Design of a co-processor for AJIT processor to perform cryptographic operations

A Dissertation
submitted in partial fulfillment of the requirements
for the Degree of

Master of Technology

with
specialization in

Electronic Systems

by

Anshul Gupta
(143070097)

Supervisor

Prof. Madhav P. Desai

Department of Electrical Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai - 400 076.

2016

Dissertation Approval

The dissertation entitled

Design of a co-processor for AJIT processor to perform cryptographic operations

by

Anshul Gupta
(Roll No. : 143070097)

is approved for the degree of

Master of Technology in Electrical Engineering

Prof.	Prof.
Dept. of Electrical Engineering	Dept. of Electrical Engineering
(Examiner)	(Examiner)

Prof.	Prof.
Dept. of Electrical Engineering	Dept. of Electrical Engineering
(Chairperson)	(Supervisor)

Date: June 30, 2016
Place: IIT, Bombay.

Declaration

I hereby declare that the dissertation entitled **"Design of a co-processor for AJIT processor to perform cryptographic operations"** represents my ideas in my own words. In all occurrences where ideas or words or diagrams are taken from books/papers/electronic media, I have adequately cited and acknowledged the original sources. I declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will result in disciplinary action as per the norms of Institute and can also evoke legal action from the sources which have thus not been properly acknowledged or from whom proper permission has not been taken.

Anshul Gupta

(Roll no. : 143070097)

Date: June 30, 2016

Place: IIT, Bombay

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my guide Prof. Madhav P. Desai who with his profound experience and never ending enthusiasm provided constant support and guidance to my dissertation.

I would like to heartily thank Mr. Piyush Soni and Titto Thomas for their support. They solved all kinds of doubts and brought immense clarity to my concepts. They have been wonderful friends and guide to me.

I acknowledge the *VLSI Lab, Embedded Systems Lab, Department of Electrical Engineering, Indian Institute of Technology Bombay* for providing the various tools to work on.

Finally, I would like to thank my parents, friends and lab mates, who provided the needed base to work conveniently.

ABSTRACT

This report deals with the study of Advanced Encryption Standard (AES), its software implementation on AJIT processor and its hardware implementation on FPGA. AES-128 algorithm have been implemented with various approaches on AJIT processor. The time consumption of various implementations have been compared to get as efficient implementation as possible. The AES-128 implementation on AJIT have also been compared with the implementation on other 32-bit processors. After this, a bulk encryption engine has been implemented on Kintex FPGA evaluation board (KC-705). This engine performs AES-128 encryption and decryption. The hardware is described using Aa language and AHIR toolchain is used to convert it into VHDL script. The hardware is pipelined to get increased throughput. Riffa framework is used to communicate data between FPGA and CPU. The encryption engine has been tested using Advanced Encryption Standard Algorithm Validation Suite (AESAVS) described by National Institute of Standards and Technology (NIST).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Coporocessor is a computer processor which aids the primary processor in performing some specific computations faster. Primary processor, being a general purpose processor, is designed to perform basic operations. It consumes time while performing complex computations like cryptographic computations. Coprocessors are designed to perform specific operations/algorithms and takes much less time to compute as compared to the primary processor. It also shares the load of the primary processor. While, coprocessor is performing some operation , primary processor is idle and can perform some other task. Thus, a coprocessor increases the computational capacity of the system. which can perform the AES encryption and decryption.

The aim of our project is to design a bulk AES encryption-decryption engine which can used in the Cryptographic Coprocessor for AJIT processor. The encryption engine should at least 10 times faster than the AES software implementation on AJIT processor.

1.1 AJIT

AJIT is a processor which is designed on the basis of SPARC V-8 processor. SPARC is a CPU instruction set architecture (ISA), derived from a reduced instruction set computer (RISC) lineage. SPARC, formulated at Sun Microsystems in 1985, is based on the RISC I & II designs engineered at the University of Cali- fornia at Berkeley from 1980 through 1982. the SPARC reg- ister window architecture, pioneered in UC Berkeley designs, allows for straightforward, high-performance compilers and a significant reduction in memory load/store instructions over other RISCs, particularly for large application programs.[1]

1.2 Advanced Encryption Algorithm AES

The Advanced Encryption Standard specifies a Federal Information Processing Standards (FIPS) approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt and decrypt information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. [2]

1.3 Organization of the Report

The report is organized into 6 chapters. Chapter 2 gives the brief overview of the AES algorithm. Chatper 3 describes the software implementations of AES on AJIT machine and their performance. In chapter 4, the design of AES bulk encryption engine and its implementation. Chapter 5 discusses the design reports and the performance of the hardware engine along with its its testing and validation. Report concludes with Chapter 6 providing critical remarks on the proposed engine along with the scope of the future work.

Chapter 2

Introduction to Advanced Encryption Algorithm

The AES algorithm is adapted from the Rijndael cipher algorithm, developed by Joan Daemen and Vincent Rijmen. It processes 128 bit plaintext along with cipher key to produce a 128- bit cipher text. The cipher key is a sequence of 128,192 or 256 bits.

The AES algorithm can be broken into 5 major operations:

- Substitute-Byte Transform (Sub-Byte)
- Shift-Row Transform
- Mix-Columns Transform
- Add-Round Key Transform
- Key Expansion

The AES algorithm performs various rounds of iterative ciphering on the plaintext. Each round perform all the above mentioned transform (except round last which doesnot perform Mix-Columns Transform). Key Expansion is used to generate the key for each round from the previous round key.

The AES operations are performed on 2-D array of bytes known as state. Thus, the 16 bytes (or 128-bits) of $plaintext(0:127)$ arranged in 4×4 array as shown below:

S_0	S_4	S_8	S_{12}
S_1	S_5	S_9	S_{13}
S_2	S_6	S_{10}	S_{14}
S_3	S_7	S_{11}	S_{15}

Table 2.1: AES State Array

where, S_i represents $(i+1)^{th}$ byte of plaintext, i.e.

$$S_i = plaintext(8 * i : 8 * i + 7) \tag{2.1}$$

AES-128 uses a 128 bit cipher key and performs 10 rounds of iterative ciphering. The transformation performed in AES-128 encryption and decryption is shown in fig 2.1.

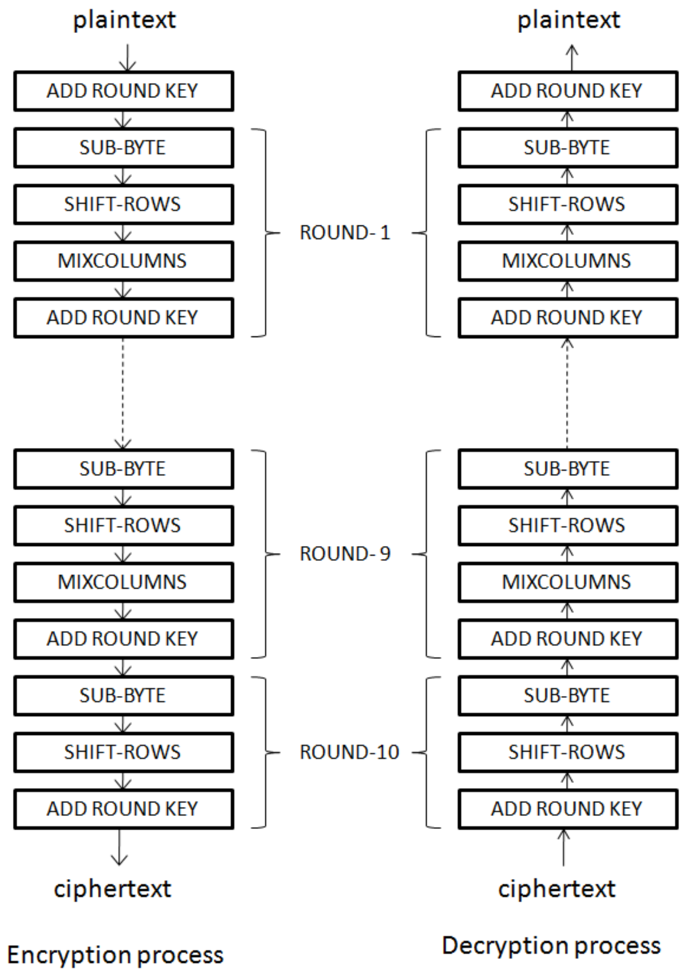


Figure 2.1: AES-128 Algorithm

2.1 Sub-Byte Transform

Sub-Byte transform is a non-linear transform implemented on each byte of AES state independently.The transform consists of two operations:

