantly reduced (albeit using 8 BRAMs). This design is a bridge between the high throughput designs of Rodriguez *et al* [4] and Zhang and Parhi [5] and the low area designs of Hernandez *et al* [6] and Rouvroy *et al* [7].

The designs presented by Brokalakis *et al* [9] and Lu and Lockwood [10] provide the most suitable comparison to this design in terms of the hardware they are built on and the capabilities they feature. Brokalakis *et al's* design [9] is a subset of a larger IPsec design. The hardware device they use is a higher capacity version of the one used in this design. Their consumption of LUTs is 25% of what this design uses with throughput roughly twice that of this design. Their design also requires a clock speed three to four times faster than this design's to produce their throughput. Furthermore it requires 44

BRAMs which as mentioned earlier are not a trivial resource and account for significant additional area occupied on the device. Additionally their AES cores are not integrated. A separate design is used for each of AES-128, AES-192 and AES-256 as opposed to this design's unified approach.

Lu and Lockwood's design [10] is a full-fledged implementation of the Rijndael algorithm of which AES is a subset. AES supports 128-bits data inputs versus Rijndael's support of 128-bit, 192-bit and 256-bit data inputs. That being said the performance of Lu and Lockwood [10] is very close to that of our design. It uses an incrementally faster clock to maintain incrementally faster throughputs for 128-bit data blocks. This comes at an expense of more than 900 additional slices and the use of 4 BRAMs. Removal of the Rijndael additions from this design would likely bring the design size on par with this design. In turn it may potentially support a smaller critical path and higher clock speed.

None of the designs above supply non-core AES features. The design presented in this thesis is the only design of these to fully incorporate the five modes of operation, feedback or otherwise, as opposed to relying on an external system. In addition this design incorporates key agility at the cost of throughput as per Chapters 5.5.1, 5.5.3 and 5.5.5. The user may change the key for any data block. If the user permanently enables the *NewKey* input, the expansion algorithm will be active for each block of data, effectively recalculating the cryptographic key for all data processed. The user controls the rate at which new keys are processed and expanded keys are preserved through the user selectable *NewKey* input.

5.7 Prototype Implementation

A prototype unit was constructed, allowing an entity to supply the full array of inputs to the FPGA. This ensures the final place-and-routed design is fully functional outside of simulations. The design was altered with the addition of thirty-two 4-to-7 decoding units that display the output signals in a human-readable format.

The prototype unit is divided into four main components. The core of the prototype is a Xilinx HW-AFX-FF1152-300 Prototype Board. This board provides complete access to the bulk of a FF1152 package's pins through breakout headers. The board is capable of generating the appropriate voltages for the FPGA's power and input needs through an external power supply. Sockets designed for clock crystals and differential clock inputs are also supplied and pre-connected to the appropriate pins through the PCB trace. Several methods exist for connecting the board to an external system for on the fly programming including Joint Test Action Group (JTAG). The board also has two PROMs whose capacity is capable of storing the program file for a Xilinx XC2VP20. As this design utilises a XC2VP50 it is necessary to program the device from a computer through the JTAG interface. In keeping with the maximum clock frequency of the design, a $50MHz$ clock crystal is connected to the board which in turn is connected to the appropriate pin of the FPGA. No $56.3MHz$ clock was available for testing. Figure 42 shows the HW-AFX-FF1152-300 assembly.