- multiplicative inverse operation in GF(2^8)
- affine transform

Since, the transform is performed individually on a byte and is independent of other by bytes of plain text, the output of the transform is fixed for a given 8-bit input. There are 256 possible input-output combinations which care be stored in a table called Rijndael's S-box.

2.2 Shift-Rows Transform

In Shift-Rows transform, the rows of the state array are circularly shifted left by different no. of times as shown in fig2.2:

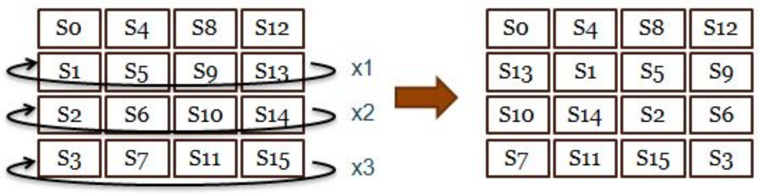


Figure 2.2: Shift-Rows Transformation

Here, S_i' represents the elements (or byte) of the transformed AES state.

Since, this transform performs a rotate left operation, the inverse of this transform (i.e. Inverse Shift-Rows Transform) can be performed by the same rotate left operation, but the no. of times a row is rotated is changed. The no. of left shifts performed on a given row during Shift-Rows and Inverse Shift-Rows Transform is tabulated in Table1.

Row. No.	No. of shifts for encryption	No.of shifts for decryption
1	0	0
2	1	3
3	2	2
4	3	1

Table 2.2: No. of shifts in Shift-Rows and Inverse Shift-Rows Transform

2.3 Mix-Columns Transform

Mix-Columns Transform is operated on the state array column-by column. The transform can be represented by matrix multiplication operation as shown in eq2.2. A column of state array is considered as a 1×4 matrix and is multiplied by a 4×4 transformation matrix.

$$\begin{bmatrix} S'_i \\ S'_{i+1} \\ S'_{i+2} \\ S'_{i+3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix} \tag{2.2}$$

where, i = 0,4,8 or 12.

In the matrix multiplication, each element belong to GF(2^8) defined by characteristic polynomial $x^8 + x^4 + x^3 + x + 1$. The addition and multiplication between two elements of matrices is performed in GF(2^8).

Like, Mix-Columns transform, Inverse Mix-Columns Transform can be represented using matrix operations in GF(2^8) as:

$$\begin{bmatrix} S'_i \\ S'_{i+1} \\ S'_{i+2} \\ S'_{i+3} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix} \tag{2.3}$$

2.4 Add Round Key Transform

In this transform, the key is XORed (i.e. addition in GF(2^8)) to the AES state.

$$\begin{bmatrix} S'_0 & S'_4 & S'_8 & S'_{12} \\ S'_1 & S'_5 & S'_9 & S'_{13} \\ S'_2 & S'_6 & S'_{10} & S'_{14} \\ S'_3 & S'_7 & S'_{11} & S'_{15} \end{bmatrix} = \begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \end{bmatrix} \uplus \begin{bmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{bmatrix} \tag{2.4}$$

This transform is performed once a round. For every round a separate key is used. In AES-128, 10 round keys are required. These round keys are generated by applying Key Expansion algorithm on the cipher key.

2.5 Key Expansion Algorithm

This algorithm is used to generate the round keys required for Add Round Key transform. Round key for a round is generated by applying the key expansion on the previous round key. The key expansion routine executes the following four operations:

- ROT WORD
- SUB WORD
- RCON
- XOR

ROT WORD means word rotation. In this operation, the 4 -byte word is cyclically rotated left by 1 byte.

SUB WORD is Sub-Byte transform of the four bytes of a word.

RCON stands for Round Constant. Rcon(i) is the round constant of i^{th} round in GF(2^8). Rcon(i) is given by

$$Rcon(i) = (02)^i$$

Round No. (i)	Rcon(i)
1	01
2	02
3	04
4	08
5	10
6	20
7	40
8	80
9	1B
10	36

Table 2.3: Rcon Table (in hex)

Round constant for computing the current round key can be calculated by multiplying the previous round constant by 2 in GF(2^8).

$$Rcon(i) = Rcon(i-1) * 2 \tag{2.5}$$

The Key Expansion algorithm is shown in fig:2.3:

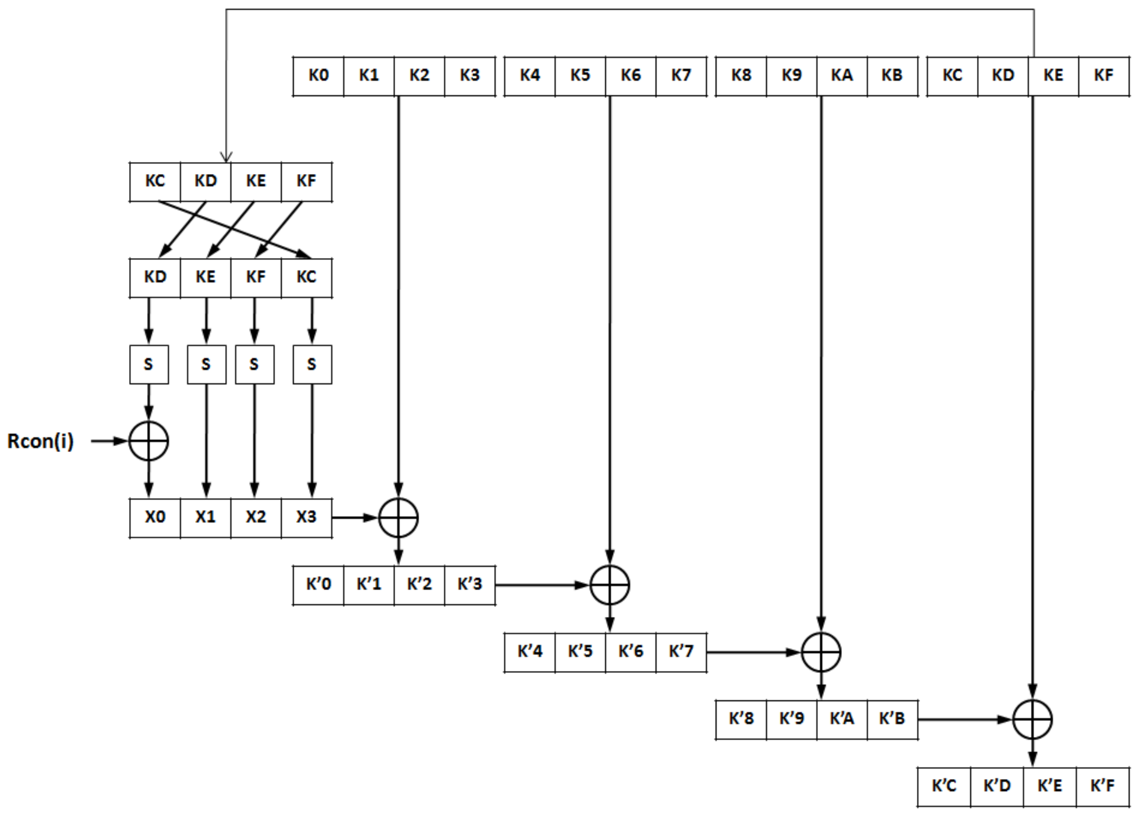


Figure 2.3: Key Expansion Algorithm

Here,

$\oplus$ represents XOR operation

S represents Sub-byte operation on a byte

Rcon(i) represents Round constant of i^{th} round

Chapter 3

Software implementation of AES

3.1 Implementation ver. 1

AJIT is a 32-bit processor. It requires 4 registers to store a state of AES. Let R0, R1, R2 and R3 be those registers. The contents of these registers is given by:

$$R0 = \{S_0, S_1, S_2, S_3\}$$

$$R1 = \{S_4, S_5, S_6, S_7\}$$

$$R2 = \{S_8, S_9, S_{10}, S_{11}\}$$

$$R3 = \{S_{12}, S_{13}, S_{14}, S_{15}\} \tag{3.1}$$

3.1.1 Implementation of Shift-Rows and Sub-Byte Transform

The optimal way to implement the Sub-byte transform is to use the look-up table approach. In this approach, the Rijndael S-box and inverse S-box (for deciphering) is stored in the memory location. Although, this approach takes up some memory, but it prevents the processor from perform the lengthy and time consuming computations. Using look-up table approach, Sub-Byte (or Inverse Sub-Byte) transform can be performed on a byte, by following 3 simple steps:

- 1. Taking out the byte (to be transformed, let's say X) from the input AES state registers.

- 2. Loading a byte (let's say Y) specified by X from the memory location in which the S-box table is stored.
- 3. Placing Y in the output AES state registers at it's place.

The implementation of the Shift-Rows transform can be merged with the Sub-Byte transform. This is done by changing the third step of Sub-Byte transform as follows:

- 3. Placing Y in the output AES state registers at a place obtained by X after Shift-Row Transform (shown in fig2.2).

By merging with Sub-Byte transform, the Shift-Rows transform can be performed without any extra effort by the processor.

Sub-Byte transform is applied on the bytes independently and Shift-Rows transform only changes the location of bytes in the AES state. Thus, the order in which the two transforms are performed, will not affect the AES state obtained after performing these two transformations. So, the Inverse Sub-Byte and Inverse Shift-Rows transforms can be implemented in a same way as Sub-Byte and Shift-Rows transforms.

3.1.2 Implementation of Mix-Column Transform

The Mixcolumn transformation matrix can be broken down as:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 02 & 02 & 00 & 00 \\ 00 & 02 & 02 & 00 \\ 00 & 00 & 02 & 02 \\ 02 & 00 & 00 & 02 \end{bmatrix} \oplus \begin{bmatrix} 01 & 01 & 01 & 01 \\ 01 & 01 & 01 & 01 \\ 01 & 01 & 01 & 01 \\ 01 & 01 & 01 & 01 \end{bmatrix}$$

$$\oplus \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix} \tag{3.2}$$

So, the Mixcolumn transform can be represented as:

$$\begin{bmatrix} S'_i \\ S'_{i+1} \\ S'_{i+2} \\ S'_{i+3} \end{bmatrix} = \begin{bmatrix} 02 & 02 & 00 & 00 \\ 00 & 02 & 02 & 00 \\ 00 & 00 & 02 & 02 \\ 02 & 00 & 00 & 02 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix} \oplus \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix}$$

$$\oplus \begin{bmatrix} S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \\ S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \\ S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \\ S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \end{bmatrix} \tag{3.3}$$

Thus, the Mix-Columns transform on a column can be performed as:

$$W = S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \tag{3.4}$$

$$X0 = S_i * 2 \tag{3.5}$$

$$X1 = S_{i+1} * 2 \tag{3.6}$$

$$X2 = S_{i+2} * 2 \tag{3.7}$$

$$X3 = S_{i+3} * 2 \tag{3.8}$$

$$S'_i = W \oplus X0 \oplus X1 \oplus S_i \tag{3.9}$$

$$S'_{i+1} = W \oplus X1 \oplus X2 \oplus S_{i+1} \tag{3.10}$$

$$S'_{i+2} = W \oplus X2 \oplus X3 \oplus S_{i+2} \tag{3.11}$$

$$S'_{i+3} = W \oplus X1 \oplus X2 \oplus S_{i+3} \tag{3.12}$$

Using the above method,a column of AES state can the Mix-Columns transformed by performing 15 XOR and four multipy by 2 (i.e. *2) operations in GF(2^8). Like Mix-Columns transformation matrix,Inverse Mix-Columns transformation matrix can be broken down as:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \end{bmatrix} \oplus \begin{bmatrix} 04 & 00 & 04 & 00 \\ 00 & 04 & 00 & 04 \\ 04 & 00 & 04 & 00 \\ 00 & 04 & 00 & 04 \end{bmatrix} \oplus \begin{bmatrix} 02 & 02 & 00 & 00 \\ 00 & 02 & 02 & 00 \\ 00 & 00 & 02 & 02 \\ 02 & 00 & 00 & 02 \end{bmatrix}$$

$$\oplus \begin{bmatrix} 01 & 01 & 01 & 01 \\ 01 & 01 & 01 & 01 \\ 01 & 01 & 01 & 01 \\ 01 & 01 & 01 & 01 \end{bmatrix} \oplus \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix} \tag{3.13}$$

So, the Inverse Mix-Columns transform can be represented as:

$$\begin{bmatrix} S'_i \\ S'_{i+1} \\ S'_{i+2} \\ S'_{i+3} \end{bmatrix} = \begin{bmatrix} 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix} \oplus \begin{bmatrix} 04 & 00 & 04 & 00 \\ 00 & 04 & 00 & 04 \\ 04 & 00 & 04 & 00 \\ 00 & 04 & 00 & 04 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix}$$

$$\oplus \begin{bmatrix} 02 & 02 & 00 & 00 \\ 00 & 02 & 02 & 00 \\ 00 & 00 & 02 & 02 \\ 02 & 00 & 00 & 02 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix} \oplus \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix} \bullet \begin{bmatrix} S_i \\ S_{i+1} \\ S_{i+2} \\ S_{i+3} \end{bmatrix}$$

$$\oplus \begin{bmatrix} S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \\ S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \\ S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \\ S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \end{bmatrix} \tag{3.14}$$

Thus, the Inverse Mix-Columns transform on a column can be performed as:

$$W = S_i \oplus S_{i+1} \oplus S_{i+2} \oplus S_{i+3} \tag{3.15}$$

$$X0 = S_i * 2 \tag{3.16}$$

$$X1 = S_{i+1} * 2 \tag{3.17}$$

$$X2 = S_{i+2} * 2 \tag{3.18}$$

$$X3 = S_{i+3} * 2 \tag{3.19}$$

$$Y0 = (X0 \oplus X2) * 2 \tag{3.20}$$

$$Y1 = (X1 \oplus X3) * 2 \tag{3.21}$$

$$Z = (Y0 \oplus Y1) * 2 \tag{3.22}$$

$$S'_i = Z \oplus Y0 \oplus W \oplus X0 \oplus X1 \oplus S_i \tag{3.23}$$

$$S'_{i+1} = Z \oplus Y1 \oplus W \oplus X1 \oplus X2 \oplus S_{i+1} \tag{3.24}$$

$$S'_{i+2} = Z \oplus Y0 \oplus W \oplus X2 \oplus X3 \oplus S_{i+2} \tag{3.25}$$

$$S'_{i+3} = Z \oplus Y1 \oplus W \oplus X1 \oplus X2 \oplus S_{i+3} \tag{3.26}$$

Using the above method, a column of AES state can be Inverse Mix-Columns transformed by performing 26 XOR and seven multipy by 2 (i.e. *2) operations in GF(2^8).

3.1.3 Implementation of Add-Round Key Transform

In this transformation the state array is XORed columnwise with the round key as shown in fig**??**

It is a simple operation and can be performed by loading the key from memory and using 32 bit-XOR operation (supported by the AJIT architecture) four times. The round keys needed for this transformation is generated by performing key expansion of cipher key.

3.1.4 Implementation of Key Expansion algorithm

Like Sub-Byte transform and Shift-Rows transform (section 3.1), the ROT WORD and SUB WORD operations are merged to save the computation time. RCON values are saved in a table. The implementation KEy Expansion is same as shown in fig:2.3

3.2 Implemntation ver. 2

The implementation of the AES algorithm can be further improved by changing the way in which AES state is stored in the registers. Instead of storing the bytes of a AES state column in a register, we can store a row of AES state in a register. Register contents will now be:

$$R0 = S_0, S_4, S_8, S_{12} \tag{3.27}$$

$$R1 = S_1, S_5, S_9, S_{13} \tag{3.28}$$

$$R2 = S_2, S_6, S_{10}, S_{14} \tag{3.29}$$

$$R3 = S_3, S_7, S_{11}, S_{15} \tag{3.30}$$

Working with transposed AES state will change the way in which some of the transformations and key expansion are performed.

3.2.1 Sub-Byte and Shift-Rows Transforms on transposed AES state

Sub-byte transform is performed on each byte independently. So, the tranposing of AES state will not have any effect on the implementation of Sub-byte transform. Shift-Rows transform can be merged with Sub-Byte transform as described in the previous section.

3.2.2 Mix-Columns Transfom on transposed AES state

The Mix-Columns transformation , which was performed column-by-column earlier, can be performed on all th columns simultaneously using transposed AES state representation. For this, the operations used in performing the Mix-column transform are modified so that they can work with 32-bit data. XOR operation will remain the same. Multiply by 2 ($*2$) is be modified to $\otimes 2$ operation. $*2$ operation performs multiplication by 2 in GF(2^8), whereas $\otimes 2$ takes a 32-bits input, treats it as four 8-bit numbers in GF(2^8) and performs four $*2$ simultaneously.

Using the modified operations, Mix-Columns transform on the transposed AES state can be performed as:

$$W = R0 \oplus R1 \oplus R2 \oplus R3 \tag{3.31}$$

$$X0 = R0 \otimes 2 \tag{3.32}$$

$$X1 = R1 \otimes 2 \tag{3.33}$$

$$X2 = R2 \otimes 2 \tag{3.34}$$

$$X3 = R3 \otimes 2 \tag{3.35}$$

$$R'0 = W \oplus X0 \oplus X1 \oplus R0 \tag{3.36}$$

$$R'1 = W \oplus X1 \oplus X2 \oplus R1 \tag{3.37}$$

$$R'2 = W \oplus X2 \oplus X3 \oplus R2 \tag{3.38}$$

$$R'3 = W \oplus X1 \oplus X2 \oplus R3 \tag{3.39}$$

Here, $R'0, R'1, R'2, R'3$ are the registers contaning the AES state after transformation.

Similarly, Inverse Mix-Columns transform on transposed AES state can be implemented as:

$$W = R0 \oplus R1 \oplus R2 \oplus R3 \tag{3.40}$$

$$X0 = R0 \otimes 2 \tag{3.41}$$

$$X1 = R1 \otimes 2 \tag{3.42}$$

$$X2 = R2 \otimes 2 \tag{3.43}$$

$$X3 = R3 \otimes 2 \tag{3.44}$$

$$Y0 = (X0 \oplus X2) \otimes 2 \tag{3.45}$$

$$Y1 = (X1 \oplus X3) \otimes 2 \tag{3.46}$$

$$Z = (Y0 \oplus Y1) \otimes 2 \tag{3.47}$$

$$R'0 = Z \oplus Y0 \oplus W \oplus X0 \oplus X1 \oplus R0 \tag{3.48}$$

$$R'1 = Z \oplus Y1 \oplus W \oplus X1 \oplus X2 \oplus R1 \tag{3.49}$$

$$R'2 = Z \oplus Y0 \oplus W \oplus X2 \oplus X3 \oplus R2 \tag{3.50}$$

$$R'3 = Z \oplus Y1 \oplus W \oplus X1 \oplus X2 \oplus R3 \tag{3.51}$$

Since, transformation on all the 4 columns is performed simultaneously, the approach of using the transposed state representation saves a lot of cycles in performing AES encryption or decryption.

3.2.3 Add Round Key transform and Key Expansion on transposed AES state

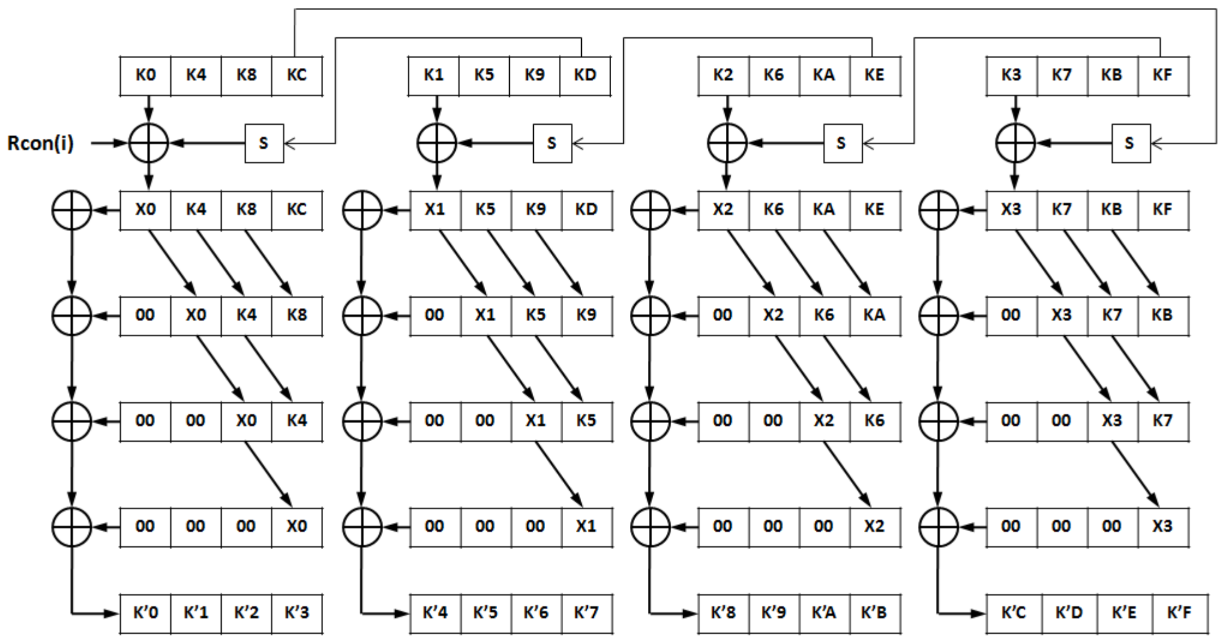


Figure 3.1: Transposed Key Expansion Algorithm

Here,

$\oplus$ represents XOR operation

S represents Sub-byte operation on a byte

Rcon(i) represents Round constant of i^{th} round

For efficient implementation of Add Round Key transform on the transposed AES state, either the round keys should be stored in transposed fashion or round key should be transposed before XORing with the AES state. Transposing every round key requires a lot of computation. Instead, the Key Expansion algorithm is modified such that the keys are generated in transposed fashion. This modified key expansion algorithm is shown in fig 3.1

The Transposed Key Expansion algorithm takes a transposed key as input and gives the transposed key for next round as output. It is required to transpose the cipher key before performing Add Round Key transformation or Key Expansion. In order to generate a round key, the algorithm performs 4 Sub-Byte operations, 21 XOR (2-input) operations, 13 shifts and 1 multiply by 2 in GF(2^8) (for computing round constant).