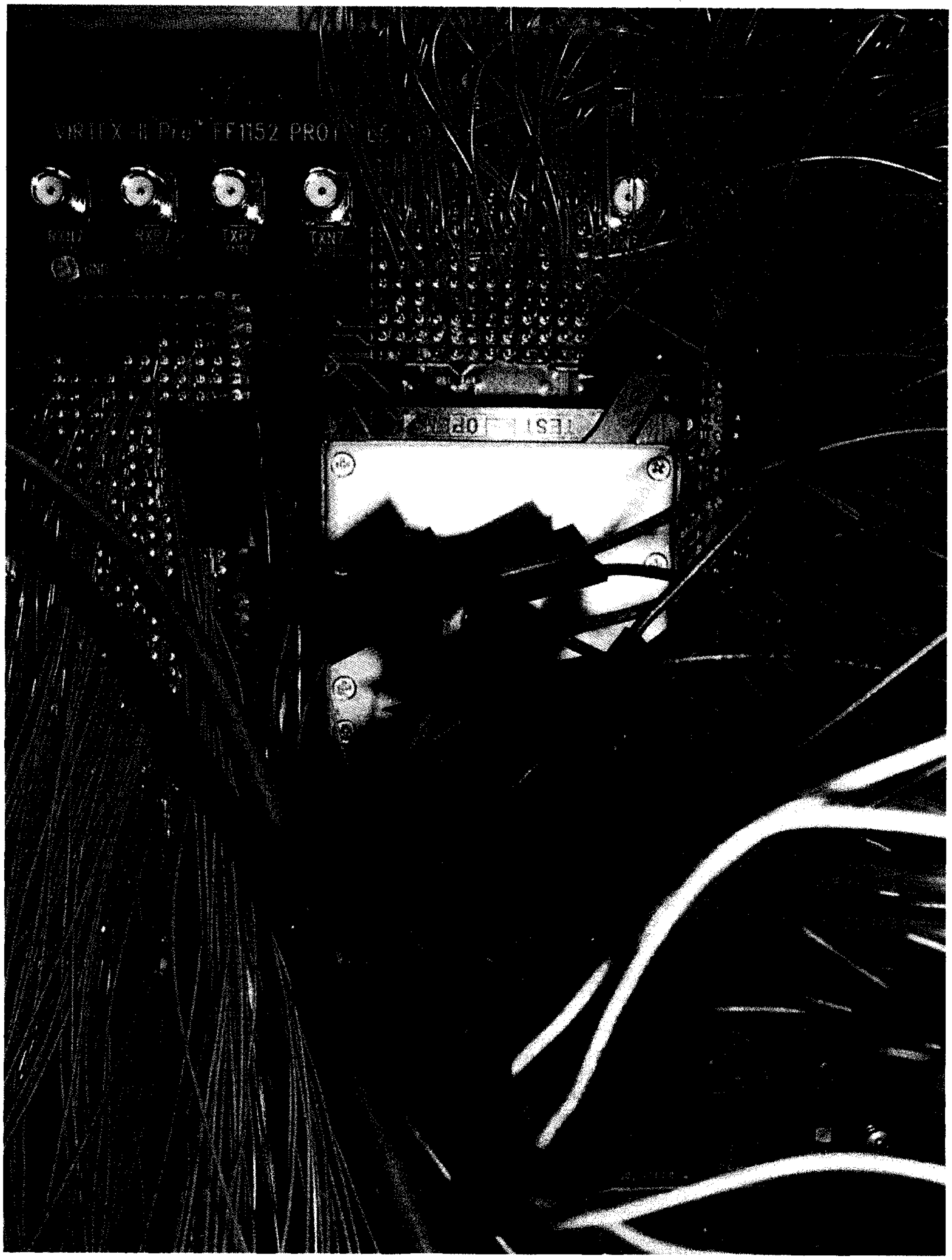

Figure 42: Xilinx HW-AFX-FF1152-300 Prototype Board & Virtex-II Pro XC2VP50

The second component is a 3M solderless breadboard featuring 96 Grayhill Series 94H 16-position rotary switches. These switches are arranged in six rows of 16 switches. Each switch corresponds to one hexadecimal character, outputting four signals corresponding to the binary value of that character. The first two rows of switches total 32 hexadecimal characters and are connected to the 128 inputs representing the data bus. The second two rows of switches total an additional 32 hexadecimal characters and are connected to the first 128 inputs representing the key bus. The fifth row of switches represents the additional values needed by a 192-bit key and are connected to the subsequent 64 inputs of the key bus. The sixth and final row of switches represents the additional values needed by a 256-bit key and are connected to the remaining 64 inputs of the key bus. Figure 43 shows the rotary switch assembly.