3.2.4 Results and Comparision

Assuming that each instruction takes 1 cycle for completion, the no. of cycles taken by various operations and transformation of AES state is tabulated in table: 3.1

S.No.	Operations/Transformation	No. of cycles taken by Impl.Ver 1	No. of cycles taken by Impl.Ver 2
1	Sub-Byte + Shift-Rows or InvSub-Byte + InvShift-Rows	40	65
2	Mix-Columns	52	54
3	Inverse Mix-Columns	208	92
4	Add Round Key	8	8
5	Transpose of AES state (or key)	-	46
6	AES-128 encryption (excluding Key expansion)	1,993	1,377
7	AES-128 decryption (excluding Key expansion)	2,097	1,721
8	Key Expansion (all 10 rounds)	305	578

Table 3.1: Performance comparision of AES software implementations on AJIT

It can be seen from table:3.1 that the transposed state implementation (Implementation Ver2) of AES-128 is better than the implementation discussed in section 3 (Im-

Processor	Key Expansion	Encryption	Decryption
AJIT Implement. Ver1	305	1,993	2,497
AJIT Implement. Ver2	578	1,377	1,721
ARM7TDMI [4]	634	1,675	2074
ARM9TDMI [4]	499	1384	1764
Pentium III [4]	370	1119	1395

Table 3.2: Performance of AES-128 implementation on various 32-bit platforms

plementation Ver1). The no. of cycles taken by Implementation Ver2 in Key expansion is almost double than that taken by Implementation Ver1. The time consumption in Mix-Column and Inverse Mix-Column Transform by Implementation Ver2 is immensly less than that by Implementation Ver1. This not even compensates the extra time taken in computations of other transforms and key expansion, but also reduces the total time taken in encryption and decryption.

On Compairing our AJIT AES-128 Implementation Ver2 with the AES-128 implementations performed by G.Bertoni, L.Breveglieri, P.Fragneto, M.Macchetti, and S.Marchesin in [4], it can be seen that the performance of our SPARC implementation Ver2 is similar to the AES-128 implementations on other platforms.

Chapter 4

AES Encryption Engine

An AES Encryption Engine is designed which can perform bulk encryption and decryption. The engine is to be used in the co-processor for AJIT. The engine can perform AES-128 encryption and decryption in Electronic Code Book (ECB) Mode. The engine is capable of performing encryption and decryption parallely.

The engine uses 6 pipes (64-bit wide) for data communication and interfacing as shown in the fig: 4.1

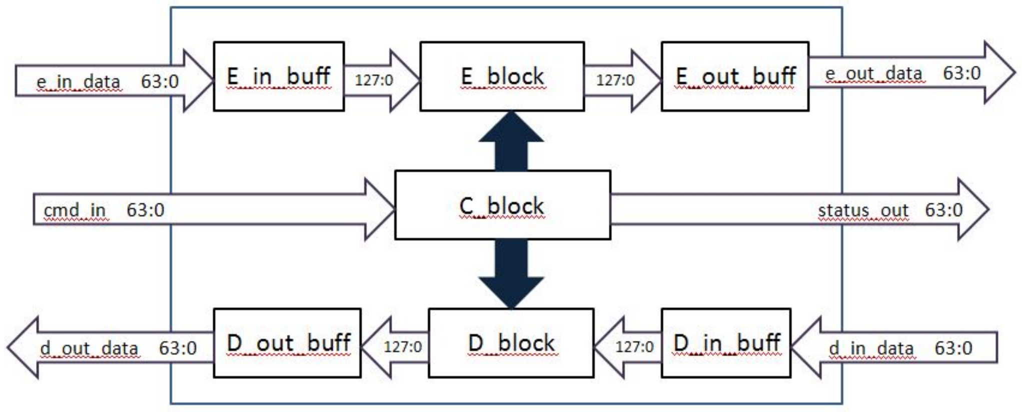


Figure 4.1: AES encryption Engine

E_in_buff, D_in_buff, E_out_buff and D_out_buff are the input and output data buffers. They also convert the 64-bit input data into 128-bit and viceversa.

4.1 Encryption and Decryption Modules

E_block and D_block are encryption and decryption modules respectively. They take the blocks (128 bits in a block) of data from the input buffer encrypt/decrypt it and give the blocks of encrypted/decrypted data to the output buffer. They also performs the Key Expansion and generates the round keys required for various rounds of encryption. The number of blocks to encrypt or decrypt and the cipher key is given by c_block. The flow of E_block and D_block is shown in fig: 4.2

They contains a key_expand_single, enc_round and dec_round modules to perform encryption/decryption.key_expand_single takes the round key and round constant (Rcon) of a round and gives the next round key and next Rcon as output. enc_round module takes AES state and round key as input and performs the single round of encryption. dec_round module takes AES state and round key as input and performs the single round of decryption. E_block contains 10 enc_round modules which are pipelined to increase the throughput of the engine. Similarly, D_block contains 10 dec_round modules arranged in a pipeline manner.

4.1.1 Implementation AES round and round Key Expansion

A single round of AES encryption and decryption is implemented in enc_round module and dec_round module respectively. The Rijndael Sbox and Inverse Sbox are are implemented as 256x1 mux. 16 Sboxs and Inverse Sboxs are used to implement Sub-Byte and Inv Sub-Byte transform. For implementing Mix-Column and inverse Mix-Column transform, the approach of breaking the matrix multiplication in GF(2^8) into various XOR and multiply by 2 in GF(2^8) operations (as shown in section 3.1.2) is used. Multiply by 2 in GF(2^8) operation consists of a left shift, a comparison and and XOR operation. It is implemented as 32-bit XOR and a 2x1 mux.