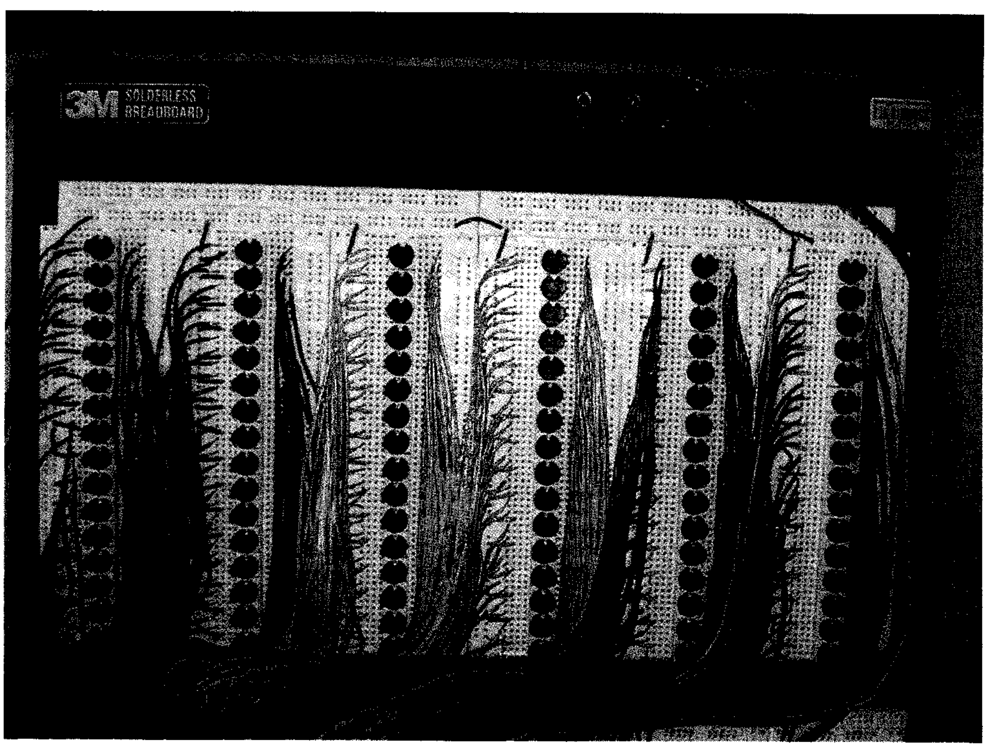

Figure 43: Rotary Switch Assembly (KeyIn and DataIn Bus)

The third component is a set of two smaller 3M solderless breadboards featuring 16 Lumex LDD-C512RI dual digit seven-segment displays. The 32 digits correspond to the 32 hexadecimal outputs (32x4-bit) of *DataOut*. It is necessary to decode each 4-bit value to 7-bits to be human-readable on the seven-segment displays. Additional code is added to the design to incorporate this feature for the purposes of this prototype unit only. It is also possible to run the unaltered design on the prototype unit. However, the average human cannot readily distinguish the 32 hexadecimal values of 128-bit separate LED outputs by sight. In using this additional multiplexer module the number of IOBs increases from 523 to 619. The eighth LED of a seven-segment display, representing a decimal point, is connected to the *OutputReady* terminal. Figure 44 shows two of the output assembly's four blocks, corresponding to 16 hexadecimal values.

Figure 44: Output Assembly (Block 1 and 3)

The final component is a Grayhill Series 78 10-position DIP switch and an Omron B3J tactile switch. The tactile switch is connected to the *Enable* input terminal. Of the remaining 10 switches, two switches are connected to the *KeySize* terminals, one switch is connected to the *Enc/Dec* terminal, three switches are connected to the *Mode* terminal, one switch is connected to the *NewKey* terminal and one switch is connected to the *LoadIV* terminal. The remaining two switches are unused. Figure 45 shows the DIP/tactile switch assembly.

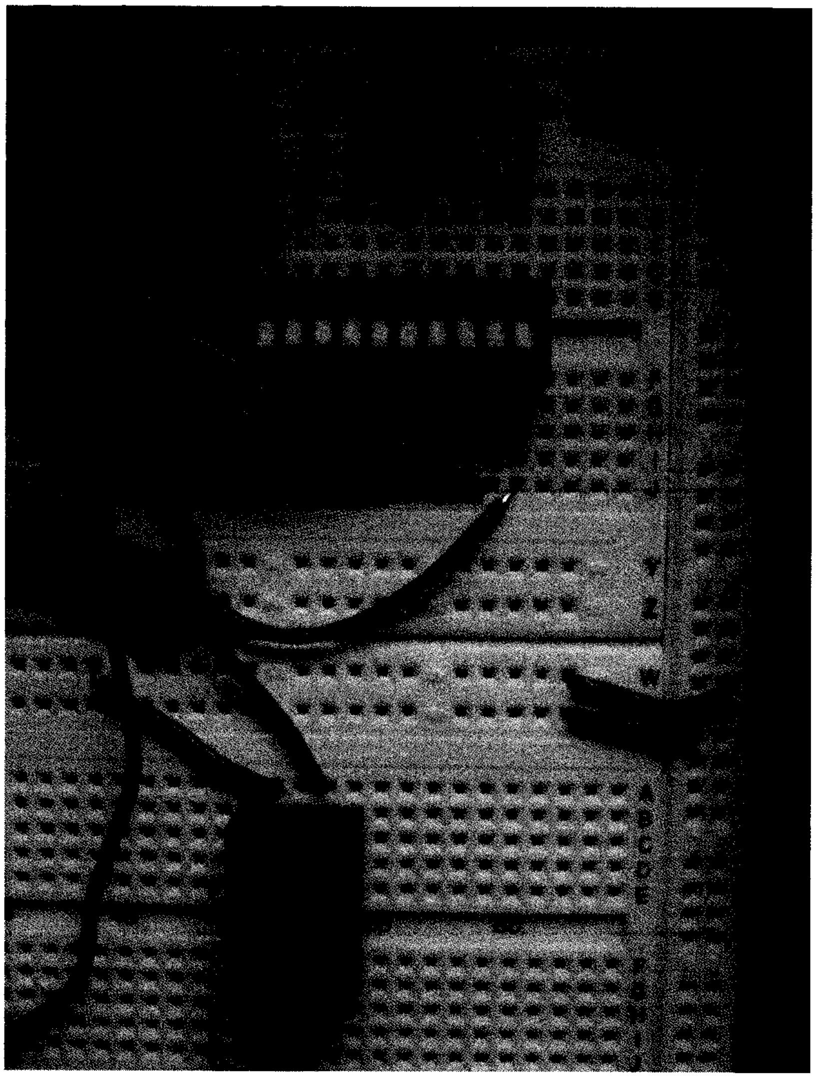

Figure 45: Output and Operation/Enable Assembly

The program file must be downloaded to the FPGA via the JTAG interface. This is accomplished using an appropriate JTAG USB interface and Xilinx iMPACT, a part of the ISE suite. Once downloaded, the user sets the hardware switches to correspond to any set of input values and presses the tactile switch to produce the corresponding output. This is functionally identical to the design interacting with an external hardware system providing inputs. The only difference is the human will be significantly slower at changing the inputs.

All outputs produced by this prototype have been consistent with simulation values and any values produced by the AES or mode algorithms. Figure 46 shows the complete prototype assembly.