A single round key expansion using previous round key is performed key_expand_single module. key_expand_single module is implemented by using 4 Sboxs and 5 32-bit XORs as shown in fig:2.3

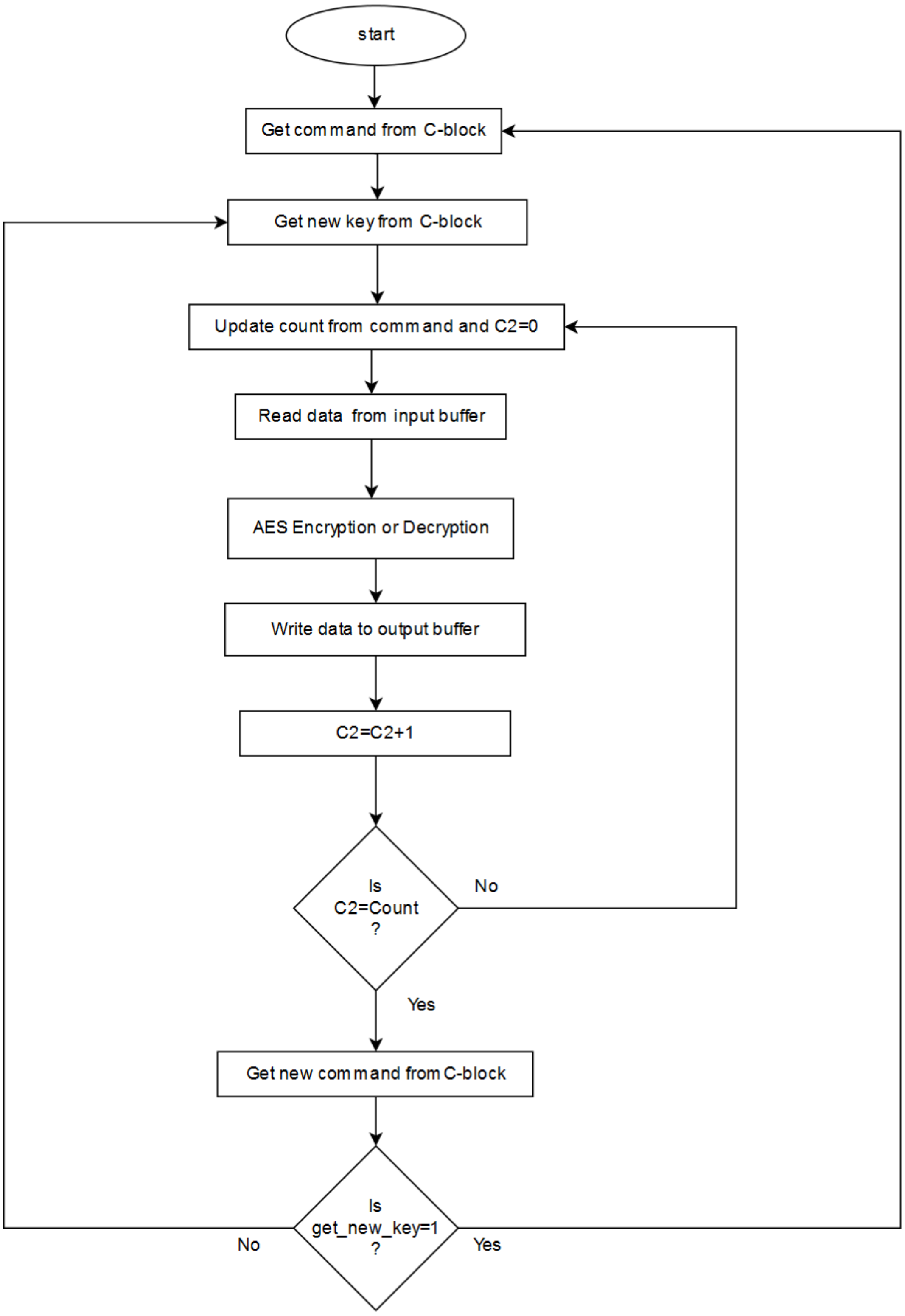
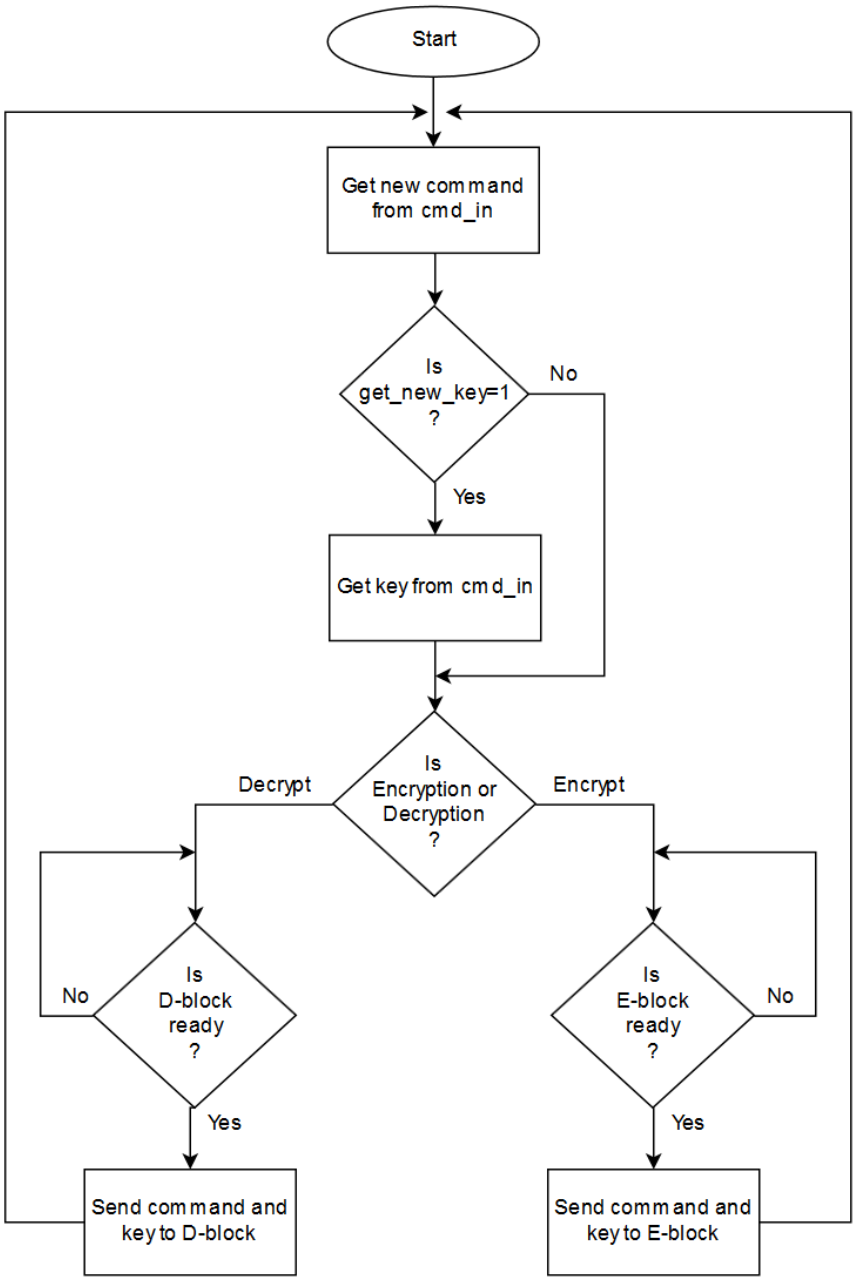

start
Get command from C-block
Get new key from C-block
Update count from command and C2=0
Read data from input buffer
AES Encryption or Decryption
Write data to output buffer
C2=C2+1
Is C2=Count ?
No
Yes
Get new command from C-block
Is get_new_key=1 ?
No
Yes

Figure 4.2: E_block and D_block flowchart

4.2 Control module

C_block is designed to control the E_block and the D_block. It receives the command and the cipher key (if needed) from the cmd_in pipe and correspondingly gives assigns the work to E_block or D_block. It also checks whether E_block/D_block is busy or not before assigning them new task. The flow of C_block is shown in fig: 4.3



Start
Get new command from cmd_in
Is get_new_key=1 ?
No
Yes
Get key from cmd_in
Is Encryption or Decryption ?
Decrypt
Encrypt
Is D-block ready ?
No
Yes
Is E-block ready ?
No
Yes
Send command and key to D-block
Send command and key to E-block

Figure 4.3: C_block flowchart

The format of the command given to encryption engine is shown below:

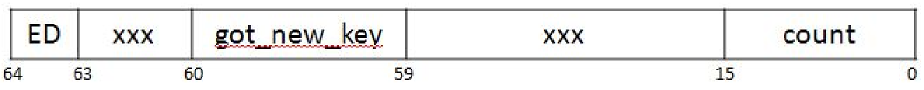


Figure 4.4: Input Command Format for AES Engine

- ED : this bit specifies whether to perform encryption or decryption.
- got_new_key: this bit represents whether a new cipher key is used for the encryption/decryption. If this bit is '1', then the next 2 inputs from the cmd_in are used for providing the new key.
- count: this specifies the number of blocks (128 bits) of input data to be encrypted or decrypted.
- xxx : bits reserved for future use.