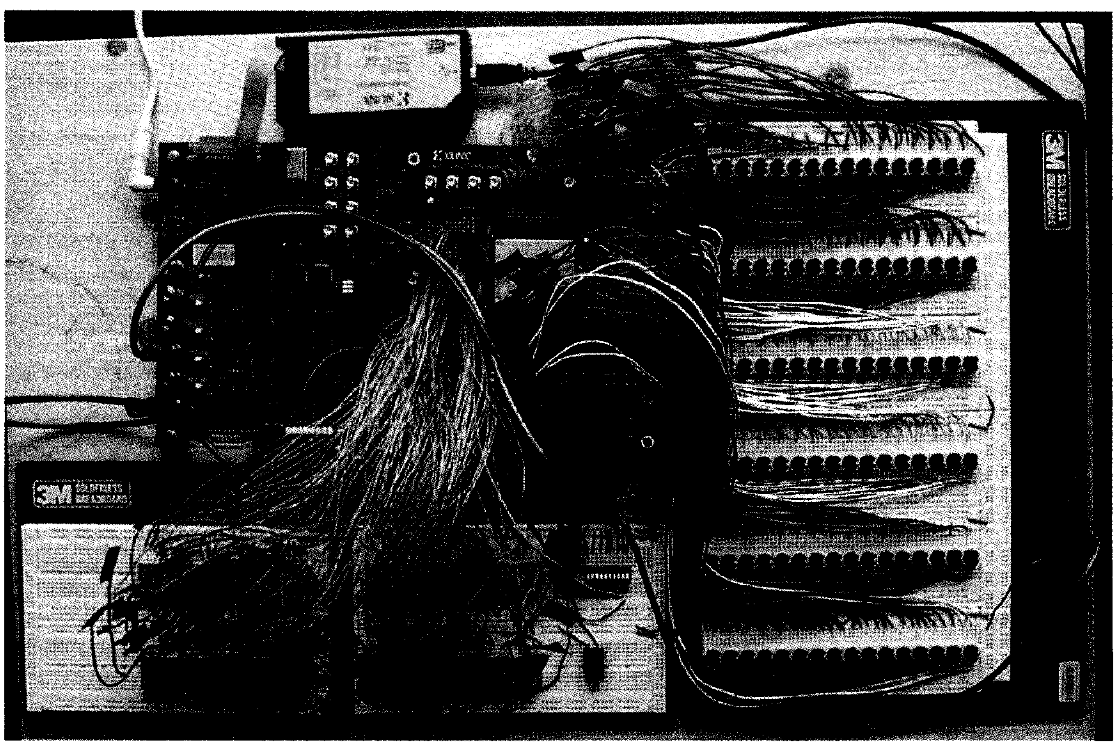

Figure 46: Complete Prototype Assembly

Chapter 6

Conclusion and Future Work

A VLSI implementation of AES supporting improved data confidentiality versus traditional FPGA designs was presented and functionally demonstrated on a simulation and practical level. The implementation was first designed in VHDL featuring the core aspects of AES including support for 128-bit, 192-bit and 256-bit key sizes with a fixed 128-bit block size and a key expansion module. The encryption or decryption key can be changed at the user's discretion for any block of data without interrupting the encryption or decryption operation. This provided a straightforward level of key agility. This design was enhanced with support for CBC, CFB, OFB and CTR confidentiality modes of operation in addition to the standard ECB mode. The targeted hardware platform was the Xilinx Virtex-II Pro FPGA XC2VP50. With a maximum clock speed of $56.3 MHz$, this design reached throughputs of $480.427 Mbps$, $423.906 Mbps$ and $379.284 Mbps$ for 128-bit, 192-bit and 256-bit keys respectively for all five modes of operation where the key had already been expanded. The throughput of the design was highly competitive while offering data confidentiality capabilities not found in contending offerings, overall providing a balanced variety of attractive features. A prototype hardware model was built and tested as a demonstration of the design's functional abilities.

Future revisions of this design would focus on three primary improvements. First, there is a sizable increase in the time the AES algorithm requires to encrypt/decrypt when key expansion is involved. Additional key agility research would greatly benefit the design since the design already integrates key expansion as part of an encryption/decryption operation. Second, despite the design's substantial throughput the clock speed is limited to $56.3 MHz$ due to its critical path of $17.792 ns$. Additional increases in throughput would be attainable by adjusting the algorithms in this design to decrease the critical path, thus increasing the permissible clock speed. Alternatively, adjusting the algorithms to perform additional operations within the same clock cycle would reduce the number of clock cycles and overall time required for an operation, presuming the critical path is not increased as a result. Finally, the design currently uses 7452 slices of a Xilinx XC2VP50 FPGA. This design's size may not be feasible for low-cost applications. This design's focus is not on small-area applications. Regardless, reducing the physical size of the design would assist it in targeting additional hardware platforms. It would be necessary to ensure this does not conflict with improvement of the performance and critical path. This module would be the first target for design review since the storage mechanism for the expanded key has been designed with a general hardware platform in mind.

References

[1] *Federal Information Processing Standards Publication 197, Specification for the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, November 2001.

[2] M. Dworkin, *NIST Special Publication 800-38A, Recommendation for Block Cipher Modes of Operation - Methods and Techniques*, National Institute of Standards and Technology, December 2001.

[3] X. Zhang, K.K. Parhi, "Implementation approaches for the Advanced Encryption Standard algorithm", *IEEE Circuits and Systems Magazine*, Vol 2, Iss 4, Fourth Quarter 2002, pp. 24-46.