4.3 Hardware Implementation of AES Engine

The AES bulk encryption is described using Aa language. AHIR toolchain is used to convert the Aa file to VHDL description. GHDL is used to simulate the design. The design is synthesized and burned on Kintex VC705 evaluation board for testing. Vivado Design suite (Xilinx) 14.3 is used for design synthesis and bit-file generation for the FPGA.

Riffa (Reusable Integration Framework for FPGA Accelerators) framework is used for the communication between FPGA and CPU. Due to the channel limitation of Riffa, a wrapper is used to communicate to the AES Engine.

4.3.1 AES Engine wrapper

The AES bulk encryption Engine designed, requires 3 channel communication (3 input pipes and 3 output pipes). A wrapper is used to wrap around the engine and convert the 3 channel communication to single channel (i.e. one input pipe and one output pipe).

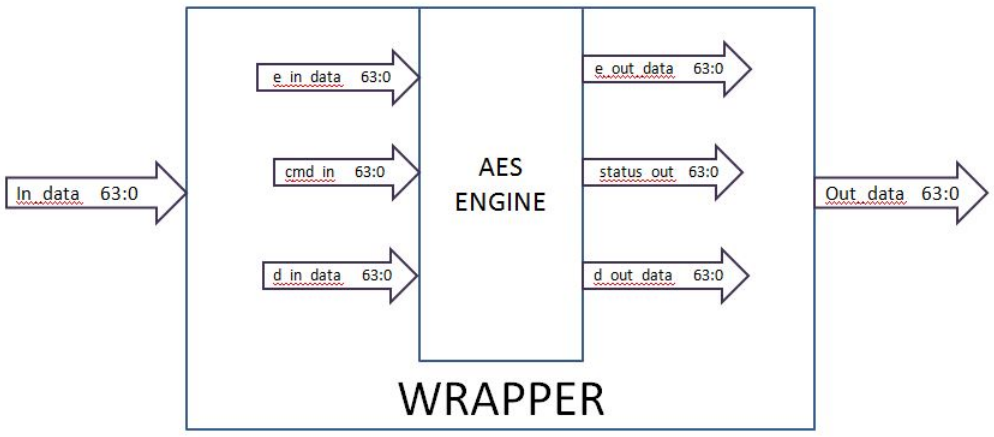


Figure 4.5: AES Wrapper

The wrapper communicates the data in the form of packets. The packet size is 512 blocks (8192 bytes). The organization of packet is:

1. The packet starts with Header (128-bits)
2. If new cipher key is to be given, then header is followed by cipher key. Key is not returned in output packet.
3. The rest is input data to be encrypted/ decrypted for input packet and output data for output packet.

Header format of wrapper is shown below:

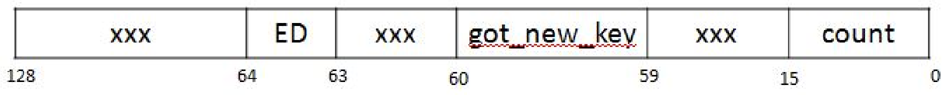


Figure 4.6: Header format of AES Wrapper

The header format of wrapper is similar to the engine command format. The terms in the header have same meaning as in engine command.

Chapter 5

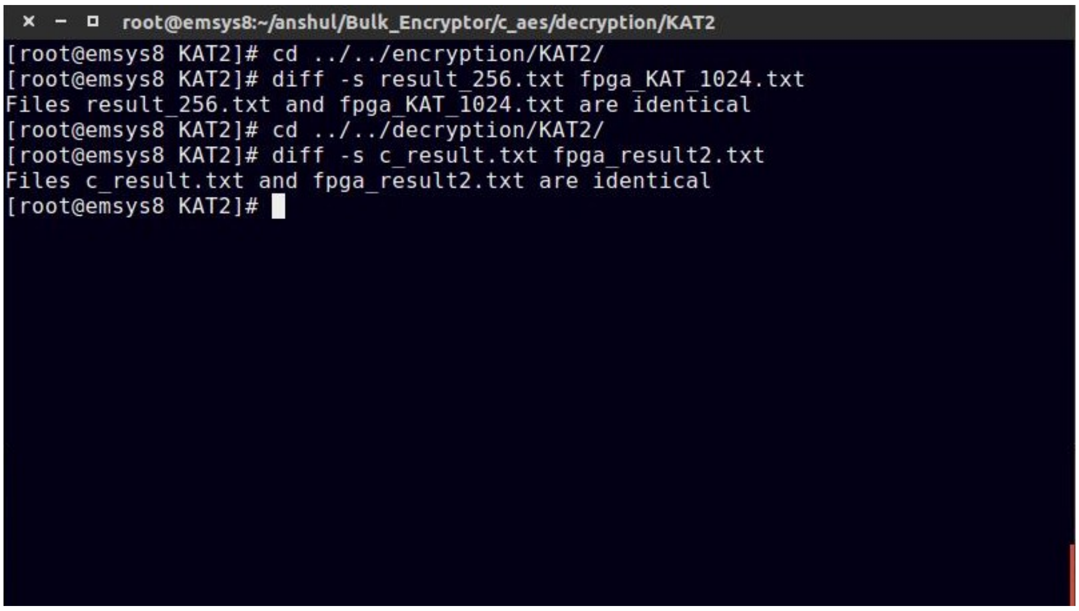
Testing and Results

5.1 AES algorithm Validation

The design it tested using Advanced Encryption Standard Algorithm Validation Suite (AESAVS) designed by NIST. AESAVS is designed to perform automated testing on Implementations Under Test (IUTs). AESAVS contains the tests to validate the AES encryption implementations.

A C code of AES-128 encryption and decryption is developed. The C code encryption is tested with the Known Answer Tests (KATs) and Monte Carlo Test (MCT). The tests results of the AES C code matches with the results given in the AESAVS. This validating the C code for AES-128 encryption. For testing the AES decryption C code, the KATs are applied on the encryption-decryption loopback system (i.e. the output of AES encryption is given as the input to AES decryption function). The tests output of loopback system matches matches the tests inputs, thus validating the AES-128 decryption C code.