[4] F. Rodriguez-Henriquez, N.A. Saqib, A. Diaz-Perez, "4.2 Gbit/s single-chip FPGA implementation of AES algorithm", *IEE Electronics Letters*, Vol 39, Iss 15, 24 July 2003, pp. 1115-1116.

[5] X. Zhang, K.K. Parhi, "High-speed VLSI architectures for the AES algorithm", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol 12, Iss 9, September 2004, pp. 957-967.

[6] O.J. Hernandez, T. Sodon, M. Adel, "Low-cost advanced encryption standard (AES) VLSI architecture: a minimalist bit-serial approach", *Proceedings of the IEEE SoutheastCon 2005*, 8-10 April 2005, pp. 121-125.

[7] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, J.-D. Legat, "Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications", *Proceedings of the International Conference on Information Technology: Coding and Computing 2004*, Vol 2, 2004, pp. 583-587.

[8] R. Sever, A.N. Ismailglu, Y.C. Tekmen, M. Askar, B. Okcan, "A high speed FPGA implementation of the Rijndael algorithm", *Euromicro Symposium on Digital System Design 2004*, 31 August - 3 September 2004, pp. 358-362.

[9] A. Brokalakis, A.P. Kakarountas, C.E. Goutis, "A high-throughput area efficient FPGA implementation of AES-128 Encryption", *IEEE Workshop on Signal Processing Systems Design and Implementation 2005*, 2-4 November 2005, pp. 116-121.

[10] J. Lu, J. Lockwood, "IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application", *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium 2005*, 4-8 April 2005, pp. 158b-158b.

[11] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996, p. 4.

[12] W. Diffie, M. Hellman, "Multiuser cryptographic techniques", *Proceedings of the AFIPS National Computer Conference 1976*, Vol 45, 7-10 June 1976, pp. 109-112.

[13] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM 1978*, Vol 21 Num 2, 1978, pp. 120-126.

[14] *Federal Information Processing Standards Publication 46-3, Data Encryption Standard (DES)*, National Institute of Standards and Technology, October 1999.

[15] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)", *Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993)*, Springer-Verlag, 1994, pp. 191-204.

[16] National Institute of Standards and Technology, "Announcing a Development of a Federal Information Processing Standard for Advanced Encryption Standard", *Federal Register: January 2, 1997*, http://csrc.nist.gov/CryptoToolkit/aes/pre-round1/aes_9701.txt (as of 20 June 2007).

[17] National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)", *Federal Register: September 12, 1997*, http://csrc.nist.gov/CryptoToolkit/aes/pre-round1/aes_9709.htm (as of 20 June 2007).

[18] E. Roback, M. Dworkin, "First Advanced Encryption Standard (AES) Candidate Conference Report", *Journal of Research of the National Institute of Standards and Technology*, Vol 103 Num 1, January-February 1999, pp. 97-105.

[19] IBM Corporation, *MARS – a candidate cipher for AES*, 22 September 1999, http://www.research.ibm.com/security/mars.pdf (as of 20 June 2007).

[20] R.L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, *The RC6^{TM} Block Cipher*, 20 August 1998, ftp://ftp.rsasecurity.com/pub/rsalabs/rc6/rc6v11.pdf (as of 20 June 2007).

[21] J.Daemen, V.Rijmen, *AES Proposal: Rijndael*, 3 September 1999, http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf (as of 20 June 2007).

[22] R.Anderson, E.Biham, L.Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*, 1999, http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf (as of 20 June 2007).

[23] B.Schneier, J.Kelsey, D.Whiting, D.Wagner, C.Hall, N.Ferguson, *Twofish: A 128-Bit Block Cipher*, 15 June 1998, http://www.schneier.com/paper-twofish-paper.pdf (as of 20 June 2007).

[24] A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", *The Third Advanced Encryption Standard Candidate Conference*, 13-14 April 2000, pp. 13-27.

[25] National Institute of Standards and Technology, *Commerce Department Announces Winner of Global Information Security Competition*, 2 October 2000, http://www.nist.gov/public_affairs/releases/g00-176.htm (as of 20 June 2007).

[26] R. Lidl, H. Niederreiter, "Finite Fields", *Encyclopedia of Mathematics and its Applications*, Vol 20, Addison-Wesley, 1983.