For validating the algorithm of FPGA implementation, the random input test cases, KATs and MCT are performed on the FPGA implementation of AES engine and the AESAVS validated C code. The test results for both encryption and decryption on our FPGA implementation matches with the results of C code. This validated the AES-128 encryption and decryption algorithm of our FPGA implementation.



```
× − ◘  root@emsys8:~/anshul/Bulk_Encryptor/c_aes/decryption/KAT2
[root@emsys8 KAT2]# cd ../../encryption/KAT2/
[root@emsys8 KAT2]# diff -s result_256.txt fpga_KAT_1024.txt
Files result_256.txt and fpga_KAT_1024.txt are identical
[root@emsys8 KAT2]# cd ../../decryption/KAT2/
[root@emsys8 KAT2]# diff -s c_result.txt fpga_result2.txt
Files c_result.txt and fpga_result2.txt are identical
[root@emsys8 KAT2]# █
```

Figure 5.1: Screenshot of result

5.2 Resource Utilization Report

The hardware resources utilized by the FPGA implementation of AES-128 are tabulated below:

Site Type	Used	Available	Utilization (%)
Slice LUTs	40237	203800	19.74
Slice Registers	27888	407600	6.84
F7 Muxes	3037	101900	2.98
F8 Muxes	1335	50950	2.62

Table 5.1: Slice logic Utilization

Site Type	Used	Available	Utilization (%)
Slice	13092	50950	25.69
LUT as Logic	38708	203800	18.99
LUT as Memory	1529	64000	2.38
LUT Flip Flop Pairs	45992	203800	22.56
Unique Control Sets	402	-	-
Minimum number of registers lost to control set restriction	872 (lost)	-	-

Table 5.2: Slice logic Distribution

5.3 Throughput

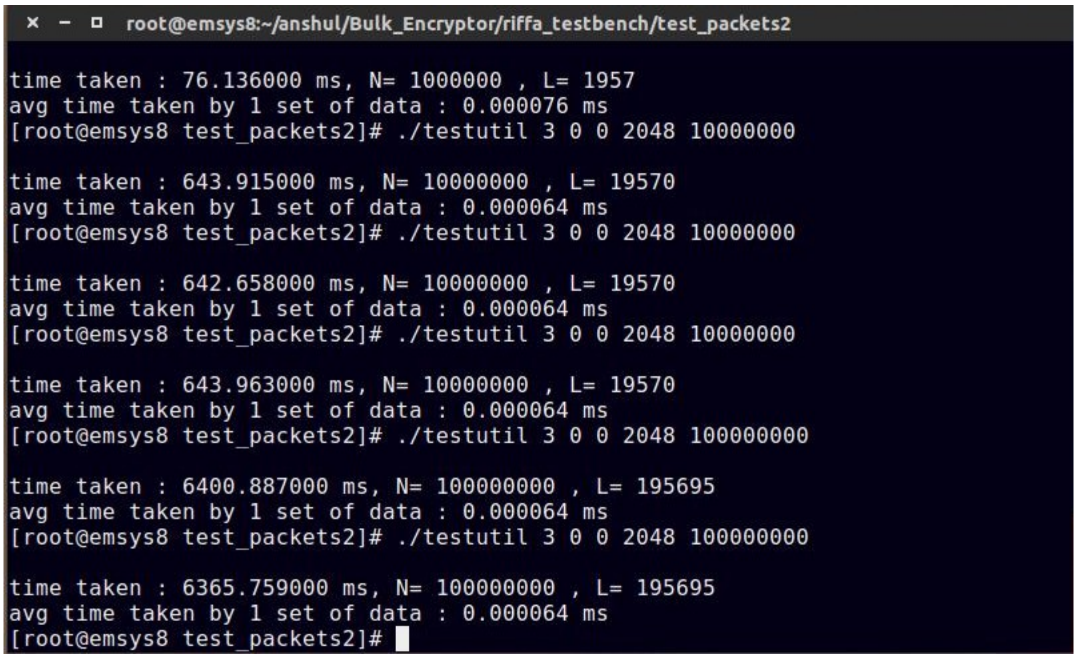
Due to the Riffa restrictions, the engine is synthesized to work at 250 MHz. The table 5.3 shows the time taken by FPGA implementation of AES-128 engine to process bulk data.

Number of blocks processed (128 bits)	Average Time taken (μs)
10,000	0.084
100,000	0.077
1,000,000	0.072
10,000,000	0.065
100,000,000	0.064

Table 5.3: Time Consumption in AES-128 encryption/decryption by the designed engine

From the table 5.3 it can be seen that our FPGA implementation takes average time of 0.64 microseconds, which accounts for **2 Gbps** processing rate at large inputs. The average processing time decreases as the input size increases. This is because the time taken by Riffa to communicate input and output data between FPGA and CPU (through PCIe slot) is significant. The significance of extra time taken by Riffa reduces as the input data size increases.

From table:3.2, the AJIT software implementation takes around 1,400 cycle for 128 bit encryption. Assuming that the AJIT processor runs at 1 GHz, its throughput will be around 90 Mbps. Thus, our bulk encryption engine, even though running at a 4 times slower clock frequency, is more that 20 times faster that the AJIT software implementation.


root@emsys8:~/anshul/Bulk_Encryptor/riffa_testbench/test_packets2

```
time taken : 76.136000 ms, N= 1000000 , L= 1957
avg time taken by 1 set of data : 0.000076 ms
[root@emsys8 test_packets2]# ./testutil 3 0 0 2048 10000000

time taken : 643.915000 ms, N= 10000000 , L= 19570
avg time taken by 1 set of data : 0.000064 ms
[root@emsys8 test_packets2]# ./testutil 3 0 0 2048 10000000

time taken : 642.658000 ms, N= 10000000 , L= 19570
avg time taken by 1 set of data : 0.000064 ms
[root@emsys8 test_packets2]# ./testutil 3 0 0 2048 10000000

time taken : 643.963000 ms, N= 10000000 , L= 19570
avg time taken by 1 set of data : 0.000064 ms
[root@emsys8 test_packets2]# ./testutil 3 0 0 2048 100000000

time taken : 6400.887000 ms, N= 100000000 , L= 195695
avg time taken by 1 set of data : 0.000064 ms
[root@emsys8 test_packets2]# ./testutil 3 0 0 2048 100000000

time taken : 6365.759000 ms, N= 100000000 , L= 195695
avg time taken by 1 set of data : 0.000064 ms
[root@emsys8 test_packets2]# 
```

Figure 5.2: Screenshot of result (throughput)

Chapter 6

Conclusion and Future Work

Software implementation of AES-128 on AJIT is done efficiently. Its performance is found to be nearly same as the performance of AES-128 implementation on other 32-bit processors.

The Bulk encryption engine is successfully implemented on FPGA Kintex KC705 evaluation board. The engine is capable of performing AES-128 encryption and decryption in ECB mode. The engine is working at a clock frequency of 250 Mhz. The engine is processing the data at a rate of 2 Gbps, which is more than 20 times faster than the software implantation on AJIT processor (assuming AJIT processor is running at 1 GHz frequency).

Currently, the designed AES bulk encryption engine performs encryption in ECB mode only. The other modes of AES (Cipher Block Chaining, Cipher Feedback, Counter, etc) are required be implemented in the engine. Support for 192-bit and 256-bit key can be introduced in the engine. After that the engine is needed to be converted into a co-processor for the AJIT processor.

Bibliography

[1] SPARC International, Inc., *The SPARC Architecture Manual, Version 8.* Revision SAV080SI9308, 1992.

[2] National Institute of Standards and Technology. *Federal Information Processing Standards Publication 197.*, 2001

[3] Joan Daemen and Vincent Rijmen,"The design of Rijndael: AES — the Advanced Encryption Standard", *Springer-Verlag*, 2002

[4] G.Bertoni, L.Breveglieri, P.Fragneto, M.Macchetti, and S.Marchesin, "Efficient Software Implementation of AES on 32Bit Platforms", *CHES 2002*, LNCS 2523, pp. 159171, 2003.

[5] V. Fischer, M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware," *Proceedings of CHES 01*, pp. 8196, 2001.

[6] M.Dworkin, "Recommendation for Block Cipher Modes of Operation : Methods and Techniques", *NIST Special Publication* 80038A, 2001

[7] Lawrence E. Bassham III, "The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)", *NIST Publication*, 2002